HALCON

a product of MVTec

HALCON/HDevelop

Operator Reference (en)

HALCON 24.11 *Progress-Steady*

MVTec Software GmbH

Building Vision for Business

HALCON/HDevelop 24.11.1.0

More information about HALCON can be found at: `http://www.mvtec.com`

Contents

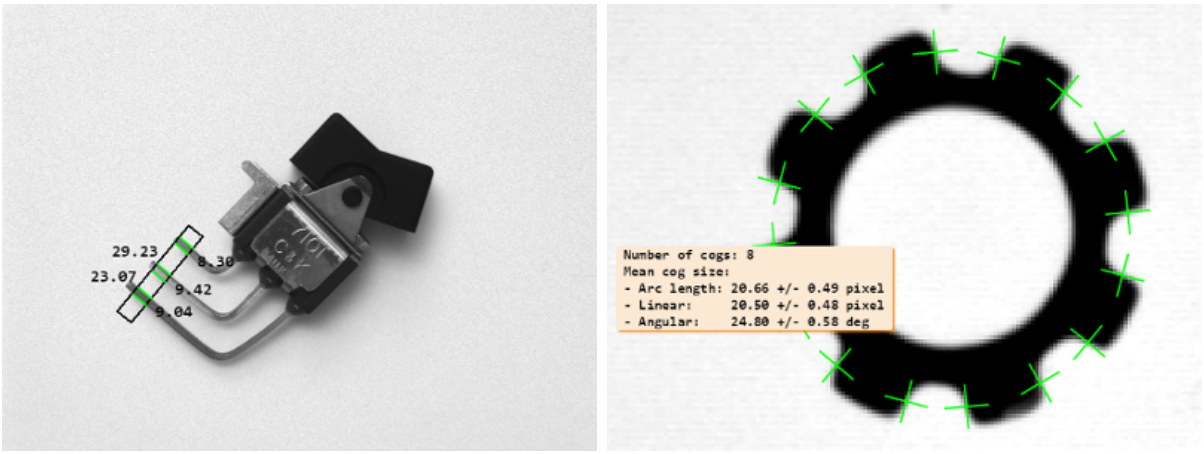
Chapter 1

1D Measuring

This chapter contains operators for 1D measuring.

Concept of 1D measuring

With 1D measuring, edges, i.e., transitions from light to dark or from dark to light, can be located along a predefined line or arc. This allows you to measure the dimension of parts fast and easily with high accuracy. Note that if you want to measure the dimensions of geometric primitives like circles, ellipses, rectangles, or lines, and approximate values for the positions, orientations, and geometric shapes are known, 2D Metrology may be a suitable alternative.



(1) (2)

Measure edges and the distances between them along a line (1) or along an arc (2). These images are from the example programs fuzzy_measure_pin.hdev and measure_ring.hdev.

In the following, the steps that are required to use 1D measuring are described briefly.

Generate measure object: First, a measure object must be generated that describes the region of interest for the measurement. If the measurement should be performed along a line, the measure object is defined by a rectangle. If it should be performed along an arc, the measure object is defined as an annular arc. The measure objects are generated by the operators

- `gen_measure_rectangle2` or
- `gen_measure_arc`.

Note that you can use shape-based matching (see chapter Matching / Shape-Based) to automatically align the measure objects.

Perform the measurement: Then, the actual measurement is performed. For this, typically one of the following operators is used:

- measure_pos extracts straight edges perpendicular to the main axis of the measure object and returns the positions of the edge centers, the edge amplitudes, and the distances between consecutive edges.
- measure_pairs extracts straight edge pairs perpendicular to the main axis of the measure object and returns the positions of the edge centers of the edge pairs, the edge amplitudes for the edge pairs, the distances between the edges of an edge pair, and the distances between consecutive edge pairs.
- measure_thresh extracts points with a particular gray value along the main axis of the measure object and returns their positions and the distances between consecutive points.

Alternatively, if there are extra edges that do not belong to the measurement, fuzzy measuring can be applied. Here, so-called fuzzy rules, which describe the features of good edges, must be defined. Possible features are, e.g., the position, the distance, the gray values, or the amplitude of edges. These functions are created with create_funct_1d_pairs and passed to the tool with set_fuzzy_measure or set_fuzzy_measure_norm_pair. Then, based on these rules, one of the following operators will extract the most appropriate edges:

- fuzzy_measure_pos extracts straight edges perpendicular to the main axis of the measure object and returns the positions of the edge centers, the edge amplitudes, the fuzzy scores, and the distances between consecutive edges.
- fuzzy_measure_pairs extracts straight edge pairs perpendicular to the main axis of the measure object and returns the positions of the first and second edges of the edge pairs, the edge amplitudes for the edge pairs, the positions of the centers of the edge pairs, the fuzzy scores, the distances between the edges of an edge pair, and the distances between consecutive edge pairs.
- fuzzy_measure_pairing is similar to fuzzy_measure_pairs with the exception that it is also possible to extract interleaving and included pairs using the parameter Pairing.

Alternatively to the automatic extraction of edges or points within the measure object, you can also extract a one-dimensional gray value profile perpendicular to the rectangle or annular arc and evaluate this gray value information according to your needs. The gray value profile within the measure object can be extracted with the operator

- measure_projection.

Destroy measure object handle: When you no longer need the measure object, you destroy it by passing the handle to

- close_measure.

Further operators

In addition to the operators mentioned above, you can use reset_fuzzy_measure to discard a fuzzy function of a fuzzy set that was set via set_fuzzy_measure or set_fuzzy_measure_norm_pair before, translate_measure to translate the reference point of the measure object to a specified position, write_measure and read_measure to write the measure object to file and read it from file again, and serialize_measure and deserialize_measure to serialize and deserialize the measure object.

Glossary

In the following, the most important terms that are used in the context of 1D Measuring are described.

measure object A data structure that contains a specific region of interest that is prepared for the extraction of straight edges which lie perpendicular to the major axis of a rectangle or an annular arc.

annular arc A circular arc with an associated width.

Further Information

See also the "Solution Guide Basics" and "Solution Guide on 1D Measuring" for further details about 1D Measuring.

Learn about 1D Measuring and many other topics in interactive online courses at our MVTec Academy .

close_measure (: : MeasureHandle :)

Delete a measure object.

close_measure deletes the measure object given by MeasureHandle. The memory used for the measure object is freed.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Parameters

▷ **MeasureHandle** (input_control) ..... measure ⇝ *handle*
Measure object handle.

Result

If the parameter values are correct the operator close_measure returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MeasureHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

gen_measure_rectangle2, gen_measure_arc, measure_pos, measure_pairs

Alternatives

clear_handle

See also

clear_handle

Module

1D Metrology

deserialize_measure (: : SerializedItemHandle : MeasureHandle)

Deserialize a serialized measure object.

deserialize_measure deserializes a measure object, that was serialized by serialize_measure (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized measure object is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created measure object with the handle MeasureHandle.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **MeasureHandle** (output_control) ..... measure ⇝ *handle*
Measure object handle.

Result

If the parameters are valid, the operator deserialize_measure returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_measure

Possible Successors

measure_pos, measure_pairs

See also

read_measure, write_measure

Module

1D Metrology

fuzzy_measure_pairing (Image : : MeasureHandle, Sigma, AmpThresh, FuzzyThresh, Transition, Pairing, NumPairs : RowEdgeFirst, ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond, ColumnEdgeSecond, AmplitudeSecond, RowPairCenter, ColumnPairCenter, FuzzyScore, IntraDistance)

Extract straight edge pairs perpendicular to a rectangle or an annular arc.

fuzzy_measure_pairing serves to extract *straight* edge pairs that lie *perpendicular* to the major axis of a rectangle or an annular arc.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

The extraction algorithm of fuzzy_measure_pairing is identical to fuzzy_measure_pairs (see there for details) with the exception, that it is also possible to extract interleaving and included pairs using the parameter Pairing. Currently only *'no_restriction'* is available, which returns all possible edge pairs, allowing interleaving and inclusion of pairs.

Only the best scored NumPairs edge pairs are returned, whereas 0 indicates to return all possible found edge combinations.

The selected edges are returned as single points, which lie on the major axis of the rectangle or annular arc. The corresponding edge amplitudes are returned in AmplitudeFirst and AmplitudeSecond, the fuzzy scores in FuzzyScore. In addition, the distance between each edge pair is returned in IntraDistance, corresponding to the distance between EdgeFirst[i] and EdgeSecond[i].

Attention

fuzzy_measure_pairing only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle or annular arc are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle or annular arc is as close to perpendicular as possible to the edges in the image. Additionally, Sigma must not become larger than approx. 0.5 * Length1 (for Length1 see gen_measure_rectangle2).

It should be kept in mind that fuzzy_measure_pairing ignores the domain of Image for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2 / real
Input image.

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **Sigma** (input_control) . . . . . number ⇝ *real*
Sigma of Gaussian smoothing.
Default: 1.0
Suggested values: Sigma ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ Sigma ≤ 100 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **AmpThresh** (input_control) ..... number ⇝ *real*
Minimum edge amplitude.
Default: 30.0
Suggested values: AmpThresh ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ AmpThresh ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **FuzzyThresh** (input_control) ..... number ⇝ *real*
Minimum fuzzy value.
Default: 0.5
Suggested values: FuzzyThresh ∈ {0.1, 0.3, 0.5, 0.7, 0.9}
Value range: 0.0 ≤ FuzzyThresh ≤ 1.0 (lin)
Recommended increment: 0.1

▷ **Transition** (input_control) ..... string ⇝ *string*
Select the first gray value transition of the edge pairs.
Default: 'all'
List of values: Transition ∈ {'all', 'positive', 'negative'}

▷ **Pairing** (input_control) ..... string ⇝ *string*
Constraint of pairing.
Default: 'no_restriction'
List of values: Pairing ∈ {'no_restriction'}

▷ **NumPairs** (input_control) ..... number ⇝ *integer*
Number of edge pairs.
Default: 10
Suggested values: NumPairs ∈ {0, 1, 10, 20, 50}
Value range: 0 ≤ NumPairs
Recommended increment: 1

▷ **RowEdgeFirst** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the first edge.

▷ **ColumnEdgeFirst** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the first edge.

▷ **AmplitudeFirst** (output_control) ..... real-array ⇝ *real*
Edge amplitude of the first edge (with sign).

▷ **RowEdgeSecond** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the second edge.

▷ **ColumnEdgeSecond** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the second edge.

▷ **AmplitudeSecond** (output_control) ..... real-array ⇝ *real*
Edge amplitude of the second edge (with sign).

▷ **RowPairCenter** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the center of the edge pair.

▷ **ColumnPairCenter** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the center of the edge pair.

▷ **FuzzyScore** (output_control) ..... real-array ⇝ *real*
Fuzzy evaluation of the edge pair.

▷ **IntraDistance** (output_control) ..... real-array ⇝ *real*
Distance between the edges of the edge pair.

Result

If the parameter values are correct the operator fuzzy_measure_pairing returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_measure_rectangle2, gen_measure_arc, set_fuzzy_measure

Possible Successors

close_measure

Alternatives

edges_sub_pix, fuzzy_measure_pairs, measure_pairs

See also

fuzzy_measure_pos, measure_pos

Module

1D Metrology

fuzzy_measure_pairs (Image : : MeasureHandle, Sigma, AmpThresh, FuzzyThresh, Transition : RowEdgeFirst, ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond, ColumnEdgeSecond, AmplitudeSecond, RowEdgeCenter, ColumnEdgeCenter, FuzzyScore, IntraDistance, InterDistance)

Extract straight edge pairs perpendicular to a rectangle or an annular arc.

fuzzy_measure_pairs serves to extract *straight* edge pairs which lie *perpendicular* to the major axis of a rectangle or an annular arc. In addition to measure_pairs it uses fuzzy functions to evaluate and select the edge pairs.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

The extraction algorithm of fuzzy_measure_pairs is identical to fuzzy_measure_pos. In addition, neighboring edges are grouped to pairs. To extract pairs that intersect or include each other, use fuzzy_measure_pairing.

If Transition = *'positive'*, the edge points with a dark-to-light transition in the direction of the major axis of the rectangle or annular arc are returned in RowEdgeFirst and ColumnEdgeFirst. In this case, the corresponding edges with a light-to-dark transition are returned in RowEdgeSecond and ColumnEdgeSecond. If Transition = *'negative'*, the behavior is exactly opposite. If Transition = *'all'*, the first detected edge defines the transition for RowEdgeFirst and ColumnEdgeFirst. I.e., dependent on the positioning of the measure object, edge pairs with a light-dark-light transition or edge pairs with a dark-light-dark transition are returned. This is suited, e.g., to measure objects with different brightness relative to the background.

Having extracted subpixel edge locations, the edges are paired. The pairing algorithm groups the edges such that interleavings and inclusions of pairs are prohibited. The features of an edge pair are evaluated by a fuzzy function, which can be set by set_fuzzy_measure or set_fuzzy_measure_norm_pair. Which edge pairs are selected can be determined with the parameter FuzzyThresh, which constitutes a threshold on the weight over all fuzzy sets, i.e., the geometric mean of the weights of the defined fuzzy functions.

The selected edges are returned as single points, which lie on the major axis of the rectangle or annular arc. The corresponding edge amplitudes are returned in AmplitudeFirst and AmplitudeSecond, the fuzzy scores in FuzzyScore. In addition, the distance between each edge pair is returned in IntraDistance and the distance between consecutive edge pairs is returned in InterDistance. Here, IntraDistance[i] corresponds to the distance between EdgeFirst[i] and EdgeSecond[i], while InterDistance[i] corresponds to the distance between EdgeSecond[i] and EdgeFirst[i+1], i.e., the tuple InterDistance contains one element less than the tuples of the edge pairs.

Attention

fuzzy_measure_pairs only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle or annular arc are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle or a annular arc is as close to perpendicular as possible to the edges in the image. Additionally, Sigma must not become larger than approx. 0.5 * Length1 (for Length1 see gen_measure_rectangle2).

It should be kept in mind that fuzzy_measure_pairs ignores the domain of Image for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2 / real
Input image.

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **Sigma** (input_control) . . . . . number ⇝ *real*
Sigma of Gaussian smoothing.
Default: 1.0
Suggested values: `Sigma` ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ `Sigma` ≤ 100 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **AmpThresh** (input_control) . . . . . number ⇝ *real*
Minimum edge amplitude.
Default: 30.0
Suggested values: `AmpThresh` ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ `AmpThresh` ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **FuzzyThresh** (input_control) . . . . . number ⇝ *real*
Minimum fuzzy value.
Default: 0.5
Suggested values: `FuzzyThresh` ∈ {0.1, 0.3, 0.5, 0.7, 0.9}
Value range: 0.0 ≤ `FuzzyThresh` ≤ 1.0 (lin)
Recommended increment: 0.1

▷ **Transition** (input_control) . . . . . string ⇝ *string*
Select the first gray value transition of the edge pairs.
Default: 'all'
List of values: `Transition` ∈ {'all', 'positive', 'negative'}

▷ **RowEdgeFirst** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinate of the first edge point.

▷ **ColumnEdgeFirst** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinate of the first edge point.

▷ **AmplitudeFirst** (output_control) . . . . . real-array ⇝ *real*
Edge amplitude of the first edge (with sign).

▷ **RowEdgeSecond** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinate of the second edge point.

▷ **ColumnEdgeSecond** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinate of the second edge point.

▷ **AmplitudeSecond** (output_control) . . . . . real-array ⇝ *real*
Edge amplitude of the second edge (with sign).

▷ **RowEdgeCenter** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinate of the center of the edge pair.

▷ **ColumnEdgeCenter** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinate of the center of the edge pair.

▷ **FuzzyScore** (output_control) . . . . . real-array ⇝ *real*
Fuzzy evaluation of the edge pair.

▷ **IntraDistance** (output_control) . . . . . real-array ⇝ *real*
Distance between edges of an edge pair.

▷ **InterDistance** (output_control) . . . . . real-array ⇝ *real*
Distance between consecutive edge pairs.

Result

If the parameter values are correct the operator `fuzzy_measure_pairs` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_measure_rectangle2, gen_measure_arc, set_fuzzy_measure

Possible Successors

close_measure

Alternatives

edges_sub_pix, fuzzy_measure_pairing, measure_pairs

See also

fuzzy_measure_pos, measure_pos

Module

1D Metrology

fuzzy_measure_pos (Image : : MeasureHandle, Sigma, AmpThresh, FuzzyThresh, Transition : RowEdge, ColumnEdge, Amplitude, FuzzyScore, Distance)

Extract straight edges perpendicular to a rectangle or an annular arc.

fuzzy_measure_pos extracts *straight* edges which lie *perpendicular* to the major axis of a rectangle or an annular arc. In addition to measure_pos it uses fuzzy functions to evaluate and select the edges.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

The algorithm of fuzzy_measure_pos works by averaging the gray values in "slices" perpendicular to the major axis of the rectangle or annular arc in order to obtain a one-dimensional edge profile. The sampling is done at subpixel positions in the image Image at integer row and column distances (in the coordinate frame of the rectangle) from the center of the rectangle. Since this involves some calculations which can be used repeatedly in several measurements, the operator gen_measure_rectangle2 is used to perform these calculations only once, thus increasing the speed of fuzzy_measure_pos significantly. Since there is a trade-off between accuracy and speed in the subpixel calculations of the gray values, and thus in the accuracy of the extracted edge positions, different interpolation schemes can be selected in gen_measure_rectangle2. (The interpolation only influences rectangles not aligned with the image axes and annular arcs.) The measure object generated with gen_measure_rectangle2 is passed in MeasureHandle.

After the one-dimensional edge profile has been calculated, subpixel edge locations are computed by convolving the profile with the derivatives of a Gaussian smoothing kernel of standard deviation Sigma. Salient edges can be selected with the parameter AmpThresh, which constitutes a threshold on the amplitude, i.e., the absolute value of the first derivative of the edge. Additionally, it is possible to select only positive edges, i.e., edges which constitute a dark-to-light transition in the direction of the major axis of the rectangle (Transition = *'positive'*), only negative edges, i.e., light-to-dark transitions (Transition = *'negative'*), or both types of edges (Transition = *'all'*). Finally, it is possible to select which edge points are returned.

Having extracted subpixel edge locations, features of these edges are evaluated by a corresponding fuzzy function, which can be set by set_fuzzy_measure. Which edges are selected can be determined with the parameter FuzzyThresh, which constitutes a threshold on the weight over all fuzzy sets, i.e., the geometric mean of the weights of the defined sets.

The selected edges are returned as single points, which lie on the major axis of the rectangle or annular arc, in (RowEdge, ColumnEdge). The corresponding edge amplitudes are returned in Amplitude, the fuzzy scores in FuzzyScore. In addition, the distance between consecutive edge points is returned in Distance. Here, Distance[i] corresponds to the distance between Edge[i] and Edge[i+1], i.e., the tuple Distance contains one element less than the tuples RowEdge and ColumnEdge.

Attention

fuzzy_measure_pos only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle is as close to perpendicular as possible

to the edges in the image. Additionally, Sigma must not become larger than approx. 0.5 * Length1 (for Length1 see gen_measure_rectangle2).

It should be kept in mind that fuzzy_measure_pos ignores the domain of Image for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
Input image.

▷ **MeasureHandle** (input_control) ..... measure ⇝ *handle*
Measure object handle.

▷ **Sigma** (input_control) ..... number ⇝ *real*
Sigma of Gaussian smoothing.
Default: 1.0
Suggested values: Sigma ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ Sigma ≤ 100 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **AmpThresh** (input_control) ..... number ⇝ *real*
Minimum edge amplitude.
Default: 30.0
Suggested values: AmpThresh ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ AmpThresh ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **FuzzyThresh** (input_control) ..... number ⇝ *real*
Minimum fuzzy value.
Default: 0.5
Suggested values: FuzzyThresh ∈ {0.1, 0.3, 0.5, 0.6, 0.7, 0.9}
Value range: 0.0 ≤ FuzzyThresh ≤ 1.0 (lin)
Recommended increment: 0.1

▷ **Transition** (input_control) ..... string ⇝ *string*
Select light/dark or dark/light edges.
Default: 'all'
List of values: Transition ∈ {'all', 'positive', 'negative'}

▷ **RowEdge** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the edge point.

▷ **ColumnEdge** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the edge point.

▷ **Amplitude** (output_control) ..... real-array ⇝ *real*
Edge amplitude of the edge (with sign).

▷ **FuzzyScore** (output_control) ..... real-array ⇝ *real*
Fuzzy evaluation of the edges.

▷ **Distance** (output_control) ..... real-array ⇝ *real*
Distance between consecutive edges.

Result

If the parameter values are correct the operator fuzzy_measure_pos returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_measure_rectangle2, gen_measure_arc, set_fuzzy_measure

Possible Successors

close_measure

Alternatives

edges_sub_pix, measure_pos

See also

fuzzy_measure_pairing, fuzzy_measure_pairs, measure_pairs

Module

1D Metrology

gen_measure_arc (: : CenterRow, CenterCol, Radius, AngleStart, AngleExtent, AnnulusRadius, Width, Height, Interpolation : MeasureHandle)

Prepare the extraction of straight edges perpendicular to an annular arc.

gen_measure_arc prepares the extraction of *straight* edges which lie *perpendicular* to an annular arc. Here, annular arc denotes a circular arc with an associated width. The center of the arc is passed in the parameters CenterRow and CenterCol, its radius in Radius, the starting angle in AngleStart, and its angular extent relative to the starting angle in AngleExtent. If AngleExtent > 0, an arc with counterclockwise orientation is generated, otherwise an arc with clockwise orientation. The radius of the annular arc, i.e., half its width, is determined by AnnulusRadius.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

The edge extraction algorithm is described in the documentation of the operator measure_pos. As discussed there, different types of interpolation can be used for the calculation of the one-dimensional gray value profile. For Interpolation = *'nearest_neighbor'*, the gray values in the measurement are obtained from the gray values of the closest pixel, i.e., by constant interpolation. For Interpolation = *'bilinear'*, bilinear interpolation is used, while for Interpolation = *'bicubic'*, bicubic interpolation is used.

To perform the actual measurement at optimal speed, all computations that can be used for multiple measurements are already performed in the operator gen_measure_arc. For this, an optimized data structure, a so-called measure object, is constructed and returned in MeasureHandle. The size of the images in which measurements will be performed must be specified in the parameters Width and Height.

The system parameter *'int_zooming'* (see set_system) affects the accuracy and speed of the calculations used to construct the measure object. If *'int_zooming'* is set to *'true'*, the internal calculations are performed using fixed point arithmetic, leading to much shorter execution times. However, the geometric accuracy is slightly lower in this mode. If *'int_zooming'* is set to *'false'*, the internal calculations are performed using floating point arithmetic, leading to the maximum geometric accuracy, but also to significantly increased execution times.

Attention

Note that when using bilinear or bicubic interpolation, not only the measurement rectangle but additionally the margin around the rectangle must fit into the image. The width of the margin (in all four directions) must be at least one pixel for bilinear interpolation and two pixels for bicubic interpolation. For projection lines that do not fulfill this condition, no gray value is computed. Thus, no edge can be extracted at these positions.

Please also note that the center coordinates of the arc are rounded internally, so that the center lies on the pixel grid. This is done to ensure consistency.

Parameters

▷ **CenterRow** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the center of the arc.
Default: 100.0
Suggested values: CenterRow ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Value range: 0.0 ≤ CenterRow (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **CenterCol** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the center of the arc.
Default: 100.0
Suggested values: CenterCol ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Value range: 0.0 ≤ CenterCol (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Radius** (input_control) . . . . . number ⇝ *real* / integer
Radius of the arc.
Default: 50.0
Suggested values: Radius ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: AnnulusRadius <= Radius

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real* / integer
Start angle of the arc in radians.
Default: 0.0
Suggested values: AngleStart ∈ {-3.14159, -2.35619, -1.57080, -0.78540, 0.0, 0.78540, 1.57080, 2.35619, 3.14159}
Value range: -3.14159 ≤ AngleStart ≤ 3.14159 (lin)
Minimum increment: 0.03142
Recommended increment: 0.31416

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real* / integer
Angular extent of the arc in radians.
Default: 6.28318
Suggested values: AngleExtent ∈ {-6.28318, -5.49779, -4.71239, -3.92699, -3.14159, -2.35619, -1.57080, -0.78540, 0.78540, 1.57080, 2.35619, 3.14159, 3.92699, 4.71239, 5.49779, 6.28318}
Value range: -6.28318 ≤ AngleExtent ≤ 6.28318 (lin)
Minimum increment: 0.03142
Recommended increment: 0.31416

▷ **AnnulusRadius** (input_control) . . . . . number ⇝ *real* / integer
Radius (half width) of the annulus.
Default: 10.0
Suggested values: AnnulusRadius ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: AnnulusRadius > 0

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the image to be processed subsequently.
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768}
Value range: 0 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 16

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the image to be processed subsequently.
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576}
Value range: 0 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 16

▷ **Interpolation** (input_control) . . . . . string ⇝ *string*
Type of interpolation to be used.
Default: 'nearest_neighbor'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear', 'bicubic'}

▷ **MeasureHandle** (output_control) . . . . . measure ⇝ *handle*
Measure object handle.

Result

If the parameter values are correct, the operator `gen_measure_arc` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`draw_circle`

Possible Successors

`measure_pos`, `measure_pairs`, `fuzzy_measure_pos`, `fuzzy_measure_pairs`, `fuzzy_measure_pairing`

Alternatives

`edges_sub_pix`

See also

`gen_measure_rectangle2`

Module

1D Metrology

gen_measure_rectangle2 (: : Row, Column, Phi, Length1, Length2, Width, Height, Interpolation : MeasureHandle)

Prepare the extraction of straight edges perpendicular to a rectangle.

`gen_measure_rectangle2` prepares the extraction of *straight* edges which lie *perpendicular* to the major axis of a rectangle. The center of the rectangle is passed in the parameters `Row` and `Column`, the direction of the major axis of the rectangle in `Phi`, and the length of the two axes, i.e., half the diameter of the rectangle, in `Length1` and `Length2`.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

The edge extraction algorithm is described in the documentation of the operator `measure_pos`. As discussed there, different types of interpolation can be used for the calculation of the one-dimensional gray value profile. For `Interpolation` = *'nearest_neighbor'*, the gray values in the measurement are obtained from the gray values of the closest pixel, i.e., by constant interpolation. For `Interpolation` = *'bilinear'*, bilinear interpolation is used, while for `Interpolation` = *'bicubic'*, bicubic interpolation is used.

To perform the actual measurement at optimal speed, all computations that can be used for multiple measurements are already performed in the operator `gen_measure_rectangle2`. For this, an optimized data structure, a so-called measure object, is constructed and returned in `MeasureHandle`. The size of the images in which measurements will be performed must be specified in the parameters `Width` and `Height`.

The system parameter *'int_zooming'* (see `set_system`) affects the accuracy and speed of the calculations used to construct the measure object. If *'int_zooming'* is set to *'true'*, the internal calculations are performed using fixed point arithmetic, leading to much shorter execution times. However, the geometric accuracy is slightly lower in this mode. If *'int_zooming'* is set to *'false'*, the internal calculations are performed using floating point arithmetic, leading to the maximum geometric accuracy, but also to significantly increased execution times.

Attention

Note that when using bilinear or bicubic interpolation, not only the measurement rectangle but additionally the margin around the rectangle must fit into the image. The width of the margin (in all four directions) must be at least one pixel for bilinear interpolation and two pixels for bicubic interpolation. For projection lines that do not fulfill this condition, no gray value is computed. Thus, no edge can be extracted at these positions.

Please also note that the center coordinates of the rectangle are rounded internally, so that the center lies on the pixel grid. This is done to ensure consistency.

Parameters

▷ **Row** (input_control) . . . . . rectangle2.center.y ⇝ *real* / integer
Row coordinate of the center of the rectangle.
Default: 300.0
Suggested values: Row ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Value range: 0.0 ≤ Row ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Column** (input_control) . . . . . rectangle2.center.x ⇝ *real* / integer
Column coordinate of the center of the rectangle.
Default: 200.0
Suggested values: Column ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Value range: 0.0 ≤ Column ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Phi** (input_control) . . . . . rectangle2.angle.rad ⇝ *real* / integer
Angle of longitudinal axis of the rectangle to horizontal (radians).
Default: 0.0
Suggested values: Phi ∈ {-1.178097, -0.785398, -0.392699, 0.0, 0.392699, 0.785398, 1.178097}
(lin)
Minimum increment: 0.001
Recommended increment: 0.1

▷ **Length1** (input_control) . . . . . rectangle2.hwidth ⇝ *real* / integer
Half width of the rectangle.
Default: 100.0
Suggested values: Length1 ∈ {3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0, 300.0, 500.0}
Value range: 1.0 ≤ Length1 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Length2** (input_control) . . . . . rectangle2.hheight ⇝ *real* / integer
Half height of the rectangle.
Default: 20.0
Suggested values: Length2 ∈ {1.0, 2.0, 3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0}
Value range: 0.0 ≤ Length2 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the image to be processed subsequently.
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768}
Value range: 0 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 16

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the image to be processed subsequently.
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576}
Value range: 0 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 16

▷ **Interpolation** (input_control) . . . . . string ⇝ *string*
Type of interpolation to be used.
Default: 'nearest_neighbor'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear', 'bicubic'}

▷ **MeasureHandle** (output_control) . . . . . measure ⇝ *handle*
Measure object handle.

Result

If the parameter values are correct the operator gen_measure_rectangle2 returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

draw_rectangle2

Possible Successors

measure_pos, measure_pairs, fuzzy_measure_pos, fuzzy_measure_pairs, fuzzy_measure_pairing, measure_thresh

Alternatives

edges_sub_pix

See also

gen_measure_arc

Module

1D Metrology

get_measure_param (: : MeasureHandle, GenParamName : GenParamValue)

Return the parameters and properties of a measure object.

The operator get_measure_param returns parameters and properties of the measure object MeasureHandle. The names of the desired properties are passed in the generic parameter GenParamName, the corresponding values are returned in GenParamValue.

The properties that can be passed to GenParamName depend on the kind of measure object as well as its parameters. If a property is not available, get_measure_param returns an error.

Properties for all measure objects

- *'type'*: Type of the measure object, either *'rectangle2'* if the object was created with gen_measure_rectangle2, or *'arc'* if it was created with gen_measure_arc.
- *'image_width'*, *'image_height'*: Image width and height, respectively, for which the measure object was created.
- *'interpolation'*: Used interpolation mode: *'nearest_neighbor'*, *'bilinear'* or *'bicubic'*.

Properties for rectangular measure objects

Properties for measure objects that were created with gen_measure_rectangle2.

- *'row'*, *'column'*: Row and column, respectively, of the center of the measurement rectangle.
- *'phi'*: Rotation angle of the measurement rectangle.
- *'length1'*, *'length2'*: Side lengths of the measurement rectangle.

Properties for annular-shaped measure objects

Properties for measure objects that were created with gen_measure_arc.

- *'row'*, *'column'*: Row and column, respectively, of the center of the annular arc.
- *'radius'*: Radius of the annular arc.

- *'angle_start'*, *'angle_extent'*: Starting angle and angular extent of annular arc.
- *'annulus_radius'*: Radius of the angular arc.

Properties for measure objects with fuzzy functions

Properties for measure objects, for which fuzzy functions have been set with `set_fuzzy_measure` or `set_fuzzy_measure_norm_pair`.

- *'fuzzy_contrast'*: Fuzzy function for evaluation of the edge amplitudes.
- *'fuzzy_position'*, *'fuzzy_position_center'*, *'fuzzy_position_end'*, *'fuzzy_position_first_edge'*, *'fuzzy_position_last_edge'*: Fuzzy function for evaluation of the distance of edge candidates to the reference point on the measure object.
- *'fuzzy_position_pair'*, *'fuzzy_position_pair_center'*, *'fuzzy_position_pair_end'*, *'fuzzy_position_first_pair'*, *'fuzzy_position_last_pair'*: Fuzzy function for evaluation of of the distance of edge pairs to the reference point on the measure object.
- *'fuzzy_size'*, *'fuzzy_size_diff'*, *'fuzzy_size_abs_diff'*: Fuzzy function for evaluation of the distance between two edges of a pair.
- *'fuzzy_gray'*: Fuzzy function for weighting the mean projected gray value between two edges of a pair.

Parameters

▷ **`MeasureHandle`** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the parameter to be returned.
Default: 'type'
List of values: `GenParamName` ∈ {'type', 'image_width', 'image_height', 'interpolation', 'row', 'column', 'phi', 'length1', 'length2', 'radius', 'angle_start', 'angle_extent', 'annulus_radius', 'fuzzy_contrast', 'fuzzy_gray', 'fuzzy_position', 'fuzzy_position_center', 'fuzzy_position_end', 'fuzzy_position_first_edge', 'fuzzy_position_last_edge', 'fuzzy_position_pair', 'fuzzy_position_pair_center', 'fuzzy_position_pair_end', 'fuzzy_position_first_pair', 'fuzzy_position_last_pair', 'fuzzy_size', 'fuzzy_size_diff', 'fuzzy_size_abs_diff'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.value(-array) ⇝ *real* / string / integer
Value of the parameter.

Result

If the parameter values are correct the operator `get_measure_param` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_measure_rectangle2`, `gen_measure_arc`

See also

`gen_measure_rectangle2`, `gen_measure_arc`, `translate_measure`

Module

1D Metrology

```
measure_pairs ( Image : : MeasureHandle, Sigma, Threshold,
    Transition, Select : RowEdgeFirst, ColumnEdgeFirst,
    AmplitudeFirst, RowEdgeSecond, ColumnEdgeSecond, AmplitudeSecond,
    IntraDistance, InterDistance )
```

Extract straight edge pairs perpendicular to a rectangle or annular arc.

measure_pairs serves to extract *straight* edge pairs which lie *perpendicular* to the major axis of a rectangle or annular arc.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

The extraction algorithm of measure_pairs is identical to measure_pos. In addition the edges are grouped to pairs: If Transition = *'positive'*, the edge points with a dark-to-light transition in the direction of the major axis of the rectangle are returned in RowEdgeFirst and ColumnEdgeFirst. In this case, the corresponding edges with a light-to-dark transition are returned in RowEdgeSecond and ColumnEdgeSecond. If Transition = *'negative'*, the behavior is exactly opposite. If Transition = *'all'*, the first detected edge defines the transition for RowEdgeFirst and ColumnEdgeFirst. I.e., dependent on the positioning of the measure object, edge pairs with a light-dark-light transition or edge pairs with a dark-light-dark transition are returned. This is suited, e.g., to measure objects with different brightness relative to the background.

If more than one consecutive edge with the same transition is found, the first one is used as a pair element. This behavior may cause problems in applications in which the threshold Threshold cannot be selected high enough to suppress consecutive edges of the same transition. For these applications, a second pairing mode exists that only selects the respective strongest edges of a sequence of consecutive rising and falling edges. This mode is selected by appending *'_strongest'* to any of the above modes for Transition, e.g., *'negative_strongest'*. Finally, it is possible to select which edge pairs are returned. If Select is set to *'all'*, all edge pairs are returned. If it is set to *'first'*, only the first of the extracted edge pairs is returned, while it is set to *'last'*, only the last one is returned.

The extracted edges are returned as single points which lie on the major axis of the rectangle. The corresponding edge amplitudes are returned in AmplitudeFirst and AmplitudeSecond. In addition, the distance between each edge pair is returned in IntraDistance and the distance between consecutive edge pairs is returned in InterDistance. Here, IntraDistance[i] corresponds to the distance between EdgeFirst[i] and EdgeSecond[i], while InterDistance[i] corresponds to the distance between EdgeSecond[i] and EdgeFirst[i+1], i.e., the tuple InterDistance contains one element less than the tuples of the edge pairs.

Attention

measure_pairs only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle is as close to perpendicular as possible to the edges in the image. Additionally, Sigma must not become larger than approx. 0.5 * Length1 (for Length1 see gen_measure_rectangle2).

It should be kept in mind that measure_pairs ignores the domain of Image for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2 / real
Input image.

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **Sigma** (input_control) . . . . . number ⇝ *real*
Sigma of gaussian smoothing.
Default: 1.0
Suggested values: Sigma ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ Sigma ≤ 100 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **Threshold** (input_control) . . . . . number ⇝ *real*
Minimum edge amplitude.
Default: 30.0
Suggested values: Threshold ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ Threshold ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **Transition** (input_control) . . . . . string ⇝ *string*
Type of gray value transition that determines how edges are grouped to edge pairs.
Default: 'all'
List of values: Transition ∈ {'all', 'positive', 'negative', 'all_strongest', 'positive_strongest',

'negative_strongest'}

▷ **Select** (input_control) ..... string ⇝ *string*
Selection of edge pairs.
Default: 'all'
List of values: Select ∈ {'all', 'first', 'last'}

▷ **RowEdgeFirst** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the center of the first edge.

▷ **ColumnEdgeFirst** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the center of the first edge.

▷ **AmplitudeFirst** (output_control) ..... real-array ⇝ *real*
Edge amplitude of the first edge (with sign).

▷ **RowEdgeSecond** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the center of the second edge.

▷ **ColumnEdgeSecond** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the center of the second edge.

▷ **AmplitudeSecond** (output_control) ..... real-array ⇝ *real*
Edge amplitude of the second edge (with sign).

▷ **IntraDistance** (output_control) ..... real-array ⇝ *real*
Distance between edges of an edge pair.

▷ **InterDistance** (output_control) ..... real-array ⇝ *real*
Distance between consecutive edge pairs.

Result

If the parameter values are correct the operator measure_pairs returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_measure_rectangle2

Possible Successors

close_measure

Alternatives

edges_sub_pix, fuzzy_measure_pairs, fuzzy_measure_pairing

See also

measure_pos, fuzzy_measure_pos

Module

1D Metrology

measure_pos (Image : : MeasureHandle, Sigma, Threshold, Transition, Select : RowEdge, ColumnEdge, Amplitude, Distance)

Extract straight edges perpendicular to a rectangle or annular arc.

measure_pos extracts *straight* edges which lie *perpendicular* to the major axis of a rectangle or annular arc.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

The algorithm of measure_pos works by averaging the gray values in "slices" perpendicular to the major axis of the rectangle or annular arc in order to obtain a one-dimensional edge profile. The sampling is done at sub-pixel positions in the image Image at integer row and column distances (in the coordinate frame of the rectangle) from the center of the rectangle. Since this involves some calculations which can be used repeatedly in several measurements, the operator gen_measure_rectangle2 or gen_measure_arc is used to perform

these calculations only once, thus increasing the speed of measure_pos significantly. Since there is a trade-off between accuracy and speed in the subpixel calculations of the gray values, and thus in the accuracy of the extracted edge positions, different interpolation schemes can be selected in gen_measure_rectangle2. (The interpolation only influences rectangles not aligned with the image axes.) The measure object generated with gen_measure_rectangle2 is passed in MeasureHandle.

After the one-dimensional edge profile has been calculated, subpixel edge locations are computed by convolving the profile with the derivatives of a Gaussian smoothing kernel of standard deviation Sigma. Salient edges can be selected with the parameter Threshold, which constitutes a threshold on the amplitude values (Amplitude), i.e., the absolute value of the first derivative of the edge. Note that the amplitude values are scaled by the factor $\text{Sigma} \cdot \sqrt{2\pi}$. Additionally, it is possible to select only positive edges, i.e., edges which constitute a dark-to-light transition in the direction of the major axis of the rectangle or the arc (Transition = *'positive'*), only negative edges, i.e., light-to-dark transitions (Transition = *'negative'*), or both types of edges (Transition = *'all'*). Finally, it is possible to select which edge points are returned. If Select is set to *'all'*, all edge points are returned. If it is set to *'first'*, only the first of the extracted edge points is returned, while it is set to *'last'*, only the last one is returned.

The extracted edges are returned as single points which lie on the major axis of the rectangle or arc in (RowEdge, ColumnEdge). The corresponding edge amplitudes are returned in Amplitude. In addition, the distance between consecutive edge points is returned in Distance. Here, Distance[i] corresponds to the distance between Edge[i] and Edge[i+1], i.e., the tuple Distance contains one element less than the tuples RowEdge and ColumnEdge.

Attention

measure_pos only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle or arc are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle or arc is as close to perpendicular as possible to the edges in the image. Additionally, Sigma must not become larger than approx. 0.5 * Length1 (for Length1 see gen_measure_rectangle2).

It should be kept in mind that measure_pos ignores the domain of Image for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
Input image.

▷ **MeasureHandle** (input_control) ..... measure ⇝ *handle*
Measure object handle.

▷ **Sigma** (input_control) ..... number ⇝ *real*
Sigma of gaussian smoothing.
Default: 1.0
Suggested values: Sigma ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ Sigma ≤ 100 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **Threshold** (input_control) ..... number ⇝ *real*
Minimum edge amplitude.
Default: 30.0
Suggested values: Threshold ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ Threshold ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **Transition** (input_control) ..... string ⇝ *string*
Light/dark or dark/light edge.
Default: 'all'
List of values: Transition ∈ {'all', 'positive', 'negative'}

▷ **Select** (input_control) ..... string ⇝ *string*
Selection of end points.
Default: 'all'
List of values: Select ∈ {'all', 'first', 'last'}

▷ **RowEdge** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinate of the center of the edge.

▷ **ColumnEdge** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinate of the center of the edge.

▷ **Amplitude** (output_control) . . . . . real-array ⇝ *real*
Edge amplitude of the edge (with sign).

▷ **Distance** (output_control) . . . . . real-array ⇝ *real*
Distance between consecutive edges.

Result

If the parameter values are correct the operator `measure_pos` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_measure_rectangle2`

Possible Successors

`close_measure`

Alternatives

`edges_sub_pix`, `fuzzy_measure_pos`

See also

`measure_pairs`, `fuzzy_measure_pairs`, `fuzzy_measure_pairing`

Module

1D Metrology

measure_projection (Image : : MeasureHandle : GrayValues)

Extract a gray value profile perpendicular to a rectangle or annular arc.

`measure_projection` extracts a one-dimensional gray value profile perpendicular to a rectangle or annular arc. This is done by averaging the gray values in "slices" perpendicular to the major axis of the rectangle or arc. The sampling is done at subpixel positions in the image `Image` at integer row and column distances (in the coordinate frame of the rectangle) from the center of the rectangle. Since this involves some calculations which can be used repeatedly in several projections, the operator `gen_measure_rectangle2` is used to perform these calculations only once, thus increasing the speed of `measure_projection` significantly. Since there is a trade-off between accuracy and speed in the subpixel calculations of the gray values, different interpolation schemes can be selected in `gen_measure_rectangle2` (the interpolation only influences rectangles not aligned with the image axes). The measure object generated with `gen_measure_rectangle2` is passed in `MeasureHandle`.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Attention

It should be kept in mind that `measure_projection` ignores the domain of `Image` for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2 / real
Input image.

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **GrayValues** (output_control) . . . . . number-array ⇝ *real*
Gray value profile.

Result

If the parameter values are correct the operator `measure_projection` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_measure_rectangle2`

Possible Successors

`close_measure`

Alternatives

`gray_projections`

Module

1D Metrology

measure_thresh (Image : : MeasureHandle, Sigma, Threshold, Select : RowThresh, ColumnThresh, Distance)

Extracting points with a particular gray value along a rectangle or an annular arc.

`measure_thresh` extracts points for which the gray value within an one-dimensional gray value profile is equal to the specified threshold `Threshold`. The gray value profile is projected onto the major axis of the measure rectangle which is passed with the parameter `MeasureHandle`, so the threshold points calculated within the gray value profile correspond to certain image coordinates on the rectangle's major axis. These coordinates are returned as the operator results in `RowThresh` and `ColumnThresh`.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

If the gray value profile intersects the threshold line for several times, the parameter `Select` determines which values to return. Possible settings are *'first'*, *'last'*, *'first_last'* (first and last) or *'all'*. For the last two cases `Distance` returns the distances between the calculated points.

The gray value profile is created by averaging the gray values along all line segments, which are defined by the measure rectangle as follows:

1. The segments are perpendicular to the major axis of the rectangle,
2. they have an integer distance to the center of the rectangle,
3. the rectangle bounds the segments.

For every line segment, the average of the gray values of all points with an integer distance to the major axis is calculated. Due to translation and rotation of the measure rectangle with respect to the image coordinates the input image `Image` is in general sampled at subpixel positions.

Since this involves some calculations which can be used repeatedly in several projections, the operator `gen_measure_rectangle2` is used to perform these calculations only once in advance. Here, the measure object `MeasureHandle` is generated and different interpolation schemes can be selected.

Attention

`measure_thresh` only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle is as close to perpendicular as possible to the edges in the image. Additionally, `Sigma` must not become larger than approx. 0.5 * Length1 (for Length1 see `gen_measure_rectangle2`).

It should be kept in mind that `measure_thresh` ignores the domain of `Image` for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2 / real
Input image.

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **Sigma** (input_control) . . . . . number ⇝ *real*
Sigma of gaussian smoothing.
Default: 1.0
Suggested values: `Sigma` ∈ {0.0, 0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.0 ≤ `Sigma` ≤ 100 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **Threshold** (input_control) . . . . . number ⇝ *real*
Threshold.
Default: 128.0
Value range: 0 ≤ `Threshold` ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 1

▷ **Select** (input_control) . . . . . string ⇝ *string*
Selection of points.
Default: 'all'
List of values: `Select` ∈ {'all', 'first', 'last', 'first_last'}

▷ **RowThresh** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinates of points with threshold value.

▷ **ColumnThresh** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinates of points with threshold value.

▷ **Distance** (output_control) . . . . . real-array ⇝ *real*
Distance between consecutive points.

Result

If the parameter values are correct the operator `measure_thresh` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_measure_rectangle2`

Possible Successors

`close_measure`

Alternatives

`measure_pos`, `edges_sub_pix`, `measure_pairs`

Module

1D Metrology

read_measure (: : FileName : MeasureHandle)

Read a measure object from a file.

`read_measure` reads a measure object, which has been written with `write_measure` from the file `FileName`. The default HALCON file extension for a measure object is 'msr'. The values contained in the read measure object are stored in a measure object with the handle `MeasureHandle`.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .msr

▷ **MeasureHandle** (output_control) . . . . . measure ⇝ *handle*
Measure object handle.

Result

If the parameters are valid, the operator read_measure returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

measure_pos, measure_pairs

See also

write_measure

Module

1D Metrology

reset_fuzzy_measure (: : MeasureHandle, SetType :)

Reset a fuzzy function.

reset_fuzzy_measure discards a fuzzy function of the fuzzy set SetType. This function should have been set by set_fuzzy_measure before.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Parameters

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **SetType** (input_control) . . . . . string ⇝ *string*
Selection of the fuzzy set.
Default: 'contrast'
List of values: SetType ∈ {'position', 'position_pair', 'size', 'gray', 'contrast'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MeasureHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_fuzzy_measure

Possible Successors

fuzzy_measure_pos, fuzzy_measure_pairs

See also

set_fuzzy_measure, set_fuzzy_measure_norm_pair

Module

1D Metrology

serialize_measure (: : MeasureHandle : SerializedItemHandle)

Serialize a measure object.

serialize_measure serializes the data of a measure object (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_measure is converted to a serialized item. The measure object is defined by the handle MeasureHandle. The serialized measure object is returned by the handle SerializedItemHandle and can be deserialized by deserialize_measure.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Parameters

▷ **MeasureHandle** (input_control) ..... measure ⇝ *handle*
Measure object handle.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_measure returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MeasureHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

gen_measure_rectangle2, gen_measure_arc

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_measure

See also

read_measure, write_measure

Module

1D Metrology

set_fuzzy_measure (: : MeasureHandle, SetType, Function :)

Specify a fuzzy function.

set_fuzzy_measure specifies a fuzzy function passed in Function. The specified fuzzy functions enable fuzzy_measure_pos and fuzzy_measure_pairs / fuzzy_measure_pairing to evaluate and select the detected edge candidates. For this purpose, weighting characteristics for different edge features can be defined by one function each. Such a specified feature is called fuzzy set. Specifying no function for a fuzzy set means not to use this feature for the final edge evaluation. Setting a second fuzzy function to a set means to discard the first defined function and replace it by the second one. A previously defined fuzzy function can be discarded completely by reset_fuzzy_measure.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Functions for five different fuzzy set types selected by the SetType parameter can be defined, the sub types of a set being mutual exclusive:

- *'contrast'* will use the fuzzy function to evaluate the amplitudes of the edge candidates. When extracting edge pairs, the fuzzy evaluation is obtained by the geometric average of the fuzzy contrast scores of both edges.
- The fuzzy function of *'position'* evaluates the distance of each edge candidate to the reference point of the measure object, generated by gen_measure_arc or gen_measure_rectangle2. The reference point is located at the beginning whereas *'position_center'* or *'position_end'* sets the reference point to the middle or the end of the one-dimensional gray value profile instead. If the fuzzy position evaluation depends on the position of the object along the profile, *'position_first_edge'* / *'position_last_edge'* sets the reference point at the position of the first/last extracted edge. When extracting edge pairs the position of a pair is referenced by the geometric average of the fuzzy position scores of both edges.
- Similar to *'position'*, *'position_pair'* evaluates the distance of each edge pair to the reference point of the measure object. The position of a pair is defined by the center point between both edges. The object's reference can be set by *'position_pair_center'*, *'position_pair_end'* and *'position_first_pair'*, *'position_last_pair'*, respectively. Contrary to *'position'*, this set is only used by fuzzy_measure_pairs/fuzzy_measure_pairing.
- *'size'* denotes a fuzzy set that evaluates the normed distance of the two edges of a pair in pixels. This set is only used by fuzzy_measure_pairs/fuzzy_measure_pairing. Specifying an upper bound for the size by terminating the function with a corresponding fuzzy value of 0.0 will speed up fuzzy_measure_pairs / fuzzy_measure_pairing because not all possible pairs need to be considered.
- *'gray'* sets a fuzzy function to weight the mean projected gray value between two edges of a pair. This set is only used by fuzzy_measure_pairs / fuzzy_measure_pairing.

A fuzzy function is defined as a piecewise linear function by at least two pairs of values, sorted in an ascending order by their x value. The x values represent the edge feature and must lie within the parameter space of the set type, i.e., in case of *'contrast'* and *'gray'* feature and, e.g., byte images within the range $0.0 \leq x \leq 255.0$. In case of *'size'* x has to satisfy $0.0 \leq x$ whereas in case of *'position'* x can be any real number. The y values of the fuzzy function represent the weight of the corresponding feature value and have to satisfy the range of $0.0 \leq y \leq 1.0$. Outside of the function's interval, defined by the smallest and the greatest x value, the y values of the interval borders are continued constantly. Such Fuzzy functions can be generated by create_funct_1d_pairs.

If more than one set is defined, fuzzy_measure_pos / fuzzy_measure_pairs / fuzzy_measure_pairing yield the overall fuzzy weighting by the geometric middle of the weights of each set.

Parameters

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **SetType** (input_control) . . . . . string ⇝ *string*
Selection of the fuzzy set.
Default: 'contrast'
List of values: SetType ∈ {'position', 'position_center', 'position_end', 'position_first_edge', 'position_last_edge', 'position_pair_center', 'position_pair_end', 'position_first_pair', 'position_last_pair', 'size', 'gray', 'contrast'}

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Fuzzy function.

Example

```
* how to use a fuzzy function
* ...
gen_measure_rectangle2 (50, 100, 0, 200, 100, 512, 512, 'nearest_neighbor', \
                        MeasureHandle)
* create a generalized fuzzy function to evaluate edge pairs
* * (30% uncertainty).
create_funct_1d_pairs ([0.7,1.0,1.3], [0.0,1.0,0.0], SizeFunction)
* and transform it to expected size of 13.45 pixels
transform_funct_1d (SizeFunction, [1.0,0.0,13.45,0.0], TransformedFunction)
set_fuzzy_measure (MeasureHandle, 'size', TransformedFunction)

fuzzy_measure_pairs (Image, MeasureHandle, 1, 30, 0.5, 'all', RowEdgeFirst, \
                     ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond, \
                     ColumnEdgeSecond, AmplitudeSecond, RowEdgeCenter, \
                     ColumnEdgeCenter, FuzzyScore, IntraDistance, \
                     InterDistance)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MeasureHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

gen_measure_arc, gen_measure_rectangle2, create_funct_1d_pairs, transform_funct_1d

Possible Successors

fuzzy_measure_pos, fuzzy_measure_pairs

Alternatives

set_fuzzy_measure_norm_pair

See also

reset_fuzzy_measure

Module

1D Metrology

set_fuzzy_measure_norm_pair (: : MeasureHandle, PairSize, SetType, Function :)

Specify a normalized fuzzy function for edge pairs.

set_fuzzy_measure_norm_pair specifies a normalized fuzzy function passed in Function. The specified fuzzy functions enables fuzzy_measure_pos, fuzzy_measure_pairs and fuzzy_measure_pairing to evaluate and select the detected candidates of edges and edge pairs. For this purpose, weighting characteristics for different edge features can be defined by one function each. Such a specified feature is called fuzzy set. Specifying no function for a fuzzy set means not to use this feature for the final edge evaluation. Setting a second fuzzy function to a fuzzy set means to discard the first defined function and replace it by the second one. In difference to set_fuzzy_measure, the abscissa x of these functions must be defined relative to the desired size s of the edge pairs (passed in PairSize). This enables a generalized

usage of the defined functions. A previously defined normalized fuzzy function can be discarded completely by reset_fuzzy_measure.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Functions for three different fuzzy set types selected by the SetType parameter can be defined, the sub types of a set being mutual exclusive:

- *'size'* denotes a fuzzy set that valuates the normalized distance of two edges of a pair in pixels:

$$x = \frac{d}{s}(x \geq 0) \ .$$

  .

  Specifying an upper bound x_{max} for the size by terminating the function with a corresponding fuzzy value of 0.0 will speed up fuzzy_measure_pairs / fuzzy_measure_pairing because not all possible pairs must be considered. Additionally, this fuzzy set can also be specified as a normalized size difference by *'size_diff'*

$$x = \frac{s - d}{s}(x \leq 1)$$

  and a absolute normalized size difference by *'size_abs_diff'*

$$x = \frac{|s - d|}{s}(0 \leq x \leq 1) \ .$$

  .

- The fuzzy function of *'position'* evaluates the signed distance p of each edge candidate to the reference point of the measure object, generated by gen_measure_arc or gen_measure_rectangle2:

$$x = \frac{p}{s} \ .$$

  .

  The reference point is located at the beginning whereas *'position_center'* or *'position_end'* sets the reference point to the middle or the end of the one-dimensional gray value profile, instead. If the fuzzy position valuation depends on the position of the object along the profile *'position_first_edge'* / *'position_last_edge'* sets the reference point at the position of the first/last extracted edge. When extracting edge pairs, the position of a pair is referenced by the geometric average of the fuzzy position scores of both edges.
- Similar to *'position'*, *'position_pair'* evaluates the signed distance of each edge pair to the reference point of the measure object. The position of a pair is defined by the center point between both edges. The object's reference can be set by *'position_pair_center'*, *'position_pair_end'* and *'position_first_pair'*, *'position_last_pair'*, respectively. Contrary to *'position'*, this set is only used by fuzzy_measure_pairs/fuzzy_measure_pairing.

A normalized fuzzy function is defined as a piecewise linear function by at least two pairs of values, sorted in an ascending order by their x value. The y values of the fuzzy function represent the weight of the corresponding feature value and must satisfy the range of $0.0 \leq y \leq 1.0$. Outside of the function's interval, defined by the smallest and the greatest x value, the y values of the interval borders are continued constantly. Such Fuzzy functions can be generated by create_funct_1d_pairs.

If more than one set is defined, fuzzy_measure_pos / fuzzy_measure_pairs / fuzzy_measure_pairing yield the overall fuzzy weighting by the geometric mean of the weights of each set.

Parameters

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **PairSize** (input_control) . . . . . number ⇝ *real* / integer
Favored width of edge pairs.
Default: 10.0
Suggested values: PairSize ∈ {4.0, 6.0, 8.0, 10.0, 15.0, 20.0, 30.0}
Value range: 0.0 ≤ PairSize
Minimum increment: 0.1
Recommended increment: 1.0

▷ **SetType** (input_control) . . . . . string ⇝ *string*
Selection of the fuzzy set.
Default: 'size_abs_diff'
List of values: SetType ∈ {'size', 'size_diff', 'size_abs_diff', 'position', 'position_center', 'position_end', 'position_first_edge', 'position_last_edge', 'position_pair_center', 'position_pair_end', 'position_first_pair', 'position_last_pair'}

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Fuzzy function.

Example

```
* how to use a fuzzy function
* ...
gen_measure_rectangle2 (50, 100, 0, 200, 100, 512, 512, 'nearest_neighbor', \
                        MeasureHandle)
* create a generalized fuzzy function to evaluate edge pairs
* * (30% uncertainty).
create_funct_1d_pairs ([0.7,1.0,1.3], [0.0,1.0,0.0], SizeFunction)
* and set it for an expected pair size of 13.45 pixels
set_fuzzy_measure_norm_pair (MeasureHandle, 13.45, 'size', SizeFunction)

fuzzy_measure_pairs (Image, MeasureHandle, 1, 30, 0.5, 'all', RowEdgeFirst, \
                     ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond, \
                     ColumnEdgeSecond, AmplitudeSecond, RowEdgeCenter, \
                     ColumnEdgeCenter, FuzzyScore, IntraDistance, \
                     InterDistance)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MeasureHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

gen_measure_arc, gen_measure_rectangle2, create_funct_1d_pairs

Possible Successors

fuzzy_measure_pairs, fuzzy_measure_pairing

Alternatives

transform_funct_1d, set_fuzzy_measure

See also

reset_fuzzy_measure

Module

1D Metrology

translate_measure (: : MeasureHandle, Row, Column :)

Translate a measure object.

translate_measure translates the reference point of the measure object given by MeasureHandle to the point (Row, Column). If the measure object and the translated measure object lie completely within the image,

the measure object is shifted to the new reference point in an efficient manner. Otherwise, the measure object is generated anew with gen_measure_rectangle2 or gen_measure_arc using the parameters that were specified when the measure object was created and the new reference point.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Parameters

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **Row** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the new reference point.
Default: 50.0
Suggested values: Row ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Value range: 0.0 ≤ Row ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Column** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the new reference point.
Default: 100.0
Suggested values: Column ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Value range: 0.0 ≤ Column ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

Result

If the parameter values are correct the operator translate_measure returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MeasureHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

gen_measure_rectangle2, gen_measure_arc

Possible Successors

measure_pos, measure_pairs, fuzzy_measure_pos, fuzzy_measure_pairs, fuzzy_measure_pairing, measure_thresh

Alternatives

gen_measure_rectangle2, gen_measure_arc

See also

close_measure

Module

1D Metrology

write_measure (: : MeasureHandle, FileName :)

Write a measure object to a file.

write_measure writes a measure object that has been created by, e.g., gen_measure_rectangle2 to the file FileName. The measure object is defined by the handle MeasureHandle. The measure object can be read with read_measure. The default HALCON file extension for a measure object is 'msr'.

For an explanation of the concept of 1D measuring see the introduction of chapter 1D Measuring.

Parameters

▷ **MeasureHandle** (input_control) . . . . . measure ⇝ *handle*
Measure object handle.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: `.msr`

Result

If the parameters are valid, the operator `write_measure` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_measure_rectangle2`, `gen_measure_arc`

See also

`read_measure`

Module

1D Metrology

Chapter 2

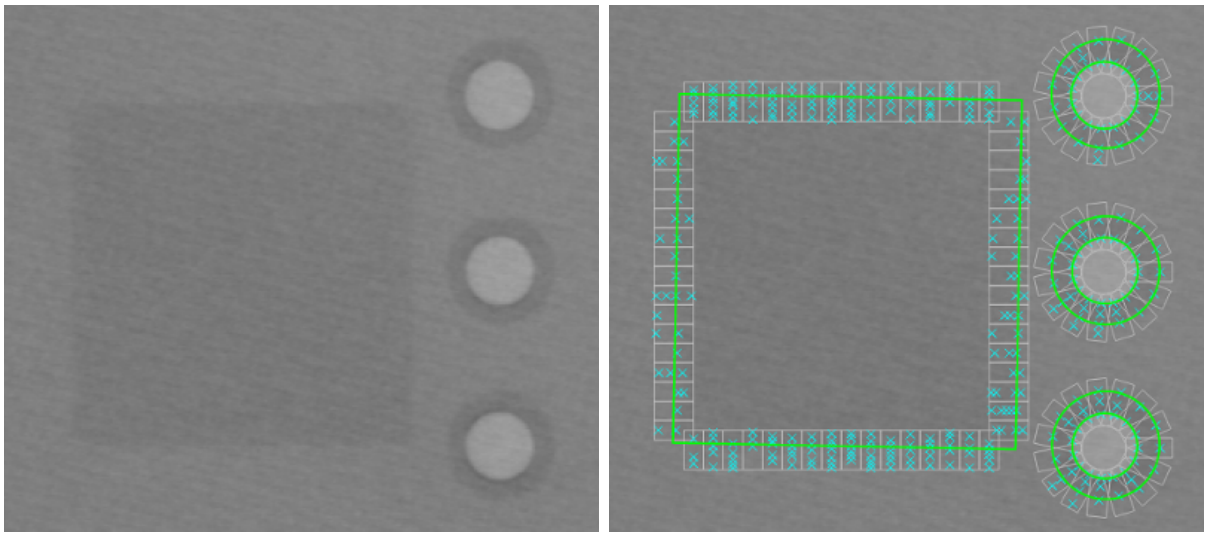
2D Metrology

This chapter contains operators for 2D metrology.

Concept of 2D Metrology

With 2D metrology, you can measure the dimensions of objects that can be represented by specific geometric primitives. The geometric shapes that can be measured comprise circles, ellipses, rectangles, and lines. You need approximate values for the positions, orientations, and dimensions of the objects to measure. Then, the real edge positions of the objects in the image are located near the boundaries of the approximate objects. With these edge positions, the parameters of the geometric shapes are optimized to better fit to the image data and are returned as measurement result.

The approximate values for the shape parameters of an object as well as some parameters that control the measurement are stored in a data structure that is called metrology object. The edges of the object in the image are located within so-called measure regions. These are rectangular regions that are arranged perpendicular to the boundaries of the metrology objects. Parameters that adjust the dimension and distribution of the measure regions are specified together with the approximate shape parameters for each metrology object. When the measurement is applied, the edge positions inside all measure regions are determined and fitted to geometric shapes using a RANSAC algorithm. All metrology objects, all further information that is necessary for the measurement, and the measurement results are stored in a data structure that is called metrology model.



(1) (2)

The geometric shapes in (1) are measured using 2D Metrology (2): A metrology model with 4 metrology objects (blue contours) is created. Using the edge positions (cyan crosses) located within the measure regions (gray rectangles) for each metrology object, the geometric shapes (green contours) are fitted and their parameters can be queried. As shown for the circles, more than one instance per object can be found. This image is from the example program `apply_metrology_model.hdev`.

In the following, the steps that are required to use 2D metrology are described briefly.

Create the metrology model and specify the image size: First, a metrology model must be created using

- create_metrology_model.

The metrology model is used as a container for one or more metrology objects. For an efficient measurement, after creating the metrology model, the image size of the image in which the measurements will be performed should be specified using

- set_metrology_model_image_size.

Provide approximate values: Then, metrology objects are added to the metrology model. Each metrology object consists of the approximate shape parameters for the corresponding object in the image and of the parameters that control the measurement. The parameters that control the measurement comprise, e.g., parameters that specify the dimension and distribution of the measure regions. Furthermore, several generic parameters can be adjusted for each metrology object. The metrology objects are specified with

- add_metrology_object_circle_measure for circles,
- add_metrology_object_ellipse_measure for ellipses,
- add_metrology_object_rectangle2_measure for rectangles, and
- add_metrology_object_line_measure for lines.
- add_metrology_object_generic allows to create metrology objects of different shapes (e.g., ellipse, circle, etc.) using one operator.

To visually inspect the defined metrology objects, you can access their XLD contours with the operator get_metrology_object_model_contour. To visually inspect the created measure regions, you can access their XLD contours with the operator get_metrology_object_measures.

Modify the model parameters: If a camera calibration has been performed, the camera parameters and the pose of the measurement plane can be set with

- set_metrology_model_param.

Then, the result of the measurements returned by get_metrology_object_result will be in world coordinates. The reference coordinate system in which the metrology objects are defined can also be changed with set_metrology_model_param.

Modify object parameters: Many parameters can be set when adding the metrology objects to the metrology model. Some of them can also be modified afterwards using the operator

- set_metrology_object_param.

Align the metrology model: To translate and rotate the metrology model before the next measurement is performed, you can use the operator

- align_metrology_model.

An alignment is temporary and is replaced by the next alignment. The metrology model itself is not changed. Note that typically the alignment parameters are obtained using shape-based matching.

Apply the measurement: The actual measurement in the image is performed with

- apply_metrology_model.

The operator locates the edges within the measure regions and fits the specified geometric shape to the edge positions using a RANSAC algorithm. The edges are located internally using the operator measure_pos or fuzzy_measure_pos (see also chapter 1D Measuring). The latter uses fuzzy methods and is used only if at least one fuzzy function was set via set_metrology_object_fuzzy_param before applying the measurement. If more than one instance of the returned object shape is needed (compare image above), the generic parameter *'num_instances'* must be set to the number of instances that should be returned. The parameter can be set when adding the individual metrology objects or afterwards with the operator set_metrology_object_param.

Access the results: After the measurement, the results can be accessed. The parameters of the adapted geometric shapes of the objects are queried with the operator

- get_metrology_object_result.

Querying only the edges used for the returned result and their amplitudes is also done using get_metrology_object_result.

The row and column coordinates of all located edges can be accessed with

- get_metrology_object_measures.

To visualize the adapted geometric shapes, you can access their XLD contours with

- get_metrology_object_result_contour.

Further operators

In addition to the operators mentioned above, you can copy the metrology handle with copy_metrology_model, write the metrology model to file with write_metrology_model, read a model from file again using read_metrology_model, and serialize or deserialize a metrology model using serialize_metrology_model or deserialize_metrology_model.

Furthermore, you can query various information from the metrology model. For example, you can query the indices of the metrology objects with get_metrology_object_indices, query parameters that are valid for the entire metrology model with get_metrology_model_param, query a fuzzy parameter of a metrology model with get_metrology_object_fuzzy_param, query the number of instances of the metrology objects of a metrology model with get_metrology_object_num_instances, and query the current configuration of the metrology model with get_metrology_object_param.

Additionally, you can reset all parameters of a metrology model using reset_metrology_object_param or reset only all fuzzy parameters and fuzzy functions of a metrology model using reset_metrology_object_fuzzy_param.

Glossary

In the following, the most important terms that are used in the context of 2D Metrology are described.

metrology model Data structure that contains all metrology objects, all information needed for the measurement, and the measurement results.

metrology object Data structure for the object to be measured with 2D metrology. The metrology object is represented by a specific geometric shape for which the shape parameters are approximately known. Additionally, it contains parameters that control the measurement, e.g., parameters that specify the dimension and distribution of the measure regions.

measure regions Rectangular regions that are arranged perpendicular to the boundaries of the approximate objects. Within these regions the edges that are used to get the exact shape parameters of the metrology objects are extracted.

returned instance of a metrology object For each metrology object, different instances of the object can be returned by the measurement, e.g., if parallel structures of the same shape exist near to the boundaries of the approximated geometric shape (see image above). The sequence of the returned instances is arbitrary, i.e., it is no measure for the quality of the fitting.

Further Information

See also the "Solution Guide on 2D Measuring" for further details about 2D metrology.

```
add_metrology_object_circle_measure ( : : MetrologyHandle, Row,
    Column, Radius, MeasureLength1, MeasureLength2, MeasureSigma,
    MeasureThreshold, GenParamName, GenParamValue : Index )
```

Add a circle or a circular arc to a metrology model.

add_metrology_object_circle_measure adds a metrology object of type circle or circular arc to a metrology model and prepares the rectangular measure regions. The handle of the model is passed in MetrologyHandle.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The geometric shape of the metrology object of type circle is specified by its center (Row, Column) and Radius. The rectangular measure regions lie *perpendicular* to the boundary of the circle. The half edge lengths of the measure regions are set in MeasureLength1 and MeasureLength2. The centers of the measure regions lie on the boundary of the circle. The parameter MeasureSigma specifies the standard deviation that is used by operator apply_metrology_model to smooth the gray values of the image. Salient edges can be selected with the parameter MeasureThreshold, which constitutes a threshold on the amplitude, i.e., the absolute value of the first derivative of the edge. The operator add_metrology_object_circle_measure returns the index of the added metrology object in parameter Index.

Furthermore, you can adjust some generic parameters with GenParamName and GenParamValue. The following values for GenParamName and GenParamValue are available:

'start_phi': The parameter specifies the angle at the start point of a circular arc. To create a closed circle the value of the parameter *'start_phi'* is set to 0 and the value of the parameter *'end_phi'* is set to 2π (with positive point order). The input value is mapped automatically to the interval $[0, 2\pi]$.

Suggested values: *0.0*, *0.78*, *6.28318*

Default: *0.0*

'end_phi': The parameter specifies the angle at the end point of a circular arc. To create a closed circle the value of the parameter *'start_phi'* is set to 0 and the value of the parameter *'end_phi'* is set to 2π (with positive point order). The input value is mapped internally automatically to the interval $[0, 2\pi]$.

Suggested values: *0.0*, *0.78*, *6.28318*

Default: *6.28318*

'point_order': The parameter specifies the direction of the circular arc. For the value *'positive'*, the circular arc is defined between *'start_phi'* and *'end_phi'* in mathematically positive direction (counterclockwise). For the value *'negative'*, the circular arc is defined between *'start_phi'* and *'end_phi'* in mathematically negative direction (clockwise).

List of values: *'positive'*, *'negative'*

Default: *'positive'*

Additionally all generic parameters, that are available for the operator set_metrology_object_param can be set. But note that for a lot of applications the default values are sufficient and no adjustment is necessary.

Parameters

▷ **MetrologyHandle** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Row** (input_control) ..... circle.center.y(-array) ⇝ *real* / integer
Row coordinate (or Y) of the center of the circle or circular arc.

▷ **Column** (input_control) ..... circle.center.x(-array) ⇝ *real* / integer
Column (or X) coordinate of the center of the circle or circular arc.

▷ **Radius** (input_control) ..... circle.radius(-array) ⇝ *real* / integer
Radius of the circle or circular arc.

▷ **MeasureLength1** (input_control) ..... number ⇝ *real* / integer
Half length of the measure regions perpendicular to the boundary.
Default: 20.0
Suggested values: MeasureLength1 ∈ {10.0, 20.0, 30.0}
Value range: 1.0 ≤ MeasureLength1
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: MeasureLength1 < Radius

▷ **MeasureLength2** (input_control) . . . . . number ⇝ *real* / integer
Half length of the measure regions tangential to the boundary.
Default: 5.0
Suggested values: MeasureLength2 ∈ {3.0, 5.0, 10.0}
Value range: 1.0 ≤ MeasureLength2
Minimum increment: 1.0
Recommended increment: 10.0

▷ **MeasureSigma** (input_control) . . . . . number ⇝ *real* / integer
Sigma of the Gaussian function for the smoothing.
Default: 1.0
Suggested values: MeasureSigma ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ MeasureSigma ≤ 100.0
Minimum increment: 0.01
Recommended increment: 0.1

▷ **MeasureThreshold** (input_control) . . . . . number ⇝ *real* / integer
Minimum edge amplitude.
Default: 30.0
Suggested values: MeasureThreshold ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ MeasureThreshold ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'distance_threshold', 'end_phi', 'instances_outside_measure_regions', 'max_num_iterations', 'measure_distance', 'measure_interpolation', 'measure_select', 'measure_transition', 'min_score', 'num_instances', 'num_measures', 'point_order', 'rand_seed', 'start_phi'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {1, 2, 3, 4, 5, 10, 20, 'all', 'true', 'false', 'first', 'last', 'positive', 'negative', 'uniform', 'nearest_neighbor', 'bilinear', 'bicubic'}

▷ **Index** (output_control) . . . . . integer ⇝ *integer*
Index of the created metrology object.

Example

```
read_image (Image, 'rings_and_nuts')
create_metrology_model (MetrologyHandle)
get_image_size (Image, Width, Height)
set_metrology_model_image_size (MetrologyHandle, Width, Height)
add_metrology_object_circle_measure (MetrologyHandle, 120, 130, 35, 10, 2, \
                                     1, 30, ['measure_distance'], [40], Index)
apply_metrology_model (Image, MetrologyHandle)
get_metrology_object_result (MetrologyHandle, Index, 'all', 'result_type', \
                             'all_param', Circle)
get_metrology_object_result_contour (Contour, MetrologyHandle, Index, \
                                     'all', 1.5)
```

Result

If the parameters are valid, the operator add_metrology_object_circle_measure returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_metrology_model_image_size

Possible Successors

align_metrology_model, apply_metrology_model

Alternatives

add_metrology_object_generic

See also

get_metrology_object_model_contour, set_metrology_model_param, add_metrology_object_ellipse_measure, add_metrology_object_line_measure, add_metrology_object_rectangle2_measure

Module

2D Metrology

add_metrology_object_ellipse_measure (: : MetrologyHandle, Row, Column, Phi, Radius1, Radius2, MeasureLength1, MeasureLength2, MeasureSigma, MeasureThreshold, GenParamName, GenParamValue : Index)

Add an ellipse or an elliptic arc to a metrology model.

add_metrology_object_ellipse_measure adds a metrology object of type ellipse or elliptic arc to a metrology model and prepares the rectangular measure regions. The handle of the model is passed in MetrologyHandle.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The geometric shape of the metrology object of type ellipse is specified by its center (Row, Column), the orientation of the main axis Phi, the length of the larger half axis Radius1, and the length of the smaller half axis Radius2. The input value for Phi is mapped automatically to the interval $]-\pi,\pi]$. The rectangular measure regions lie *perpendicular* to the boundary of the ellipse. The half edge lengths of the measure regions perpendicular and tangential to the boundary of the ellipse are set in MeasureLength1 and MeasureLength2. The centers of the measure regions lie on the boundary of the geometric shape. The parameter MeasureSigma specifies the standard deviation that is used by the operator apply_metrology_model to smooth the gray values of the image. Salient edges can be selected with the parameter MeasureThreshold, which constitutes a threshold on the amplitude, i.e., the absolute value of the first derivative of the edge. The operator add_metrology_object_ellipse_measure returns the index of the added metrology object in the parameter Index.

Furthermore, you can adjust some generic parameters within GenParamName and GenParamValue. The following values for GenParamName and GenParamValue are available:

'start_phi': The parameter specifies the angle at the start point of an elliptic arc. The angle at the start point is measured relative to the positive main axis specified with Phi and corresponds to the smallest surrounding circle of the ellipse. The actual start point of the ellipse is the intersection of the ellipse with the orthogonal projection of the corresponding circle point onto the main axis. The angle refers to the coordinate system of the ellipse, i.e., it is specified relative to the main axis and in a mathematical positive direction. Thus, the two main poles correspond to the angles 0 and π, the two minor poles to the angle $\pi/2$ and $3\pi/2$. To create a closed ellipse the value of the parameter *'start_phi'* is set to 0 and the value of the parameter *'end_phi'* is set to 2π (with positive point order). The input value is mapped internally automatically to the interval $[0, 2\pi]$.

Suggested values: *0.0*, *0.78*, *6.28318*

Default: *0.0*

'end_phi': The parameter specifies the angle at the end point of an elliptic arc. The angle at the end point are measured relative to the positive main axis specified with Phi and corresponds to the smallest surrounding

circle of the ellipse. The actual end point of the ellipse is the intersection of the ellipse with the orthogonal projection of the corresponding circle point onto the main axis. The angle refers to the coordinate system of the ellipse, i.e., it is specified relative to the main axis and in a mathematical positive direction. Thus, the two main poles correspond to the angles 0 and π, the two minor poles to the angle $\pi/2$ and $3\pi/2$. To create a closed ellipse the value of the parameter *'start_phi'* is set to 0 and the value of the parameter *'end_phi'* is set to 2π (with positive point order). The input value is mapped automatically to the interval $[0, 2\pi]$.

Suggested values: *0.0*, *0.78*, *6.28318*

Default: *6.28318*

'point_order': The parameter specifies the direction of the elliptic arc. For the value *'positive'*, the elliptic arc is defined between *'start_phi'* and *'end_phi'* in mathematically positive direction (counterclockwise). For the value *'negative'*, the elliptic arc is defined between *'start_phi'* and *'end_phi'* in mathematically negative direction (clockwise).

List of values: *'positive'*, *'negative'*

Default: *'positive'*

Additionally, all generic parameters that are available for the operator `set_metrology_object_param` can be set. But note that for a lot of applications the default values are sufficient and no adjustment is necessary.

Parameters

▷ **`MetrologyHandle`** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **`Row`** (input_control) . . . . . ellipse.center.y(-array) ⇝ *real* / integer
Row (or Y) coordinate of the center of the ellipse.

▷ **`Column`** (input_control) . . . . . ellipse.center.x(-array) ⇝ *real* / integer
Column (or X) coordinate of the center of the ellipse.

▷ **`Phi`** (input_control) . . . . . ellipse.angle.rad(-array) ⇝ *real* / integer
Orientation of the main axis [rad].

▷ **`Radius1`** (input_control) . . . . . ellipse.radius1(-array) ⇝ *real* / integer
Length of the larger half axis.

▷ **`Radius2`** (input_control) . . . . . ellipse.radius2(-array) ⇝ *real* / integer
Length of the smaller half axis.

▷ **`MeasureLength1`** (input_control) . . . . . number ⇝ *real* / integer
Half length of the measure regions perpendicular to the boundary.
Default: 20.0
Suggested values: `MeasureLength1` ∈ {10.0, 20.0, 30.0}
Value range: 1.0 ≤ MeasureLength1
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: MeasureLength1 < Radius1 && MeasureLength1 < Radius2

▷ **`MeasureLength2`** (input_control) . . . . . number ⇝ *real* / integer
Half length of the measure regions tangential to the boundary.
Default: 5.0
Suggested values: `MeasureLength2` ∈ {3.0, 5.0, 10.0}
Value range: 1.0 ≤ MeasureLength2
Minimum increment: 1.0
Recommended increment: 10.0

▷ **`MeasureSigma`** (input_control) . . . . . number ⇝ *real* / integer
Sigma of the Gaussian function for the smoothing.
Default: 1.0
Suggested values: `MeasureSigma` ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ `MeasureSigma` ≤ 100.0
Minimum increment: 0.01
Recommended increment: 0.1

▷ **MeasureThreshold** (input_control) ..... number ⇝ *real* / integer
Minimum edge amplitude.
Default: 30.0
Suggested values: MeasureThreshold ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ MeasureThreshold ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **GenParamName** (input_control) ..... attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'distance_threshold', 'end_phi', 'instances_outside_measure_regions', 'max_num_iterations', 'measure_distance', 'measure_interpolation', 'measure_select', 'measure_transition', 'min_score', 'num_instances', 'num_measures', 'point_order', 'rand_seed', 'start_phi'}

▷ **GenParamValue** (input_control) ..... attribute.value-array ⇝ *real* / integer / string
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {1, 2, 3, 4, 5, 10, 20, 'all', 'true', 'false', 'first', 'last', 'positive', 'negative', 'uniform', 'nearest_neighbor', 'bilinear', 'bicubic'}

▷ **Index** (output_control) ..... integer ⇝ *integer*
Index of the created metrology object.

Result

If the parameters are valid, the operator add_metrology_object_ellipse_measure returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_metrology_model_image_size

Possible Successors

align_metrology_model, apply_metrology_model

Alternatives

add_metrology_object_generic

See also

get_metrology_object_model_contour, set_metrology_model_param, add_metrology_object_circle_measure, add_metrology_object_line_measure, add_metrology_object_rectangle2_measure

Module

2D Metrology

add_metrology_object_generic (: : MetrologyHandle, Shape, ShapeParam, MeasureLength1, MeasureLength2, MeasureSigma, MeasureThreshold, GenParamName, GenParamValue : Index)

Add a metrology object to a metrology model.

add_metrology_object_generic adds a metrology object of type Shape to a metrology model and prepares the rectangular measure regions.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The handle of the model is passed in MetrologyHandle.

Shape specifies which type of object is added to the metrology model. The operator add_metrology_object_generic returns the index of the added metrology object in the parameter Index. Note that add_metrology_object_generic provides the functionality of the operators add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_rectangle2_measure and add_metrology_object_line_measure in one operator.

Possible shapes

Depending on the object specified in Shape the following values are expected:

'circle': The geometric shape of the metrology object of type circle is specified by its center (Row, Column) and radius.
ShapeParam=*[Row, Column, Radius]*

'rectangle2': The geometric shape of the metrology object of type rectangle is specified by its center (Row, Column), the orientation of the main axis Phi, and the half edge lengths Length1 and Length2. The input value for Phi is mapped automatically to the interval $]-\pi, \pi]$.
ShapeParam=*[Row, Column, Phi, Length1, Length2]*

'ellipse': The geometric shape of the metrology object of type ellipse is specified by its center (Row, Column), the orientation of the main axis Phi, the length of the larger half axis Radius1, and the length of the smaller half axis Radius2. The input value for Phi is mapped automatically to the interval $]-\pi, \pi]$.
ShapeParam=*[Row, Column, Phi, Radius1, Radius2]*

'line': The geometric shape of the metrology object of type line is described by the coordinates of its start point (RowBegin, ColumnBegin) and the coordinates of its end point (RowEnd, ColumnEnd).
ShapeParam=*[RowBegin, ColumnBegin, RowEnd, ColumnEnd]*

Definition of measure regions

add_metrology_object_generic also prepares the rectangular measure regions. The rectangular measure regions lie *perpendicular* to the boundary of the object. The half edge lengths of the measure regions perpendicular and tangential to the boundary of the object are set in MeasureLength1 and MeasureLength2. The centers of the measure regions lie on the boundary of the object. The parameter MeasureSigma specifies a standard deviation that is used by the operator apply_metrology_model to smooth the gray values of the image. Salient edges can be selected with the parameter MeasureThreshold, which constitutes a threshold on the amplitude, i.e., the absolute value of the first derivative of the edge.

Generic parameters

Generic parameters and their values can be specified using GenParamName and GenParamValue. All generic parameters that are available in the operator set_metrology_object_param can also be set in add_metrology_object_generic. But note that for a lot of applications the default values are sufficient and no adjustment is necessary. Furthermore, the following values for GenParamName and GenParamValue are available only for Shape = *'circle'* and *'ellipse'*:

'start_phi': The parameter specifies the angle at the start point of a circular or elliptic arc. For an ellipse, the angle at the start point is measured relative to the positive main axis and corresponds to the smallest surrounding circle of the ellipse. The actual start point of the ellipse is the intersection of the ellipse with the orthogonal projection of the corresponding circle point onto the main axis. To create a closed circle or ellipse the value of the parameter *'start_phi'* is set to 0 and the value of the parameter *'end_phi'* is set to 2π (with positive point order). The input value is mapped automatically to the interval $[0, 2\pi]$.
Suggested values: *0.0*, *0.78*, *6.28318*
Default: *0.0*

'end_phi': The parameter specifies the angle at the end point of a circular or elliptic arc. For an ellipse, the angle at the end point is measured relative to the positive main axis and corresponds to the smallest surrounding circle of the ellipse. The actual end point of the ellipse is the intersection of the ellipse with the orthogonal projection of the corresponding circle point onto the main axis. To create a closed circle or ellipse the value of the parameter *'start_phi'* is set to 0 and the value of the parameter *'end_phi'* is set to 2π (with positive point order). The input value is mapped internally automatically to the interval $[0, 2\pi]$.
Suggested values: *0.0*, *0.78*, *6.28318*
Default: *6.28318*

'point_order': The parameter specifies the direction of the circular or elliptic arc. For the value *'positive'*, the arc is defined between *'start_phi'* and *'end_phi'* in mathematically positive direction (counterclockwise). For the value *'negative'*, the arc is defined between *'start_phi'* and *'end_phi'* in mathematically negative direction (clockwise).

List of values: *'positive'*, *'negative'*

Default: *'positive'*

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Shape** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Type of the metrology object to be added.
Default: 'circle'
List of values: Shape ∈ {'circle', 'ellipse', 'rectangle2', 'line'}

▷ **ShapeParam** (input_control) . . . . . attribute.value-array ⇝ *real* / integer
Parameters of the metrology object to be added.

▷ **MeasureLength1** (input_control) . . . . . number ⇝ *real* / integer
Half length of the measure regions perpendicular to the boundary.
Default: 20.0
Suggested values: MeasureLength1 ∈ {10.0, 20.0, 30.0}
Value range: 1.0 ≤ MeasureLength1 ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **MeasureLength2** (input_control) . . . . . number ⇝ *real* / integer
Half length of the measure regions tangential to the boundary.
Default: 5.0
Suggested values: MeasureLength2 ∈ {3.0, 5.0, 10.0}
Value range: 1.0 ≤ MeasureLength2 ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **MeasureSigma** (input_control) . . . . . number ⇝ *real* / integer
Sigma of the Gaussian function for the smoothing.
Default: 1.0
Suggested values: MeasureSigma ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ MeasureSigma ≤ 100 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **MeasureThreshold** (input_control) . . . . . number ⇝ *real* / integer
Minimum edge amplitude.
Default: 30.0
Suggested values: MeasureThreshold ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ MeasureThreshold ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'distance_threshold', 'end_phi', 'instances_outside_measure_regions', 'max_num_iterations', 'measure_distance', 'measure_interpolation', 'measure_select', 'measure_transition', 'min_score', 'num_instances', 'num_measures', 'point_order', 'rand_seed', 'start_phi'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {1, 2, 3, 4, 5, 10, 20, 'all', 'true', 'false', 'first', 'last', 'positive', 'negative', 'uniform', 'nearest_neighbor', 'bilinear', 'bicubic'}

▷ **Index** (output_control) . . . . . integer ⇝ *integer*
Index of the created metrology object.

Example

```
create_metrology_model (MetrologyHandle)
read_image (Image, 'fabrik')
get_image_size (Image, Width, Height)
set_metrology_model_image_size (MetrologyHandle, Width, Height)
LinePar := [45,360,415,360]
RectPar1 := [270,232,rad(0),30,25]
RectPar2 := [360,230,rad(0),30,25]
LinePar := [45,360,415,360]
RectPar3 := [245,320,rad(-90),70,35]
* Add two rectangles
add_metrology_object_generic (MetrologyHandle, 'rectangle2', \
                              [RectPar1,RectPar2], 20, 5, 1, 30, [], [], \
                              Indices)
* Add a rectangle and a line
add_metrology_object_generic (MetrologyHandle, ['rectangle2','line'], \
                              [RectPar3,LinePar], 20, 5, 1, 30, [], [], \
                              Index)
get_metrology_object_model_contour (Contour, MetrologyHandle, 'all', 1.5)
apply_metrology_model (Image, MetrologyHandle)
get_metrology_object_result_contour (Contour1, MetrologyHandle, 'all', \
                                     'all', 1.5)
```

Result

If the parameters are valid, the operator `add_metrology_object_generic` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`set_metrology_model_image_size`, `set_metrology_model_param`

Possible Successors

`align_metrology_model`, `apply_metrology_model`, `set_metrology_model_param`

See also

`get_metrology_object_model_contour`

Module

2D Metrology

add_metrology_object_line_measure (: : MetrologyHandle, RowBegin, ColumnBegin, RowEnd, ColumnEnd, MeasureLength1, MeasureLength2, MeasureSigma, MeasureThreshold, GenParamName, GenParamValue : Index)

Add a line to a metrology model.

add_metrology_object_line_measure adds a metrology object of type line to a metrology model and prepares the rectangular measure regions. The handle of the model is passed in MetrologyHandle.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The geometric shape of the metrology object of type line is described by the coordinates of the start point (RowBegin, ColumnBegin) and the coordinates of the end point (RowEnd, ColumnEnd). The rectangular measure regions lie *perpendicular* to the line. The half edge lengths of the measure regions perpendicular and tangential to the line are set in MeasureLength1 and MeasureLength2. The centers of the measure regions lie on the line. The parameter MeasureSigma specifies a standard deviation that is used by the operator apply_metrology_model to smooth the gray values of the image. Salient edges can be selected with the parameter MeasureThreshold, which constitutes a threshold on the amplitude, i.e., the absolute value of the first derivative of the edge.

Furthermore, you can adjust some generic parameters within GenParamName and GenParamValue. In particular, all generic parameters that are available in the operator set_metrology_object_param can be set. But note that for a lot of applications the default values are sufficient and no adjustment is necessary.

The operator add_metrology_object_line_measure returns the index of the added metrology object in the parameter Index.

Parameters

▷ **MetrologyHandle** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **RowBegin** (input_control) ..... line.begin.y(-array) ⇝ *real* / integer
Row (or Y) coordinate of the start of the line.

▷ **ColumnBegin** (input_control) ..... line.begin.x(-array) ⇝ *real* / integer
Column (or X) coordinate of the start of the line.

▷ **RowEnd** (input_control) ..... line.end.y(-array) ⇝ *real* / integer
Row (or Y) coordinate of the end of the line.

▷ **ColumnEnd** (input_control) ..... line.end.x(-array) ⇝ *real* / integer
Column (or X) coordinate of the end of the line.

▷ **MeasureLength1** (input_control) ..... number ⇝ *real* / integer
Half length of the measure regions perpendicular to the boundary.
Default: 20.0
Suggested values: MeasureLength1 ∈ {10.0, 20.0, 30.0}
Value range: 1.0 ≤ MeasureLength1
Minimum increment: 1.0
Recommended increment: 10.0

▷ **MeasureLength2** (input_control) ..... number ⇝ *real* / integer
Half length of the measure regions tangential to the boundary.
Default: 5.0
Suggested values: MeasureLength2 ∈ {3.0, 5.0, 10.0}
Value range: 1.0 ≤ MeasureLength2
Minimum increment: 1.0
Recommended increment: 10.0

▷ **MeasureSigma** (input_control) ..... number ⇝ *real* / integer
Sigma of the Gaussian function for the smoothing.
Default: 1.0
Suggested values: MeasureSigma ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ MeasureSigma ≤ 100.0
Minimum increment: 0.01
Recommended increment: 0.1

▷ **MeasureThreshold** (input_control) ..... number ⇝ *real* / integer
Minimum edge amplitude.
Default: 30.0
Suggested values: MeasureThreshold ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ MeasureThreshold ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'distance_threshold', 'instances_outside_measure_regions', 'max_num_iterations', 'measure_distance', 'measure_interpolation', 'measure_select', 'measure_transition', 'min_score', 'num_instances', 'num_measures', 'rand_seed'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {1, 2, 3, 4, 5, 10, 20, 'all', 'true', 'false', 'first', 'last', 'positive', 'negative', 'uniform', 'nearest_neighbor', 'bilinear', 'bicubic'}

▷ **Index** (output_control) . . . . . integer ⇝ *integer*
Index of the created metrology object.

Result

If the parameters are valid, the operator add_metrology_object_line_measure returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_metrology_model_image_size

Possible Successors

align_metrology_model, apply_metrology_model

Alternatives

add_metrology_object_generic

See also

get_metrology_object_model_contour, set_metrology_model_param, add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_rectangle2_measure

Module

2D Metrology

add_metrology_object_rectangle2_measure (: : MetrologyHandle, Row, Column, Phi, Length1, Length2, MeasureLength1, MeasureLength2, MeasureSigma, MeasureThreshold, GenParamName, GenParamValue : Index)

Add a rectangle to a metrology model.

add_metrology_object_rectangle2_measure adds a metrology object of type rectangle to a metrology model and prepares the rectangular measure regions. The handle of the model is passed in MetrologyHandle.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The geometric shape of the metrology object of type rectangle is specified by its center (Row, Column), the orientation of the main axis Phi, and the half edge lengths Length1 and Length2. The input value for Phi is mapped automatically to the interval $]-\pi,\pi]$. The rectangular measure regions lie *perpendicular* to the boundary

of the rectangle. The half edge lengths of the measure regions perpendicular and tangential to the boundary of the rectangle are set in MeasureLength1 and MeasureLength2. The centers of the measure regions lie on the boundary of the rectangle. The parameter MeasureSigma specifies a standard deviation that is used by the operator apply_metrology_model to smooth the gray values of the image. Salient edges can be selected with the parameter MeasureThreshold, which constitutes a threshold on the amplitude, i.e., the absolute value of the first derivative of the edge.

Furthermore, you can adjust some generic parameters within GenParamName and GenParamValue. In particular, all generic parameters that are available in the operator set_metrology_object_param can be set. But note that for a lot of applications the default values are sufficient and no adjustment is necessary.

The operator add_metrology_object_rectangle2_measure returns the index of the added metrology object within the metrology model in the parameter Index.

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Row** (input_control) . . . . . rectangle2.center.y(-array) ⇝ *real* / integer
Row (or Y) coordinate of the center of the rectangle.

▷ **Column** (input_control) . . . . . rectangle2.center.x(-array) ⇝ *real* / integer
Column (or X) coordinate of the center of the rectangle.

▷ **Phi** (input_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real* / integer
Orientation of the main axis [rad].

▷ **Length1** (input_control) . . . . . rectangle2.hwidth(-array) ⇝ *real* / integer
Length of the larger half edge of the rectangle.

▷ **Length2** (input_control) . . . . . rectangle2.hheight(-array) ⇝ *real* / integer
Length of the smaller half edge of the rectangle.

▷ **MeasureLength1** (input_control) . . . . . number ⇝ *real* / integer
Half length of the measure regions perpendicular to the boundary.
Default: 20.0
Suggested values: MeasureLength1 ∈ {10.0, 20.0, 30.0}
Value range: 1.0 ≤ MeasureLength1
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: MeasureLength1 < Length1 && MeasureLength1 < Length2

▷ **MeasureLength2** (input_control) . . . . . number ⇝ *real* / integer
Half length of the measure regions tangential to the boundary.
Default: 5.0
Suggested values: MeasureLength2 ∈ {3.0, 5.0, 10.0}
Value range: 1.0 ≤ MeasureLength2
Minimum increment: 1.0
Recommended increment: 10.0

▷ **MeasureSigma** (input_control) . . . . . number ⇝ *real* / integer
Sigma of the Gaussian function for the smoothing.
Default: 1.0
Suggested values: MeasureSigma ∈ {0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0}
Value range: 0.4 ≤ MeasureSigma ≤ 100.0
Minimum increment: 0.01
Recommended increment: 0.1

▷ **MeasureThreshold** (input_control) . . . . . number ⇝ *real* / integer
Minimum edge amplitude.
Default: 30.0
Suggested values: MeasureThreshold ∈ {5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0}
Value range: 1 ≤ MeasureThreshold ≤ 255 (lin)
Minimum increment: 0.5
Recommended increment: 2

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'distance_threshold', 'instances_outside_measure_regions',

'max_num_iterations', 'measure_distance', 'measure_interpolation', 'measure_select', 'measure_transition', 'min_score', 'num_instances', 'num_measures', 'rand_seed'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {1, 2, 3, 4, 5, 10, 20, 'all', 'true', 'false', 'first', 'last', 'positive', 'negative', 'uniform', 'nearest_neighbor', 'bilinear', 'bicubic'}

▷ **Index** (output_control) . . . . . integer ⇝ *integer*
Index of the created metrology object.

Result

If the parameters are valid, the operator add_metrology_object_rectangle2_measure returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_metrology_model_image_size

Possible Successors

align_metrology_model, apply_metrology_model

Alternatives

add_metrology_object_generic

See also

get_metrology_object_model_contour, set_metrology_model_param, add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_line_measure

Module

2D Metrology

align_metrology_model (: : MetrologyHandle, Row, Column, Angle :)

Alignment of a metrology model.

align_metrology_model moves and rotates the whole metrology model MetrologyHandle relative to the image coordinate system which has its origin in the top left corner.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

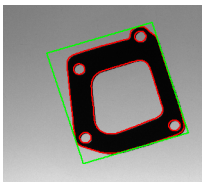
An alignment ensures, that the position and orientation of the metrology model is adapted to the objects to be measured in the current image. The alignment is then used by apply_metrology_model to perform the measurement. First the metrology model is rotated by Angle, then the metrology model is translated by Row and Column. The values of the alignment are overwritten by the next call of align_metrology_model.

Computation of the parameters of the alignment

The parameters of the alignment can be determined using diverse methods. Here, three possibilities to determine the parameters are listed:

Using region analysis:

If the metrology model can be extracted using region processing and if the pose of the Region changes only slightly in subsequent images, the parameters of the reference system of the metrology model and of the alignment can be derived using region analysis. In the following picture `threshold` and `smallest_rectangle2` were used to obtain these parameter.

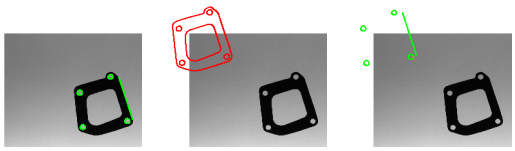


The region extracted with `threshold` is shown in red. The rectangle computed with `smallest_rectangle2` is shown in green.

1. Setting the reference system
   In the image in which the metrology model was defined, extract a region containing the metrology objects. The pose of this region with respect to the image coordinate system is determined and set as the reference system of the metrology model using `set_metrology_model_param`. This step is only performed once when setting up the metrology model.
   Example:
   ```
   threshold(Image, Region, 0, 50)
   smallest_rectangle2(Region, RowOrig, ColumnOrig, AngleOrig, Length1,
   Length2)
   set_metrology_model_param(MetrologyHandle, 'reference_system',
   [RowOrig, ColumnOrig, AngleOrig])
   ```
2. Determining the alignment
   In an image where the metrology model occurs in a different Pose, the current pose if the extracted region is determined. This pose is then used to align the metrology model.
   Example:
   ```
   threshold(CurrentImage, Region, 0, 50)
   smallest_rectangle2(Region, RowAlign, ColumnAlign, AngleAlign,
   Length1, Length2)
   align_metrology_model(MetrologyHandle, RowAlign, ColumnAlign,
   AngleAlign)
   ```

Using a shape model:

If a shape model is used to align the metrology model, the reference system with respect to which the metrology objects are given has to be set so that it coincides with the coordinate system used by the shape model. Only then, the results (*'row'*, *'column'*, *'angle'*) of `get_generic_shape_model_result` can be used directly in `align_metrology_model` to align the metrology model in the current image. The individual steps that are needed, are shown below.



(1) (2) (3)

(1) The contours of the metrology model (2) The contours of the shape model (3) The contours of the metrology model after setting the correct reference system.

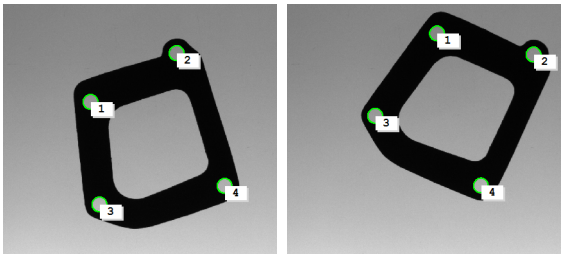
1. Setting the reference system
   In the image in which the metrology model was defined, the pose of the origin of the shape model is determined and set as the reference system of the metrology model using `set_metrology_model_param`. This step is only performed once when setting up the metrology model.
   Example:
   ```
   train_generic_shape_model(Image, ModelID)
   area_center(Image, Area, RowOrig, ColumnOrig)
   set_metrology_model_param(MetrologyHandle, 'reference_system',
   [RowOrig,ColumnOrig,0])
   ```

2. Determining the alignment
   In an image, in which the object to be measured occurs in a different pose, the current pose of the shape model is determined and set in the metrology model using `align_metrology_model`.
   Example:

```
find_generic_shape_model(CurrentImage, ModelID, MatchResultID,
NumMatchResult)
get_generic_shape_model_result(MatchResultID, 'all', 'row', RowAlign)
get_generic_shape_model_result(MatchResultID, 'all', 'column',
ColumnAlign)
get_generic_shape_model_result(MatchResultID, 'all', 'angle',
AngleAlign)
align_metrology_model(MetrologyHandle, RowAlign, ColumnAlign,
AngleAlign)
```

Using a rigid 2D transformation:

If certain model points (given as [PRowModel], [PColumnModel]) can be clearly identified and if they can still be clearly identified in further images in which the objects to be measured can occur shifted or rotated, a rigid transformation can be calculated between those points. The transformation parameters can then directly be used for aligning the model. In this case, the reference point of the metrology model does not have to be changed.



(1) (2)

(1) The contours of the metrology object and the four corresponding points in the image that was used for the creation of the metrology model. (2) The contours of the metrology object and the four corresponding points in a new image.

1. Determine the point correspondences
2. Estimate the model pose
   The following operator sequence calculates the parameters of the model pose (`Row`, `Column`, `Angle`) from corresponding points in the model image and one other image.
   Example:

```
vector_to_rigid(PRowModel, PColumnModel, PRowCurrent, PColumnCurrent,
HomMat2D)
hom_mat2d_to_affine_par(HomMat2D, Sx, Sy, Angle, Theta, Row, Column)
align_metrology_model(MetrologyHandle, Row, Column, Angle)
```

Parameters

▷ **`MetrologyHandle`** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **`Row`** (input_control) . . . . . real ⇝ *real* / integer
Row coordinate of the alignment.
Default: 0

▷ **`Column`** (input_control) . . . . . real ⇝ *real* / integer
Column coordinate of the alignment.
Default: 0

▷ **`Angle`** (input_control) . . . . . angle.rad ⇝ *real* / integer
Rotation angle of the alignment.
Default: 0

Example

```
read_image (Image, 'metal-parts/circle_plate_01')
```

```
create_metrology_model (MetrologyHandle)
get_image_size (Image, Width, Height)
set_metrology_model_image_size (MetrologyHandle, Width, Height)
CircleParam := [354,274,53]
CircleParam := [CircleParam,350,519,53]
add_metrology_object_generic (MetrologyHandle, 'circle', CircleParam, 20,\
                              5, 1, 30, [], [], CircleIndices)
create_generic_shape_model (ModelID)
set_generic_shape_model_param (ModelID, 'metric', 'use_polarity')
set_generic_shape_model_param (ModelID, 'min_contrast', 20)
train_generic_shape_model (Image, ModelID)
* Determine location of shape model origin
area_center (Image, Area, RowOrigin, ColOrigin)
set_metrology_model_param (MetrologyHandle, 'reference_system', \
                           [RowOrigin,ColOrigin,0])
read_image (CurrentImage, 'metal-parts/circle_plate_02')
find_generic_shape_model (CurrentImage, ModelID, MatchResultID, \
                          NumMatchResult)
get_generic_shape_model_result (MatchResultID, 'all', 'row', Row)
get_generic_shape_model_result (MatchResultID, 'all', 'column', Col)
get_generic_shape_model_result (MatchResultID, 'all', 'angle', Angle)
align_metrology_model (MetrologyHandle, Row, Col, Angle)
apply_metrology_model (CurrentImage, MetrologyHandle)
get_metrology_object_result (MetrologyHandle, CircleIndices, 'all', \
                             'result_type', 'all_param', Rectangle)
get_metrology_object_result_contour (Contour, MetrologyHandle, \
                                     CircleIndices, 'all', 1.5)
```

Result

If the parameters are valid, the operator `align_metrology_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`set_metrology_model_param`, `add_metrology_object_generic`

Possible Successors

`apply_metrology_model`

See also

`get_metrology_object_model_contour`

Module

2D Metrology

apply_metrology_model (Image : : MetrologyHandle :)

Measure and fit the geometric shapes of all metrology objects of a metrology model.

apply_metrology_model locates the edges inside the measure regions of the metrology objects of the metrology model MetrologyHandle within Image and fits the corresponding geometric shapes to the resulting edge positions.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The measurements are performed as follows:

Determining the edge positions

Within the measure regions of the metrology objects, the positions of the edges are determined. The edge location is calculated internally with the operator measure_pos or fuzzy_measure_pos. The latter is used if at least one fuzzy function was set for the metrology objects with set_metrology_object_fuzzy_param.

Fitting geometric shapes to the edge positions

The geometric shapes of the metrology objects are adapted to fit optimally to the resulting edge positions. In particular, a RANSAC algorithm is used to select a set of initial edge positions that is necessary to create an instance of the specific geometric shape, e.g., three edge positions are selected for a metrology object of type circle. Then, those edge positions that are near the corresponding instance of the geometric shape are determined and, if the number of suitable edge positions is sufficient (see the generic parameter *'min_score'* of set_metrology_object_param), are selected for the final fitting of the geometric shape. If the number of suitable edge positions is not sufficient, another set of initial edge positions is tested until a suitable selection of edge positions is found. Into the edge positions that are selected for the final fitting, the geometric shape is fitted and its parameters are stored in the metrology model. Note that more than one instance for each metrology object is returned if the generic parameter *'num_instances'* is set to value larger than 1. This and other parameters can be set when adding the metrology objects to the metrology model or separately with the operator set_metrology_object_param. Note that for each instance of the metrology object different initial edge positions are used, i.e., a second instance is based on edge positions that were not already used for the fitting of the first instance. The algorithm stops either when *'num_instances'* instances were found or if the remaining number of suitable initial edge positions is too low for a further fitting of the geometric shape.

Accessing the results

The results of the measurements can be accessed from the metrology model using get_metrology_object_result. Note that if more than one instance of an object is returned, the order of the returned instances is arbitrary and therefore no measure for the quality of the fitting. Note further that if the parameters *'camera_param'* and *'plane_pose'* were set for the metrology model using set_metrology_model_param, world coordinates are used for the fitting. Otherwise, image coordinates are used. The XLD contours for the measured objects can be obtained using get_metrology_object_result_contour.

Attention

Note that all measure regions of all metrology objects must be recomputed if the width or the height of the input Image is not equal to the width and height stored in the metrology object (e.g., set with set_metrology_model_image_size). This leads to longer execution times of the operator.

Note further that apply_metrology_model ignores the domain of Image for efficiency reasons (see also measure_pos).

Parameters

- ▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
  Handle of the metrology model.

Result

If the parameters are valid, the operator apply_metrology_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

add_metrology_object_generic, add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_line_measure, add_metrology_object_rectangle2_measure, align_metrology_model, set_metrology_model_param, set_metrology_object_param

Possible Successors

get_metrology_object_result, get_metrology_object_result_contour, get_metrology_object_measures

See also

set_metrology_object_fuzzy_param, read_metrology_model, write_metrology_model

Module

2D Metrology

clear_metrology_model (: : MetrologyHandle :)

Delete a metrology model and free the allocated memory.

clear_metrology_model deletes a metrology model that was created by create_metrology_model, copy_metrology_model, read_metrology_model, or deserialize_metrology_model. Note that by deleting the model also the including metrology objects are deleted. All memory used by the metrology model and the metrology objects is freed. The handle of the model is passed in MetrologyHandle. After the operator call, the metrology model is invalid.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

Parameters

▷ **MetrologyHandle** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

Result

The operator clear_metrology_model returns the value 2 (H_MSG_TRUE) if a valid handle was passed and the referred metrology model can be freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

get_metrology_object_result, write_metrology_model

Module

2D Metrology

clear_metrology_object (: : MetrologyHandle, Index :)

Delete metrology objects and free the allocated memory.

clear_metrology_object deletes in a metrology model metrology objects created, e.g, by add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_line_measure, or add_metrology_object_rectangle2_measure.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

All memory used by the metrology objects is freed. The handle of the metrology model is passed in MetrologyHandle. The index of the metrology objects is passed in Index. If Index is set to *'all'*, all metrology objects are deleted. After the operator call the metrology objects are invalid.

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: Index ∈ {'all', 0, 1, 2}

Result

If the parameters are valid, the operator clear_metrology_object returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

2D Metrology

copy_metrology_model (: : MetrologyHandle,
Index : CopiedMetrologyHandle)

Copy a metrology model.

copy_metrology_model creates a new metrology model and copies the selected metrology objects of the input metrology model to this new output metrology model.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The input metrology model is defined by a handle MetrologyHandle. The parameter Index determines the metrology objects that are copied. With Index set to *'all'*, all metrology objects are copied. The operator returns the handle CopiedMetrologyHandle of the new metrology model. It can be used to save memory space. Access to the parameters of the metrology objects is possible, e.g., with the operator get_metrology_object_param.

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: Index ∈ {'all', 0, 1, 2}

▷ **CopiedMetrologyHandle** (output_control) . . . . . integer ⇝ *integer*
Handle of the copied metrology model.

Result

If the parameters are valid, the operator `copy_metrology_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

2D Metrology

create_metrology_model (: : : MetrologyHandle)

Create the data structure that is needed to measure geometric shapes.

`create_metrology_model` creates a metrology model, i.e., the data structure that is needed to measure objects with a specific geometric shape (metrology object) via 2D metrology, and returns it in the handle `MetrologyHandle`.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

Attention

Note, that after calling the operator `create_metrology_model` the operator `set_metrology_model_image_size` should be called for efficiency reasons.

Parameters

▷ **MetrologyHandle** (output_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

Example

```
read_image (Image, 'fabrik')
create_metrology_model (MetrologyHandle)
get_image_size (Image, Width, Height)
set_metrology_model_image_size (MetrologyHandle, Width, Height)
add_metrology_object_rectangle2_measure (MetrologyHandle, 270, 230, 0, 30, \
                                         25, 10, 2, 1, 30, [], [], Index)
apply_metrology_model (Image, MetrologyHandle)
get_metrology_object_result (MetrologyHandle, Index, 'all', 'result_type', \
                             'all_param', Rectangle)
get_metrology_object_result_contour (Contour, MetrologyHandle, \
                                     Index, 'all', 1.5)
```

Result

If the parameters are valid, the operator `create_metrology_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`set_metrology_model_image_size`

Module

2D Metrology

deserialize_metrology_model (: : SerializedItemHandle : MetrologyHandle)

Deserialize a serialized metrology model.

deserialize_metrology_model deserializes a metrology model, that was serialized by serialize_metrology_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized metrology model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created metrology model with the handle MetrologyHandle. Access to the parameters of the metrology model is possible, e.g., with the operators get_metrology_object_param or get_metrology_object_fuzzy_param.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **MetrologyHandle** (output_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

Result

If the parameters are valid, the operator deserialize_metrology_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_metrology_model

Possible Successors

get_metrology_object_param, get_metrology_object_fuzzy_param, apply_metrology_model

Module

2D Metrology

get_metrology_model_param (: : MetrologyHandle, GenParamName : GenParamValue)

Get parameters that are valid for the entire metrology model.

get_metrology_model_param queries parameters that are valid for the entire metrology model.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle.

The following generic parameter names for GenParamName are possible:

'camera_param': The internal camera parameters that are set for the metrology model.

'plane_pose': The 3D pose of the measurement plane that is set for the metrology model. The 3D pose is given in camera coordinates.

'reference_system': The rotation and translation of the current reference coordinate system with respect to the image coordinate system. The tuple returned in GenParamValue contains [row, column, angle].

'scale': The scaling factor or unit of the results of the measurement returned by `get_metrology_object_result`.

Parameters

▷ **`MetrologyHandle`** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **`GenParamName`** (input_control) ..... attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'camera_param'
List of values: `GenParamName` ∈ {'camera_param', 'plane_pose', 'scale', 'reference_system'}

▷ **`GenParamValue`** (output_control) ..... attribute.value(-array) ⇝ *string* / real / integer
Value of the generic parameter.

Result

If the parameters are valid, the operator `get_metrology_model_param` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_metrology_object_indices`, `set_metrology_model_param`

Possible Successors

`get_metrology_object_param`

See also

`get_metrology_object_param`, `get_metrology_object_num_instances`

Module

2D Metrology

`get_metrology_object_fuzzy_param` (: : `MetrologyHandle`, `Index`, `GenParamName` : `GenParamValue`)

Get a fuzzy parameter of a metrology model.

`get_metrology_object_param` allows to access the fuzzy parameters of metrology objects.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle `MetrologyHandle`. The parameter `Index` specifies for which metrology objects the information is accessed. With `Index` set to *'all'*, the parameters of all metrology objects are accessed. The names of the desired parameters are passed in the generic parameter `GenParamName`, the corresponding values are returned in `GenParamValue` in the same order. All these fuzzy parameters can be set and changed at any time with the operator `set_metrology_object_fuzzy_param`.

The following parameters can be accessed:

'fuzzy_thresh': The meaning and the use of this parameter is equivalent to the parameter `FuzzyThresh` of the operator `fuzzy_measure_pos` and is described there.

'function_contrast': With this parameter the fuzzy function of type contrast that is set in the operator `set_metrology_object_param` can be queried. The meaning and the use of this parameter is equivalent to the parameter `SetType` with the value *'contrast'* of the operator `set_fuzzy_measure` and is described there. The return value `GenParamValue` contains the function of the metrology object.

'function_position': With this parameter the fuzzy function of type position that is set in the operator `set_metrology_object_param` can be queried. Because only one fuzzy function of a type can be set, only the last set function can be returned. The type can be *'function_position'*, *'function_position_center'*, *'function_position_end'*, *'function_position_first_edge'*, or *'function_position_last_edge'*.

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: Index ∈ {'all', 0, 1, 2}

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: 'fuzzy_thresh'
List of values: GenParamName ∈ {'function_contrast', 'function_position', 'fuzzy_thresh'}

▷ **GenParamValue** (output_control) . . . . . attribute.value-array ⇝ *real* / integer
Values of the generic parameters.

Result

If the parameters are valid, the operator get_metrology_object_fuzzy_param returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_metrology_object_indices, set_metrology_object_fuzzy_param

Possible Successors

set_metrology_object_fuzzy_param

See also

get_metrology_object_param

Module

2D Metrology

get_metrology_object_indices (: : MetrologyHandle : Indices)

Get the indices of the metrology objects of a metrology model.

get_metrology_object_indices allows to access the indices of the metrology objects.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle. The operator get_metrology_object_indices returns the indices of the metrology object in the parameter Indices. Access to the parameters of the metrology object is possible, e.g., with the operator get_metrology_object_param.

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Indices** (output_control) . . . . . integer(-array) ⇝ *integer*
Indices of the metrology objects.

Result

If the parameters are valid, the operator get_metrology_object_indices returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_metrology_model

Possible Successors

get_metrology_object_param, get_metrology_object_fuzzy_param

See also

get_metrology_object_num_instances

Module

2D Metrology

get_metrology_object_measures (: Contours : MetrologyHandle, Index, Transition : Row, Column)

Get the measure regions and the results of the edge location for the metrology objects of a metrology model.

get_metrology_object_measures allows to access the measure regions of the metrology objects that were created with add_metrology_object_generic, add_metrology_object_circle_measure, etc. as XLD contours and the results of the edge location in image coordinates that was performed by apply_metrology_model.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle. The parameter Index determines for which metrology objects the information is accessed. With Index set to *'all'*, the measure regions and the results of the edge location for all metrology objects are accessed.

If positive and negative edges are available in the measure regions (see the generic parameter value *'measure_transition'* of the operator set_metrology_object_param), with the parameter Transition the desired edges (positive or negative) can be selected. If Transition is set to *'positive'*, only positive edges are returned. If Transition is set to *'negative'*, only negative edges are returned. All edges are returned if the parameter Transition is set to *'all'*.

The operator get_metrology_object_measures returns for each measure region one rectangular XLD contour with the boundary of the measure region in the parameter Contours. After calling apply_metrology_model, additionally the image coordinates of the results of the edge location are returned as single points in the parameters Row and Column. Note that the order for the values of these points is not defined. Furthermore, there is no possibility to assign the results of the edge location to specific measure regions. If get_metrology_object_measures is called before apply_metrology_model, the parameters Row and Column remain empty.

Parameters

▷ **Contours** (output_object) ..... xld_cont-array ⇝ *object*
Rectangular XLD Contours of measure regions.

▷ **MetrologyHandle** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) ..... integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: Index ∈ {'all', 0, 1, 2}

▷ **Transition** (input_control) ..... string ⇝ *string*
Select light/dark or dark/light edges.
Default: 'all'
List of values: Transition ∈ {'all', 'negative', 'positive'}

▷ **Row** (output_control) ..... point.y-array ⇝ *real*
Row coordinates of the measured edges.

▷ **Column** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinates of the measured edges.

Result

If the parameters are valid, the operator `get_metrology_object_measures` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`apply_metrology_model`

See also

`add_metrology_object_generic`, `add_metrology_object_ellipse_measure`, `add_metrology_object_line_measure`, `add_metrology_object_rectangle2_measure`, `add_metrology_object_circle_measure`

Module

2D Metrology

get_metrology_object_model_contour (: Contour : MetrologyHandle, Index, Resolution :)

Query the model contour of a metrology object in image coordinates.

`get_metrology_object_model_contour` returns the contours for the chosen metrology objects in image coordinates.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle `MetrologyHandle`. The parameter `Index` specifies for which metrology objects the contours are queried. For `Index` set to *'all'*, the contours of all metrology objects are returned.

The form and pose of each contour is determined by the parameters set when adding the object using e.g., `add_metrology_object_generic`, `add_metrology_object_circle_measure`, etc. If a different reference coordinate system was set for the metrology model using `set_metrology_model_param` or an alignment of the metrology model was performed using `align_metrology_model`, these values influence the current pose of the metrology objects and thus the pose of the contours returned in `Contour`.

The resolution of the returned `Contour` is controlled via `Resolution` containing the Euclidean distance (in pixel) between neighboring contour points. If the input value falls below the minimal possible value (1.192e-7), the resolution is set internally to the smallest valid value.

Parameters

▷ **Contour** (output_object) . . . . . xld_cont(-array) ⇝ *object*
Model contour.

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Index of the metrology object.
Default: 0
Suggested values: `Index` ∈ {'all', 0, 1, 2}

▷ **Resolution** (input_control) . . . . . real ⇝ *real*
Distance between neighboring contour points.
Default: 1.5
Restriction: Resolution >= 1.192e-7

Result

If the parameters are valid, the operator `get_metrology_object_model_contour` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`add_metrology_object_generic`, `add_metrology_object_circle_measure`, `add_metrology_object_ellipse_measure`, `add_metrology_object_rectangle2_measure`, `add_metrology_object_line_measure`

Possible Successors

`apply_metrology_model`

See also

`set_metrology_model_param`, `get_metrology_object_measures`, `align_metrology_model`

Module

2D Metrology

`get_metrology_object_num_instances` (: : `MetrologyHandle`, `Index` : `NumInstances`)

Get the number of instances of the metrology objects of a metrology model.

`get_metrology_object_num_instances` allows to access the number of instances (results) of measurements applied by `apply_metrology_model` for the metrology objects. Note that by default, the maximum number of instances of each metrology object is set to *1*. Thus, by default, the result of `get_metrology_object_num_instances` will typically be *1* as well. To allow more instances, before applying the measurement with `apply_metrology_model` you have to explicitly set the parameter *'num_instances'* to a higher value or to *'all'* using `set_metrology_object_param`.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle `MetrologyHandle`. The parameter `Index` specifies for which metrology object the instances are queried. For `Index` set to *'all'*, the number of instances of all metrology objects are returned. The number of instances is returned in `NumInstances` for each metrology object that was passed in `Index`.

Parameters

▷ **`MetrologyHandle`** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **`Index`** (input_control) ..... integer(-array) ⇝ *integer* / string
Index of the metrology objects.
Default: 0
Suggested values: `Index` ∈ {'all', 0, 1, 2}

▷ **`NumInstances`** (output_control) ..... integer(-array) ⇝ *real* / integer
Number of Instances of the metrology objects.

Result

If the parameters are valid, the operator `get_metrology_object_num_instances` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

apply_metrology_model

Possible Successors

clear_metrology_model

See also

get_metrology_object_indices

Module

2D Metrology

get_metrology_object_param (: : MetrologyHandle, Index, GenParamName : GenParamValue)

Get one or several parameters of a metrology model.

get_metrology_object_param allows to access the parameters that are used by a metrology object.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle. The parameter Index determines for which metrology objects the information is accessed. With Index set to *'all'*, the parameters of all metrology objects are accessed. The names of the desired parameters are passed in the generic parameter GenParamName, the corresponding values are returned in GenParamValue in the same order. All these general parameters can be set and changed at any time with the operator set_metrology_object_param. Parameters that describe the geometry of an object can only be set by creating the metrology object with the operators add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_line_measure, or add_metrology_object_rectangle2_measure.

The following parameters can be accessed:

- Valid for all types of metrology objects:

  'min_score', *'num_instances'*, *'instances_outside_measure_regions'*: The meaning and the use of these parameters is described with the operator set_metrology_object_param.

  'rand_seed', *'distance_threshold'*, *'max_num_iterations'*: The meaning and the use of these parameters is described with the operator set_metrology_object_param.

  'measure_length1', *'measure_length2'*: The meaning and the use of these parameters is described with the operator set_metrology_object_param.

  'measure_sigma', *'measure_threshold'*, *'measure_transition'*, *'measure_select'*: The meaning and the use of these parameters is described with the operator measure_pos by the parameters Sigma, Threshold, Transition, and Select.

  'measure_interpolation': The meaning and the use of this parameter is described with the operator gen_measure_rectangle2 by the parameter Interpolation.

  'measure_distance_min': Returns the minimum distance between the centers of the generated measure regions, which depends on the geometry of the object and the value of the input parameter *'measure_distance'* or the value of the input parameter *'num_measures'* of the operator set_metrology_object_param. For a metrology object circle or a metrology object line the distances between measure regions are uniformly distributed. Therefore, *'measure_distance_min'* and *'measure_distance_max'* return the same value.

  'measure_distance_max': Returns the maximum distance between the centers of the generated measure regions, which depends on the geometry of the object and the value of the input parameter *'measure_distance'* or the value of the input parameter *'num_measures'* of the operator set_metrology_object_param. For a metrology object circle or a metrology object line the distances between measure regions are uniformly distributed. Therefore, *'measure_distance_min'* and *'measure_distance_max'* return the same value.

  'num_measures': Returns the number of generated measure regions, which depends on the geometry of the object and the value of the input parameter *'measure_distance'* or the value of the input parameter *'num_measures'* of the operator set_metrology_object_param.

'object_type': Type of the geometric shape of the metrology object. For a metrology object of type circle, the output parameter GenParamValue contains the value *'circle'*. For a metrology object of type ellipse, the output parameter GenParamValue contains the value *'ellipse'*. For a metrology object of type line, the output parameter GenParamValue contains the value *'line'*. For a metrology object of type rectangle, the output parameter GenParamValue contains the value *'rectangle'*.

'object_params': The parameters of the geometric shape of the metrology object. For a metrology object of type circle, the output parameter GenParamValue contains the geometry of the circle in the following order: *'row'*, *'column'*, *'radius'*. The meaning and the use of these parameters is described with the operator add_metrology_object_circle_measure. For a metrology object of type ellipse, the output parameter GenParamValue contains the geometry of the ellipse in the following order: *'row'*, *'column'*, *'phi'*, *'radius1'*, *'radius2'*. The meaning and the use of these parameters is described with the operator add_metrology_object_ellipse_measure. For a metrology object of type line, the output parameter GenParamValue contains the geometry of the line in the following order: *'row_begin'*, *'column_begin'*, *'row_end'*, *'column_end'*. The meaning and the use of these parameters is described with the operator add_metrology_object_line_measure. For a metrology object of type rectangle, the output parameter GenParamValue contains the geometry of the rectangle in the following order: *'row'*, *'column'*, *'phi'*, *'length1'*, *'length2'*. The meaning and the use of these parameters is described with the operator add_metrology_object_rectangle2_measure.

- Only valid for a metrology object of type circle:

  'row', *'column'*, *'radius'*: These are parameters for a metrology object of type circle. The meaning and the use of these parameters is described with the operator add_metrology_object_circle_measure.

- Only valid for a metrology object of type ellipse:

  'row', *'column'*, *'phi'*, *'radius1'*, *'radius2'*: These are parameters for a metrology object of type ellipse. The meaning and the use of these parameters is described with the operator add_metrology_object_ellipse_measure.

- Only valid for a metrology object of type line:

  'row_begin', *'column_begin'*, *'row_end'*, *'column_end'*: These are parameters for a metrology object of type line. The meaning and the use of these parameters is described with the operator add_metrology_object_line_measure.

- Only valid for a metrology object of type rectangle:

  'row', *'column'*, *'phi'*, *'length1'*, *'length2'*: These are parameters for a metrology object of type rectangle. The meaning and the use of these parameters is described with the operator add_metrology_object_rectangle2_measure.

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: Index ∈ {'all', 0, 1, 2}

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: 'num_measures'
List of values: GenParamName ∈ {'column', 'column_begin', 'column_end', 'distance_threshold', 'end_phi', 'instances_outside_measure_regions', 'length1', 'length2', 'max_num_iterations', 'measure_distance_min', 'measure_distance_min', 'measure_interpolation', 'measure_length1', 'measure_length2', 'measure_select', 'measure_sigma', 'measure_threshold', 'measure_transition', 'min_score', 'num_instances', 'num_measures', 'object_params', 'object_type', 'phi', 'point_order', 'radius', 'radius1', 'radius2', 'rand_seed', 'row', 'row_begin', 'row_end', 'start_phi', 'x', 'y', 'x_begin', 'y_begin', 'x_end', 'y_end'}

▷ **GenParamValue** (output_control) . . . . . attribute.value-array ⇝ *string* / real / integer
Values of the generic parameters.

Result

If the parameters are valid, the operator `get_metrology_object_param` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_metrology_object_indices`, `set_metrology_object_param`

Possible Successors

`set_metrology_object_param`

See also

`get_metrology_object_fuzzy_param`, `get_metrology_object_num_instances`

Module

2D Metrology

`get_metrology_object_result` `( : : MetrologyHandle, Index, Instance, GenParamName, GenParamValue : Parameter )`

Get the results of the measurement of a metrology model.

`get_metrology_object_result` allows to access the results of a measurement obtained by `apply_metrology_model` for the metrology objects of the metrology model `MetrologyHandle`.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The parameter `Index` specifies for which metrology objects the results are queried. For `Index` set to *'all'*, the results of all metrology objects are returned. With the parameter `Instance` it can be specified, which instances of the results are returned in `Parameter`. The results for all instances are returned by setting `Instance` to *'all'*. Different generic parameters can be used to control the returned values in `Parameter`. The generic parameter names are passed in `GenParamName`. The corresponding values are passed in `GenParamValue`. The following parameters and values are possible:

'result_type': If `GenParamName` is set to *'result_type'*, then `GenParamValue` allows to control how and what results are returned for a metrology object. All measured parameters of the queried metrology object can be queried at once, specific parameters can be queried individually or the score for the metrology object can be queried.

'Obtaining all parameters': If `GenParamValue` is set to *'all_param'*, then all measured parameters of a metrology object are returned. If camera parameters and a pose have been set (see `set_metrology_model_param`), the results are returned in metric coordinates, otherwise in pixels.
For a *circle*, the return values are the coordinates of the center and the radius of the circle. The order is [*'row'*, *'column'*, *'radius'*] or [*'x'*, *'y'*, *'radius'*] respectively.
For an *ellipse*, the return values are the coordinates of the center, the orientation of the major axis *'phi'*, the length of the larger half axis *'radius1'*, and the length of the smaller half axis *'radius2'* of the ellipse. The order is [*'row'*, *'column'*, *'phi'*, *'radius1'*, *'radius2'*] or [*'x'*, *'y'*, *'phi'*, *'radius1'*, *'radius2'*] respectively.
For a *line*, the start and end point of the line is returned. The order is [*'row_begin'*, *'column_begin'*, *'row_end'*, *'column_end'*] or [*'x_begin'*, *'y_begin'*, *'x_end'*, *'y_end'*]
For a *rectangle*, the return values are the coordinates of the center, the orientation of the main axis *'phi'*, the length of the larger half edge *'length1'*, and the length of the smaller half edge *'length2'* of the rectangle. The order is [*'row'*, *'column'*, *'phi'*, *'length1'*, *'length2'*] or [*'x'*, *'y'*, *'phi'*, *'length1'*, *'length2'*] respectively.

'Obtaining specific parameters': Measured object parameters can also be queried individually by providing the desired parameter name in `GenParamName`.

When *no camera parameters* and no measurement plane are set, the following parameters can be queried individually, depending on whether they are available for the respective object. Note that for lines, additionally the 3 parameters of the hessian normal form can be queried, i.e., the unit normal vector *'nrow'*, *'ncolumn'* and the orthogonal distance *'distance'* of the line from the origin of the coordinate system. The sign of the distance determines the side of the line on which the origin is located.
List of values: *'row'*, *'column'*, *'radius'*, *'phi'* , *'radius1'*, *'radius2'*, *'length1'*, *'length2'*, *'row_begin'*, *'column_begin'*, *'row_end'*, *'column_end'*, *'nrow'*, *'ncolumn'*, *'distance'*
If *camera parameters* and a measurement plane was set, the parameters are returned in metric coordinates, the following parameters can be queried individually, depending on whether they are available for the respective object. Note that for lines, additionally the 3 parameters of the hessian normal form can be queried, i.e., the unit normal vector *'nx'*, *'ny'* and the orthogonal distance *'distance'* of the line from the origin of the coordinate system. The sign of the distance determines the side of the line on which the origin is located.
List of values: *'x'*, *'y'*, *'radius'*, *'phi'* , *'radius1'*, *'radius2'*, *'length1'*, *'length2'*, *'radius1'*, *'radius2'*, *'length1'*, *'length2'*, *'x_begin'*, *'y_begin'*, *'x_end'*, *'y_end'*, *'nx'*, *'ny'*, *'distance'*

***'Obtaining the score'*:** If `GenParamName` is set to the *'score'*, the fitting scores are returned. The score represents the number of measurements that are used for the calculation of the results divided by the maximum number of measure regions.

'used_edges': To query the edge points, that were actually used for a fitted metrology object, you can choose between following values for `GenParamValue`:

'row': Return the row coordinate of the edges that were used to fit the metrology object.

'column': Return the column coordinate of the edges that were used to fit the metrology object.

'amplitude': Return the edge amplitude of the edges that were used to fit the metrology object.

List of values: *'row'*, *'column'*, *'amplitude'*

'angle_direction': The parameter determines the rotation direction for angles that result from the fitting. Setting the parameter *'angle_direction'* to *'positive'* the angle is specified between the main axis of the object and the horizontal axis of the coordinate system in the mathematically positive direction (counterclockwise). Setting the parameter *'angle_direction'* to *'negative'* the angle is specified between the main axis of the object and the horizontal axis of the coordinate system in the mathematically negative direction (clockwise). The results of the angles are returned in radians.
List of values: *'positive'*, *'negative'*
Default: *'positive'*

It is possible to query the results of several metrology objects (see the parameter `Index`) and several instances (see the parameter `Instance`) of the metrology objects simultaneously. The results are returned in the following order in `Parameter`: 1st instance of 1st metrology object, 2nd instance of 1st metrology object, etc., 1st instance of 2nd metrology object, 2nd instance of 2nd metrology object, etc.

Parameters

▷ **`MetrologyHandle`** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **`Index`** (input_control) ..... integer(-array) ⇝ *integer* / string
Index of the metrology object.
Default: 0
Suggested values: `Index` ∈ {'all', 0, 1, 2}

▷ **`Instance`** (input_control) ..... integer(-array) ⇝ *string* / integer
Instance of the metrology object.
Default: 'all'
Suggested values: `Instance` ∈ {'all', 0, 1, 2}

▷ **`GenParamName`** (input_control) ..... attribute.name-array ⇝ *string*
Name of the generic parameter.
Default: 'result_type'
List of values: `GenParamName` ∈ {'result_type', 'angle_direction', 'used_edges'}

▷ **`GenParamValue`** (input_control) ..... attribute.value-array ⇝ *string* / real
Value of the generic parameter.
Default: 'all_param'
Suggested values: `GenParamValue` ∈ {'all_param', 'score', 'true', 'false', 'row', 'column', 'amplitude', 'radius', 'phi', 'radius1', 'radius2', 'length1', 'length2', 'row_begin', 'column_begin', 'row_end',

'column_end', 'nrow', 'ncolumn', 'distance', 'x', 'y', 'x_begin', 'y_begin', 'x_end', 'y_end', 'nx', 'ny', 'positive', 'negative'}

▷ **Parameter** (output_control) . . . . . real(-array) ⇝ *real* / integer / string
Result values.

Result

If the parameters are valid, the operator `get_metrology_object_result` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`apply_metrology_model`

Possible Successors

`clear_metrology_model`

See also

`get_metrology_object_result_contour`, `get_metrology_object_measures`

Module

2D Metrology

get_metrology_object_result_contour (
: Contour : MetrologyHandle, Index, Instance, Resolution :)

Query the result contour of a metrology object.

`get_metrology_object_result_contour` returns for the chosen metrology objects and object instances, the result contours of a measurement performed by `apply_metrology_model` in image coordinates.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle `MetrologyHandle`. The parameter `Index` specifies for which metrology objects the result contours are queried. For `Index` set to *'all'*, the result contours of all metrology objects are returned. If for a metrology object several results (instances) were computed, then the parameter `Instance` specifies, for which instances the result contours are returned in `Contour`. The result contours for all instances are obtained by setting `Instance` to *'all'*.

The resolution of the resulting contour `Contour` is controlled via `Resolution` containing the Euclidean distance between neighboring contour points in pixel. If the input value falls below the minimal possible value (1.192e-7), then the resolution is set internally to the smallest valid value.

Parameters

▷ **Contour** (output_object) . . . . . xld_cont(-array) ⇝ *object*
Result contour for the given metrology object.

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Index of the metrology object.
Default: 0
Suggested values: `Index` ∈ {'all', 0, 1, 2}

▷ **Instance** (input_control) . . . . . integer(-array) ⇝ *string* / integer
Instance of the metrology object.
Default: 'all'
Suggested values: `Instance` ∈ {'all', 0, 1, 2}

▷ **Resolution** (input_control) . . . . . real ⇝ *real*
Distance between neighboring contour points.
Default: 1.5
Restriction: Resolution >= 1.192e-7

Result

If the parameters are valid, the operator `get_metrology_object_result_contour` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`apply_metrology_model`

See also

`get_metrology_object_result`, `get_metrology_object_measures`

Module

2D Metrology

read_metrology_model (: : FileName : MetrologyHandle)

Read a metrology model from a file.

`read_metrology_model` reads a metrology model, which has been written to file with `write_metrology_model`, from the file `FileName`. The default HALCON file extension for a metrology model is 'mtr'. The values contained in the read metrology model are stored in a metrology model with the handle `MetrologyHandle`. Access to the parameters of the metrology model is possible, e.g., with the operator `get_metrology_object_param` or `get_metrology_object_fuzzy_param`.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: `.mtr`

▷ **MetrologyHandle** (output_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

Result

If the parameters are valid, the operator `read_metrology_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`get_metrology_object_indices`, `apply_metrology_model`

See also

`write_metrology_model`

Module

2D Metrology

reset_metrology_object_fuzzy_param (: : MetrologyHandle, Index :)

Reset all fuzzy parameters and fuzzy functions of a metrology model.

reset_metrology_object_fuzzy_param discards all fuzzy parameters and fuzzy functions of the metrology objects that can be set by the operator set_metrology_object_fuzzy_param and restores the default values.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle. The parameter Index determines the metrology objects to reset. With Index set to *'all'*, all metrology objects are reset.

Parameters

▷ **MetrologyHandle** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) ..... integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: Index ∈ {'all', 0, 1, 2}

Result

If the parameters are valid, the operator reset_metrology_object_fuzzy_param returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_metrology_object_fuzzy_param

See also

reset_metrology_object_param

Module

2D Metrology

reset_metrology_object_param (: : MetrologyHandle, Index :)

Reset all parameters of a metrology model.

reset_metrology_object_param discards all settings of the parameters for the metrology objects that can be set by the operator set_metrology_object_param and restores the default values.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle. The parameter Index determines the metrology objects to reset. With Index set to *'all'*, all metrology objects are reset.

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: Index ∈ {'all', 0, 1, 2}

Result

If the parameters are valid, the operator reset_metrology_object_param returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_metrology_object_param

See also

reset_metrology_object_fuzzy_param

Module

2D Metrology

serialize_metrology_model (: : MetrologyHandle : SerializedItemHandle)

Serialize a metrology model.

serialize_metrology_model serializes the data of a metrology model (see fwrite_serialized_item for an introduction of the basic principle of serialization).

The same data that is written in a file by write_metrology_model is converted to a serialized item. The metrology model is defined by the handle MetrologyHandle. The serialized metrology model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_metrology_model.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_metrology_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

create_metrology_model, add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_line_measure, add_metrology_object_rectangle2_measure, set_metrology_object_param, set_metrology_object_fuzzy_param, read_metrology_model

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_metrology_model

Module

2D Metrology

set_metrology_model_image_size (: : MetrologyHandle, Width, Height :)

Set the size of the image of metrology objects.

set_metrology_model_image_size is used to set or change the size of the image in which the edge detection that is related to a metrology model will be performed.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle. The image width must be specified by the parameter Width. The image height must specified by the parameter Height.

Attention

Note that the operator set_metrology_model_image_size should be called before adding metrology objects to the metrology model using the operators add_metrology_object_generic, add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_line_measure, or add_metrology_object_rectangle2_measure. Otherwise, all measure regions of existing metrology objects will be recomputed automatically upon calling set_metrology_model_image_size or apply_metrology_model.

Parameters

▷ **MetrologyHandle** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the image to be processed.
Default: 640
Suggested values: Width ∈ {128, 192, 256, 512, 640, 768, 1024, 1280, 2048}
Value range: 0 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 16

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the image to be processed.
Default: 480
Suggested values: Height ∈ {128, 192, 256, 512, 640, 768, 1024, 1280, 2048}
Value range: 0 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 16

Result

If the parameters are valid, the operator set_metrology_model_image_size returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_metrology_model

Possible Successors

set_metrology_model_param, add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_line_measure, add_metrology_object_rectangle2_measure, add_metrology_object_generic

Module

2D Metrology

set_metrology_model_param (: : MetrologyHandle, GenParamName, GenParamValue :)

Set parameters that are valid for the entire metrology model.

set_metrology_model_param sets or changes parameters that are valid for the entire metrology model MetrologyHandle.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The following values for GenParamName and GenParamValue are possible:

Calibration

If both internal camera parameters and the 3D pose of the measurement plane are set, apply_metrology_model calculates the results in metric coordinates.

'camera_param': Often the internal camera parameters are the result of calibrating the camera with the operator calibrate_cameras (see Calibration for the sequence of the parameters and the underlying camera model). It is possible to discard the internal camera parameters by setting *'camera_param'* to [].
Default: []

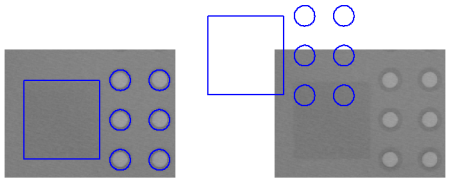
'plane_pose': The 3D pose of the measurement plane in camera coordinates. It is possible to discard the pose by setting *'plane_pose'* to [].
Default: []

Definition of a new reference system

When adding the metrology objects to the metrology model using e.g., add_metrology_object_generic, add_metrology_object_circle_measure etc. the positions and orientations are given with respect to the image coordinate system which has its origin in the upper left corner of the image. In some cases it may be necessary to change the reference system with respect to which the metrology objects are given. This is for instance the case when using a shape model to align the metrology model in a new image. The results from find_generic_shape_model can only be directly used in align_metrology_model if the reference system of the metrology model is the same as the system in which the shape model is given (see align_metrology_model for more details).

'reference_system': The tuple given in GenParamValue should contain [row, column, angle]. By default the reference system is the image coordinate system which has its origin in the top left corner. A new reference system is defined with respect to the image coordinate system by its translation (row,colum) and its rotation angle (angle). All components of the metrology model are converted into the new reference coordinate system. In the following figure, the reference system of the metrology model is set to the center of the image.

```
set_metrology_model_param(MetrologyHandle, 'reference_system',
[Height/2,Width/2,0])
```



(1) (2)

(1) Several metrology objects and their contours are shown in blue. (2) The new reference system for the metrology model is placed in the center of the image. As a consequence, the positions and orientations of the metrology objects are moved into the reverse direction. The resulting contours of the metrology objects are shown in blue.

Default: *[0, 0, 0]*

Scaling the results

The results of the measurement queried by `get_metrology_object_result` can be scaled by setting a scaling factor.

'scale': The parameter *'scale'* must be specified as the ratio of the *desired unit* to the *original unit*. If no camera parameters are given, the default unit is pixel.

If *'camera_param'* and *'plane_pose'* are set, the original unit is determined by the coordinates of the calibration object. Standard HALCON calibration plates are defined in metric coordinates. If it was used for the calibration, the desired unit can be set directly. The relation of units to scaling factors is given in the following table:

Unit	Scaling factor
m	1
dm	10
cm	100
mm	1000
um, microns	10^6

Suggested values: *1.0*, *0.1*, *'m'*, *'cm'*, *'mm'*, *'microns'*, *'um'*

Default: *1.0*

Parameters

▷ **`MetrologyHandle`** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'camera_param'
List of values: `GenParamName` ∈ {'camera_param', 'plane_pose', 'scale', 'reference_system'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer
Value of the generic parameter.
Default: []
Suggested values: `GenParamValue` ∈ {1.0, 0.1, 'm', 'cm', 'mm', 'microns', 'um'}

Result

If the parameters are valid, the operator `set_metrology_model_param` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_metrology_model, set_metrology_model_image_size

Possible Successors

add_metrology_object_generic, get_metrology_object_model_contour

See also

set_metrology_object_param, align_metrology_model, get_metrology_model_param

Module

2D Metrology

set_metrology_object_fuzzy_param (: : MetrologyHandle, Index, GenParamName, GenParamValue :)

Set fuzzy parameters or fuzzy functions for a metrology model.

set_metrology_object_param is used to set or change the fuzzy parameters or fuzzy functions of a metrology object in order to adapt the model to a particular edge selection before applying the operator apply_metrology_model.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle. The parameter Index specifies the metrology objects for which the parameters should be changed or set. The parameters of all metrology objects are set if the parameter Index is set to *'all'*.

The fuzzy parameter or the type of fuzzy function is passed in the parameter GenParamName. The corresponding value or the fuzzy function is passed in the parameter GenParamValue. If at least one fuzzy function is set, internally the operator fuzzy_measure_pos will be used when searching the objects with the operator apply_metrology_model. More information about fuzzy functions can be found with the operator fuzzy_measure_pos. The following generic parameters and parameter values for GenParamName and GenParamValue are possible:

'fuzzy_thresh': The parameter specifies the minimum fuzzy value. The meaning and the use of this parameter is described with the operator fuzzy_measure_pos. There, the parameter corresponds to the parameter FuzzyThresh.

Default: *0.5*

'function_contrast': The parameter specifies a fuzzy function of type contrast. The meaning and the use of this parameter is described with the operator set_fuzzy_measure. There, the parameter corresponds to the parameter SetType with the value *'contrast'* and its value corresponds to the parameter Function.

Default: *'disabled'*

'function_position': The parameter specifies a fuzzy function of type position. The meaning and the use of this parameter is described with the operator set_fuzzy_measure. There, the parameter corresponds to the parameter SetType with the value *'position'* and its value corresponds to the parameter Function.

Default: *'disabled'*

'function_position_center': The parameter specifies a fuzzy function of type position_center. The meaning and the use of this parameter is described with the operator set_fuzzy_measure. There, the parameter corresponds to the parameter SetType with the value *'position'* and its value corresponds to the parameter Function.

Default: *'disabled'*

'function_position_end': The parameter specifies a fuzzy function of type position_end. The meaning and the use of this parameter is described with the operator set_fuzzy_measure. There, the parameter corresponds to the parameter SetType with the value *'position_end'* and its value corresponds to the parameter Function.

Default: *'disabled'*

'function_position_first_edge': The parameter specifies a fuzzy function of type `position_first_edge`. The meaning and the use of this parameter is described with the operator `set_fuzzy_measure`. There, the parameter corresponds to the parameter `SetType` with the value *'position_first_edge'* and its value corresponds to the parameter `Function`.

Default: *'disabled'*

'function_position_last_edge': The parameter specifies a fuzzy function of type `position_last_edge`. The meaning and the use of this parameter is described with the operator `set_fuzzy_measure`. There, the parameter corresponds to the parameter `SetType` with the value *'position_last_edge'* and its value corresponds to the parameter `Function`.

Default: *'disabled'*

A fuzzy function is discarded if the fuzzy function value is set to *'disabled'*. All previously defined fuzzy functions and fuzzy parameters can be discarded completely using `reset_metrology_object_fuzzy_param`. The current configuration of the metrology objects can be accessed with `get_metrology_object_fuzzy_param`. Note that if at least one fuzzy function is specified, the operator `fuzzy_measure_pos` is used for the edge detection.

Parameters

▷ **`MetrologyHandle`** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **`Index`** (input_control) . . . . . integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: `Index` ∈ {'all', 0, 1, 2}

▷ **`GenParamName`** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: 'fuzzy_thresh'
List of values: `GenParamName` ∈ {'function_contrast', 'function_position', 'function_position_center', 'function_position_end', 'function_position_first_edge', 'function_position_last_edge', 'fuzzy_thresh'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value-array ⇝ *real* / integer
Values of the generic parameters.
Default: 0.5
Suggested values: `GenParamValue` ∈ {0.1, 0.3, 0.5, 0.6, 0.7, 0.9, 1, 2, 3, 4, 5, 10, 20}

Result

If the parameters are valid, the operator `set_metrology_object_fuzzy_param` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`get_metrology_object_fuzzy_param`

Possible Successors

`apply_metrology_model`, `reset_metrology_object_fuzzy_param`, `get_metrology_object_fuzzy_param`

See also

`set_metrology_object_param`

Module

2D Metrology

set_metrology_object_param (: : MetrologyHandle, Index, GenParamName, GenParamValue :)

Set parameters for the metrology objects of a metrology model.

set_metrology_object_param is used to set or change the different parameters of a metrology object.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle. The parameter Index specifies the metrology objects for which the parameters are set. The parameters of all metrology objects are set if the parameter Index is set to *'all'*. All parameters can also be set when creating a metrology object with add_metrology_object_generic, add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_line_measure, or add_metrology_object_rectangle2_measure. The current configuration of the metrology model can be accessed with get_metrology_object_param. All parameters that can be set with set_metrology_object_param can be reset with reset_metrology_object_param.

In the following all generic parameters with the default values are listed. But note that for a lot of applications the default values are sufficient and no adjustment is necessary. The following values for GenParamName and GenParamValue are possible - ordered by different categories:

Creating measure regions:

'measure_length1': The value of this parameter specifies the half length of the measure regions perpendicular to the metrology object boundary. It is equivalent to the measure tolerance. The unit of this value is pixel.

Suggested values: *10.0*, *20.0*, *30.0*

Default: *20.0*

Restriction: *'measure_length1'* >= *1.0*

'measure_length2': The value of this parameter specifies the half length of the measure regions tangential to the metrology object boundary. The unit of this value is pixel.

Suggested values: *3.0*, *5.0*, *10.0*

Default: *5.0*

Restriction: *'measure_length2'* >= *0.0*

'measure_distance': The value of this parameter specifies the desired distance between the centers of two measure regions. If the value leads to too few measure regions, the parameter has no influence and the number of measure regions will be increased to the minimum required number of measure regions (circle = 3, ellipse = 5, line = 2, rectangle = 2 per side = 8). The unit of this value is pixel.

If this value is set, the parameter *'num_measures'* has no influence.

Suggested values: *5.0*, *15.0*, *20.0*, *30.0*

Default: *10.0*

'num_measures': The value of this parameter specifies the desired number of measure regions.

The minimum number of measure regions depends on the type of the metrology object:

- Line: 2 measure regions
- Circle: 3 measure regions
- Circular arc: 4 measure regions
- Ellipse: 5 measure regions
- Elliptic arc: 6 measure regions
- Rectangle: 8 measure regions (2 regions each side)

If the chosen value is too low, *'num_measures'* is automatically set to the respective minimum value.

If this value is set, the parameter *'measure_distance'* has no influence.

Suggested values: *8*, *10*, *16*, *20*, *30*, *50*, *100*

Edge detection:

'measure_sigma': The parameter specifies the sigma for the Gaussian smoothing. The meaning, the use, and the default value of this parameter are described with the operator measure_pos by the parameter Sigma.

'measure_threshold': The parameter specifies the minimum edge amplitude. The meaning, the use, and the default value of this parameter are described with the operator measure_pos by the parameter Threshold.

'measure_select': The parameter specifies the selection of end points of the edges. The meaning, the use, and the default value of this parameter are described with the operator measure_pos by the parameter Select.

'measure_transition': The parameter specifies the use of dark/light or light/dark edges. The meaning and the use of the values *'all'*, *'positive'*, and *'negative'* for the parameter *'measure_transition'* is described with the operator measure_pos by the parameter Transition. Additionally, *'measure_transition'* can be set to the value *'uniform'*. Then, all positive edges (dark/light edges) and all negative edges (light/dark edges) are detected by the edge detection but when fitting the geometric shapes, the edges with different edge types are used separately, i.e., for each instance of a geometric shape either only the positive edges or the negative edges are used.

The measure direction within the measure regions is from the inside to the outside of the metrology object for objects of the types circle, ellipse, or rectangle. For metrology objects of the type line measure direction within the measure regions is from the left to the right, seen from the first point of the line (see RowBegin and ColumnBegin of the operator add_metrology_object_line_measure).

List of values: *'all'*, *'negative'*, *'positive'*, *'uniform'*

Default: *'all'*

'measure_interpolation': The parameter specifies the type of interpolation to be used. The meaning, the use and the default value of this parameter is described with the operator gen_measure_rectangle2 by the parameter Interpolation.

Fitting the geometric shapes:

'min_score': The parameter determines what score a potential instance must at least have to be regarded as a valid instance of the metrology object. The score is the number of detected edges that are used to compute the results divided by the maximum number of measure regions (see apply_metrology_model). If it can be expected that all edges of the metrology object are present, the parameter *'min_score'* can be set to a value as high as *0.8* or even *0.9*. Note that in images with a high degree of clutter or strong background texture the parameter *'min_score'* should be set to a value not much lower than *0.7* since otherwise false instances of a metrology object could be found.

Suggested values: *0.5*, *0.7*, *0.9*

Default: *0.7*

'num_instances': The parameter specifies the maximum number of successfully fitted instances of each metrology object after which the fitting will stop (see apply_metrology_model). Successfully fitted instances of the metrology objects must have a score of at least the value of *'min_score'*.

Suggested values: *1*, *2*, *3*, *4*

Default: *1*

'distance_threshold': apply_metrology_model uses a randomized search algorithm (RANSAC) to fit the geometric shapes. An edge point is considered to be part of a fitted geometric shape, if the distance of the edge point to the geometric shape does not exceed the value of *'distance_threshold'*.

Suggested values: *0*, *1.0*, *2.0*, *3.5*, *5.0*

Default: *3.5*

'max_num_iterations': The RANSAC algorithm estimates the number of iterations necessary for fitting the requested geometric shape. The estimation is based on the extracted edge data and the complexity of the shape. When setting the value of the parameter *'max_num_iterations'*, an upper limit for the computed number of iterations is defined. The number of iterations is still estimated by the RANSAC algorithm but cannot exceed the value of *'max_num_iterations'*. Setting this parameter can be helpful, if the quality of the fitting is not as important as observing time limits. However, if *'max_num_iterations'* is set too low, the algorithm will return low-quality or no results.

By default, *'max_num_iterations'* is set to -1, indicating that no additional upper limit is set for the number of iterations of the RANSAC algorithm.

Suggested values: *10*, *100*, *1000*

Default: *-1*

'rand_seed': The parameter specifies the seed for the random number generator for the RANSAC algorithm that is used by the selection of the edges the in operator apply_metrology_model. If the value of the parameter *'rand_seed'* is set to a number unequal to the value *0*, the operator yields the same result on every call with the same parameters, because the internally used random number generator is initialized with the value of the parameter *'rand_seed'*.

If the parameter *'rand_seed'* is set to the value *0*, the random number generator is initialized with the current time. In this case, the results are not reproducible.

Suggested values: *0*, *1*, *42*

Default: *42*

'instances_outside_measure_regions': The parameter specifies the validation of the results of measurements. If the value of the parameter *'instances_outside_measure_regions'* is set to the value *'false'*, only resulting instances of an metrology object are valid that are inside the major axis of the measure regions of this metrology object. Instances which are not valid are not stored. If the value of the parameter *'instances_outside_measure_regions'* is set to the value *'true'*, all instances of a metrology object are valid.

List of values: *'true'*, *'false'*

Default: *'false'*

Parameters

▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: Index ∈ {'all', 0, 1, 2}

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: 'num_instances'
List of values: GenParamName ∈ {'distance_threshold', 'instances_outside_measure_regions', 'max_num_iterations', 'measure_distance', 'measure_interpolation', 'measure_length1', 'measure_length2', 'measure_select', 'measure_sigma', 'measure_threshold', 'measure_transition', 'min_score', 'num_instances', 'num_measures', 'rand_seed'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / real / integer
Values of the generic parameters.
Default: 1
Suggested values: GenParamValue ∈ {1, 2, 3, 4, 5, 10, 20, 'all', 'true', 'false', 'first', 'last', 'positive', 'negative', 'uniform', 'nearest_neighbor', 'bilinear', 'bicubic'}

Result

If the parameters are valid, the operator set_metrology_object_param returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

get_metrology_object_param

Possible Successors

apply_metrology_model, reset_metrology_object_param, get_metrology_object_param

See also

set_metrology_object_fuzzy_param

Module

2D Metrology

write_metrology_model (: : MetrologyHandle, FileName :)

Write a metrology model to a file.

write_metrology_model writes a metrology model to the file FileName. The metrology model is defined by the handle MetrologyHandle. The metrology model can be read with read_metrology_model. The default HALCON file extension for a metrology model is 'mtr'.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

Attention

Note that only the input values are saved, i.e., no measure regions and no results obtained by the operator apply_metrology_model are saved.

Parameters

▷ **MetrologyHandle** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
File name.
File extension: .mtr

Result

If the parameters are valid, the operator write_metrology_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

apply_metrology_model

Possible Successors

clear_metrology_model

See also

read_metrology_model

Module

2D Metrology

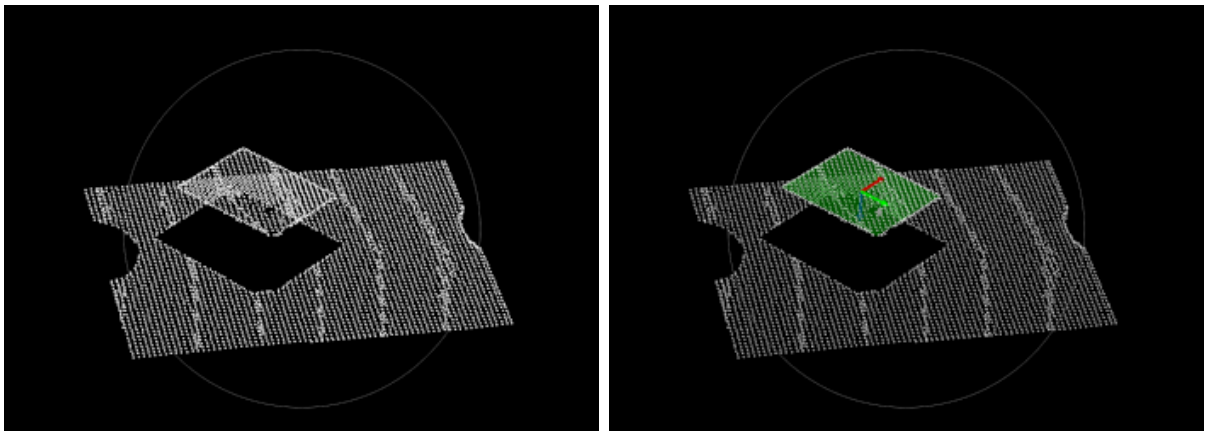
Chapter 3

3D Matching

This chapter gives an overview of the different 3D matching approaches available in HALCON.

3D Box Finder

As it is already contained in its name, the box finder can be used to locate box-shaped objects in 3D data. Thereby, no model of the object is needed as an input for the operator `find_box_3d`, but only the dimensions of the boxes to be found. As a result you can retrieve the pose of a gripping point, which can be especially useful in the case of a bin picking application.



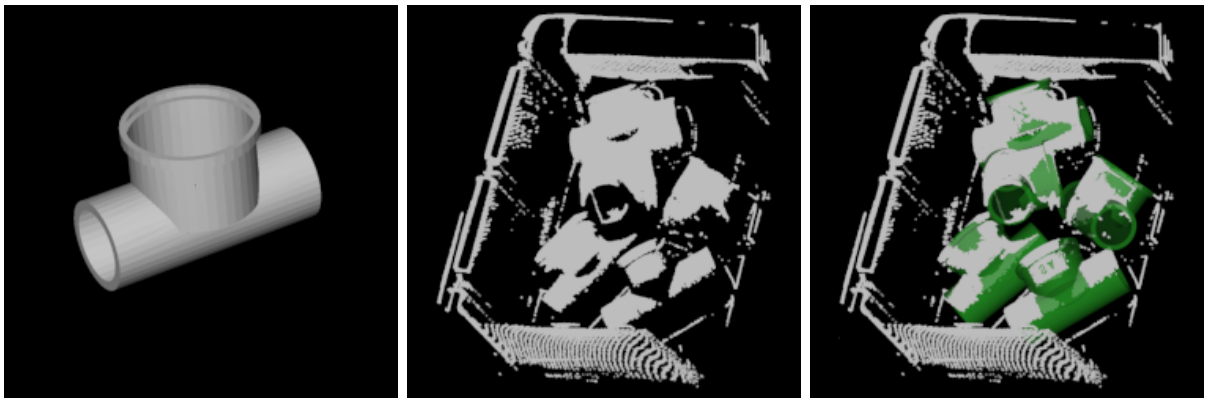
(1) (2)

(1) 3D input data (scene), (2) found instance, including a gripping point.

Surface-Based Matching

The surface-based matching approach is suited to locate more complex objects as well. The shape of these objects is passed to the operator `find_surface_model`, or `find_surface_model_image` respectively, in the form of a surface model. The poses of the found object instances in the scene are then returned.

Note that there are several different approaches when using surface-based matching. For detailed explanations regarding when and how to use these approaches, tips, tricks, and troubleshooting, have a look at the technical note on `Surface-Based Matching`.

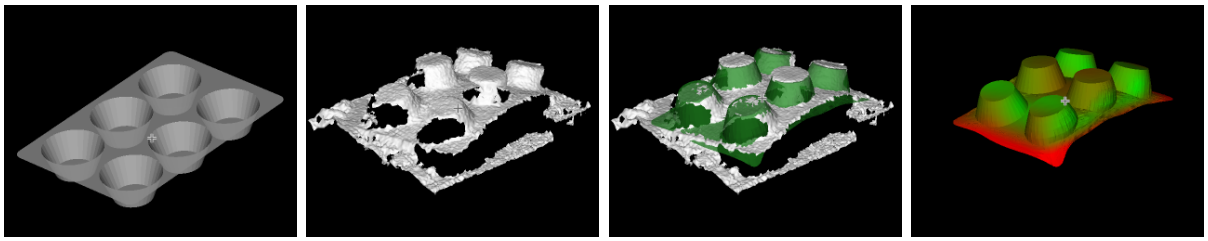


(1) (2) (3)

(1) 3D model to be searched for, (2) 3D input data (scene), (3) matching result.

Deformable Surface-Based Matching

In case an object can occur in the scene in different, deformed states you can use a deformable surface model to locate the object in the scene. If an instance of such an object is found by the operator `find_deformable_surface_model`, the object model can be retrieved featuring the respective deformation and pose.

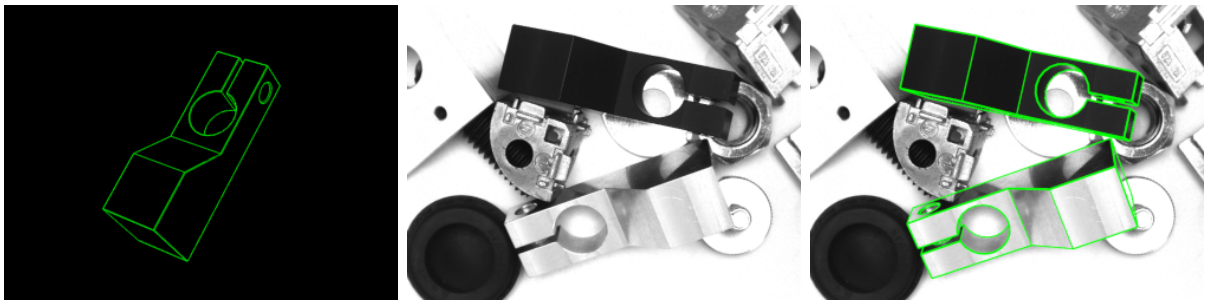


(1) (2) (3) (4)

(1) 3D object model, (2) 3D input data to be searched (scene), (3) model, transformed into the matched pose, (4) deformed object model.

Shape-Based Matching

With shape-based matching, instances of a 3D CAD model are searched in 2D images instead of 3D point clouds. For this, the edges of the wanted object need to be clearly visible in the image and the used camera needs to be calibrated beforehand. As a result, the object pose is computed and returned by the operator `find_shape_model_3d`.

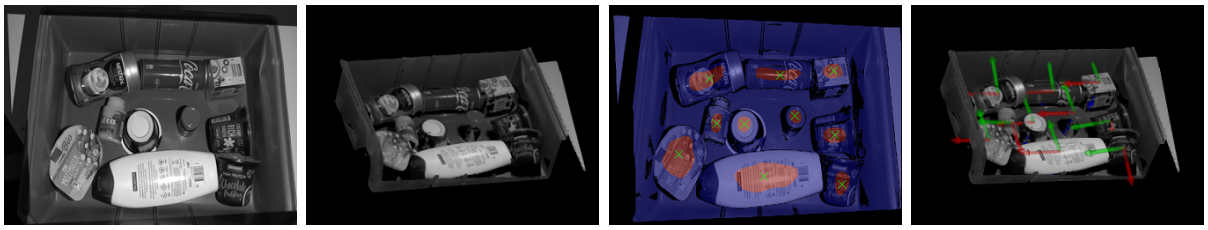


(1) (2) (3)

(1) 3D shape model, (2) input image, (3) found shape model, projected into the image.

3D Gripping Point Detection

3D Gripping Point Detection is a deep-learning-based approach to detect gripping points on arbitrary objects in a 3D scene. For further information please see the chapter 3D Matching / 3D Gripping Point Detection.



(1) (2) (3) (4)

(1) 2D input image (intensity image), (2) 3D scene (generated from XYZ-images), (3) Visualization of estimated gripping points on the 2D input image, (4) Visualization of estimated gripping points as poses in the 3D scene.

Deep 3D Matching

Deep 3D Matching is a deep-learning-based approach to detect objects in a scene and compute their 3D pose. For further information please see the chapter 3D Matching / Deep 3D Matching.

(1) (2) (3)

(1) Input scenes for an object, (2) computed 3D poses of the object in 2D image, (3) computed 3D poses of the object in 3D plot.

3.1 3D Box

```
find_box_3d ( : : ObjectModel3DScene, SideLen1, SideLen2, SideLen3,
    MinScore, GenParam : GrippingPose, Score, ObjectModel3DBox,
    BoxInformation )
```

Find boxes in 3D data.

`find_box_3d` finds boxes in the 3D object model `ObjectModel3DScene` and returns the pose of a gripping point `GrippingPose`, a 3D object model `ObjectModel3DBox` a score value `Score` and a dictionary `BoxInformation`, containing further information about the found boxes, for each found box.

The side lengths of the boxes are passed in `SideLen1`, `SideLen2`, and `SideLen3`. Each length consists of a tuple of two values, indicating the minimum and maximum length of that side. If only a single face of the box is expected to be visible, or no restriction should be applied to the remaining box length, `SideLen3` can be set to *-1*.

The parameter `MinScore` sets the minimum score for boxes to be returned. Boxes with a score smaller than this value will not be returned.

`ObjectModel3DScene` must contain a XYZ-mapping, like it is, e.g., the case, when creating it with `xyz_to_object_model_3d`.

Typical Workflow

A typical workflow for detecting 3D boxes in 3D data looks as follows:

1. Obtain the 3D data either as XYZ-images, or directly as a 3D object model with XYZ-mapping.

2. Remove as much background and clutter that is not part of any box from the scene as possible, in order to increase robustness and speed. Therefore, e.g., use `threshold` and `reduce_domain` on the XYZ-images before calling `xyz_to_object_model_3d`. Further options are described in the section "Troubleshooting" below.
3. If the 3D data exists in the form of XYZ-images, convert them to a 3D object model using `xyz_to_object_model_3d`.
4. Obtain the approximate box edge lengths that should be found. Note that changing those lengths later on might make it necessary to also change other parameters, such as `MinScore`.
5. Call `find_box_3d`, passing the 3D object model with the scene and the approximate box edge lengths.
6. Use the procedure `visualize_object_model_3d` to visualize the results, if necessary.

Understanding the Results

The boxes are returned in several ways.

First, the pose of a gripping point is returned in `GrippingPose`. The used side of the box and the z-axis of the gripping pose are set according to the XYZ-mapping. If only a single side of the box is visible, the center of the gripping pose is in the center of that side, and its z-axis is oriented away from the viewing point of the XYZ-mapping. If multiple sides of the box are visible, the gripping pose lies in the center of the side that is most parallel to the viewing point of the XYZ-mapping. The z-axis is again oriented away from the viewing point of the XYZ-mapping. The x-axis of the gripping pose is set to the box axis that is roughly the most aligned with the column direction of the XYZ-mapping. The y-axis is computed based on the x- and z-axis.

The box is also returned in triangulated form in `ObjectModel3DBox`. This allows a quick visualization of the results.

For each found box, a score between *0* and *1* is returned in `Score`. The score indicates how well the box and its edges are visible, and how well the found box matches the specified dimensions.

Finally, additional information about the results is returned in the dictionary `BoxInformation`. `get_dict_param` and `get_dict_tuple` can be used to obtain further information about the results. Also, the HDevelop handle inspect window can be used to inspect the returned dictionary.

The dictionary `BoxInformation` contains the following keys:

`results`: This key references a dictionary containing the found boxes. They are sorted according to their score in descending order with ascending integer keys starting at `0`.

Each box result is a dictionary with the following keys:

`box_pose`: This is the box's pose in the coordinate system of the scene. This pose is used for visualizing the found box.

`box_length_x`, `box_length_y`, `box_length_z`: The side lengths of the found box corresponding to `box_pose`. `box_length_x` and `box_length_y` will always contain a positive number. If only a single side of the box is visible, `box_length_z` will be set to *0*.

`gripping_pose`: The same pose as returned in `GrippingPose`.

`gripping_length_x`, `gripping_length_y`, `gripping_length_z`: The side lengths of the found box corresponding to `GrippingPose`. `gripping_length_x` and `gripping_length_y` will always contain a positive number. If only a single side of the box is visible, `gripping_length_z` will be set to *0*.

`score`: The same score as returned in `Score`.

`one_side_only`: Boolean indicating whether only one side of the box is visible (*'true'*) or not (*'false'*).

`gen_param`: This is a dictionary with the parameters passed to `find_box_3d`. `SideLen1`, `SideLen2`, and `SideLen3` are pooled in a tuple with key `lengths`. The key `min_score` references `MinScore`. The other keys are denoted analogously to the generic parameters of the dictionary `GenParam`.

`sampled_edges`: This is the 3D object model with sampled edges. It contains the viewing direction of the edge points as normal vectors.

`sampled_edges_direction`: This is the 3D object model with sampled edges (same as for key `sampled_edges`. It contains the edge directions of the edge points as normal vectors.

`sampled_scene`: This is the sampled scene in which the boxes are looked for. It can be used for visualization or debugging the sampling distance.

sampled_reference_points: This is a 3D object model with all points from the 3D scene that were used as reference points in the matching process. For each reference point, the optimum pose of the box is computed under the assumption that the reference point lies on the surface of the box.

Generic Parameters

Additional parameters can be passed as key/tuple pairs in the dictionary GenParam in order to improve the matching process. The following parameter names serve as keys to their corresponding tuples (see create_dict and set_dict_tuple).

3d_edges: Allows to manually set the 3D scene edges. The parameter must be a 3D object model handle. The edges are usually a result of the operator edges_object_model_3d but can further be filtered in order to remove outliers. If this parameter is not given, find_box_3d will internally extract the 3D edges similar to the operator edges_object_model_3d.

3d_edge_min_amplitude: Sets the minimum amplitude of a discontinuity in order for it to be classified as an edge. Note that if edges were passed manually with the generic parameter 3d_edges, this parameter is ignored. Otherwise, it behaves similar to the parameter MinAmplitude of the operator edges_object_model_3d.

Restriction: 3d_edge_min_amplitude >= *0*

Default: 10% of the smallest box diagonal.

max_gap: If no edges are passed with 3d_edges, the operator will extract 3D edges internally. The parameter can be used to control the edge extraction.

max_gap has the same meaning as in edges_object_model_3d.

remove_outer_edges: Removes the outermost edges when set to *'true'*. This is for example helpful for bin picking applications in order to remove the bin.

List of values: *'false'*, *'true'*

Default: *'false'*

max_num_boxes: Limits the number of returned boxes. By default, find_box_3d will return all detected boxes with a score larger than MinScore. This parameter can be used to limit the number of boxes respectively.

Default: *0* (return all boxes)

box_type: Sets the type of boxes to search for. For *'full_box_visible'* only boxes with more than one side visible are returned. If *'single_side_visible'* is set, boxes with only one visible side are searched for. If further box sides are visible nonetheless, they are ignored. For *'all'* both types are returned.

List of values: *'all'*, *'single_side_visible'*, *'full_box_visible'*

Default: *'all'*

Troubleshooting

***Visualizing extracted edges and sampled scene*:** To debug the box detector, some of the internally used data can be visualized by obtaining it from the returned dictionary BoxInformation, using get_dict_tuple.

The sampled 3D scene can be extracted with the key sampled_scene. Finding smaller boxes requires a denser sampling and subsequently slows down the box detection.

The sampled 3D edges can be extracted with the key sampled_edges and sampled_edges_directions. Both 3D object models contain the same points, however, sampled_edges contains the viewing direction of the edge points as normal vectors, while sampled_edges_directions contains the edge directions of the edge points as normal vectors. Note that the edge directions should be perpendicular to the edges, pointing outwards of the boxes.

***Improve performance*:** If find_box_3d is taking too long, the following steps might help to increase its performance.

- **Remove more background and clutter**: A significant improvement in runtime and detection accuracy can usually be achieved by removing as much of the background and clutter from the 3D scene as possible.
  The most common approaches for removing unwanted data are:
  - Thresholding the X-, Y- and Z-coordinates, either by using threshold and reduce_domain on the XYZ-images before calling xyz_to_object_model_3d, or by using select_points_object_model_3d directly on the 3D object model that contains the scene.

  - Some sensors return an intensity image along with the 3D data. Filters on the intensity image can be used to remove parts of the image that contain background.
  - Use background subtraction. If the scene is static, for example, if the sensor is mounted in a fixed position over a conveyor belt, the XYZ-images of the background can be acquired once without any boxes in it. Afterwards, `sub_image` and `threshold` can be used on the Z-images to select parts of the 3D data that are not part of the background.
- **Increase minimum score**: An increased minimum score `MinScore` might lead to more boxes being removed earlier in the detection pipeline.
- **Increase the smallest possible box**: The smaller the smallest possible box side is, the slower `find_box_3d` runs. For example, if all boxes are usually seen from a single side, it might make sense to set `SideLen3` to *-1*. Additionally, `box_type` can be set to limit the type of boxes that are searched.
- **Manually computing and filtering edges**: The edges of the scene can be extracted manually, using `edges_object_model_3d`, and passed to `find_box_3d` using the generic parameter `3d_edges` (see above). Thus, the manual extraction can be used as a further way of filtering the edges.

Parameters

▷ **`ObjectModel3DScene`** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of 3D object model where to search the box.

▷ **`SideLen1`** (input_control) . . . . . real-array ⇝ *real*
Length of the first box side.

▷ **`SideLen2`** (input_control) . . . . . real-array ⇝ *real*
Length of the second box side.

▷ **`SideLen3`** (input_control) . . . . . real-array ⇝ *real*
Length of the third box side.
Default: -1

▷ **`MinScore`** (input_control) . . . . . real ⇝ *real* / integer
Minimum score of the returned boxes.
Default: 0.6
Restriction: 0 <= MinScore <= 1

▷ **`GenParam`** (input_control) . . . . . dict ⇝ *handle*
Dictionary for generic parameters.
Default: []

▷ **`GrippingPose`** (output_control) . . . . . pose(-array) ⇝ *real* / integer
Gripping poses of the detected boxes.

▷ **`Score`** (output_control) . . . . . real-array ⇝ *real*
Scores of the detected boxes.

▷ **`ObjectModel3DBox`** (output_control) . . . . . object_model_3d-array ⇝ *handle*
Detected boxes as triangulated 3D object models.

▷ **`BoxInformation`** (output_control) . . . . . dict ⇝ *handle*
Additional debug information as dictionary.

Result

If all parameters are valid and no error occurs, `find_box_3d` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`read_object_model_3d`, `xyz_to_object_model_3d`

Possible Successors

`gen_box_object_model_3d`, `get_dict_tuple`

Alternatives

find_surface_model

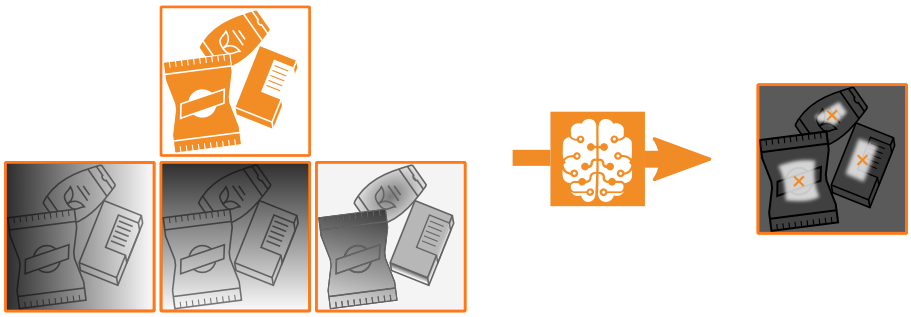
Module

3D Metrology

3.2 3D Gripping Point Detection

This chapter explains how to use 3D Gripping Point Detection.

3D Gripping Point Detection is used to find suitable gripping points on the surface of arbitrary objects in a 3D scene. The results can be used to target the gripping points with a robot arm and pick up the objects using vacuum grippers with suction cups.



A possible example for a 3D Gripping Point Detection application: A 3D scene (e.g., an RGB image and XYZ-images) is analyzed and possible gripping points are suggested.

HALCON provides a pretrained model which is ready for inference without an additional training step. To finetune the model for a specific task, it is possible to retrain it on a custom application domain. 3D Gripping Point Detection also works on objects that were not seen in training. Thus, there is no need to provide a 3D model of the objects that are to be targeted. 3D Gripping Point Detection can also cope with scenes containing various different objects at once, scenes with partly occluded objects, and with scenes containing cluttered 3D data.

The general inference workflow as well as the retraining are described in the following sections.

General Inference Workflow

This paragraph describes how to determine a suitable gripping point on arbitrary object surfaces using a 3D Gripping Point Detection model. An application scenario can be seen in the HDevelop example `3d_gripping_point_detection_workflow.hdev`.

1. Read the pretrained 3D Gripping Point Detection model by using
   - `read_dl_model`.
2. Set the model parameter regarding, e.g., the used devices or image dimensions using
   - `set_dl_model_param`.
3. Generate a data dictionary `DLSample` for each 3D scene. This can be done using the procedure
   - `gen_dl_samples_3d_gripping_point_detection`,

   which can cope with different kinds of 3D data. For further information on the data requirements see the section “Data” below.
4. Preprocessing of the data before the inference. For this, you can use the procedure
   - `preprocess_dl_samples`.

   The required preprocessing parameters can be generated from the model with

- create_dl_preprocess_param_from_model

or set manually using

- create_dl_preprocess_param.

Note that the preprocessing of the data has significant impact on the inference. See the section "3D scenes" below for further details.

5. Apply the model using the operator
   - apply_dl_model.
6. Perform a post-processing step on the resulting DLResult to retrieve gripping points for your scene using the procedure
   - gen_dl_3d_gripping_points_and_poses.
7. Visualize the 2D and 3D results using the procedure
   - dev_display_dl_data or
   - dev_display_dl_3d_data, respectively.

Training and Evaluation of the Model

This paragraph describes how the 3D Gripping Point Detection model can be retrained and evaluated using custom data. An application scenario can be seen in the HDevelop example 3d_gripping_point_detection_training_workflow.hdev.

Preprocess the data This part is about how to preprocess your data.

1. The information content of your dataset needs to be converted. This is done by the procedure
   - read_dl_dataset_3d_gripping_point_detection.

   It creates a dictionary DLDataset which serves as a database and stores all necessary information about your data. For more information about the data and the way it is transferred, see the section "Data" below and the chapter Deep Learning / Model.
2. Split the dataset represented by the dictionary DLDataset. This can be done using the procedure
   - split_dl_dataset.
3. The network imposes several requirements on the images. These requirements (for example the image size and gray value range) can be retrieved with
   - get_dl_model_param.

   For this you need to read the model first by using
   - read_dl_model.
4. Now you can preprocess your dataset. For this, you can use the procedure
   - preprocess_dl_dataset.

   To use this procedure, specify the preprocessing parameters as, e.g., the image size. Store all the parameter with their values in a dictionary DLPreprocessParam, for which you can use the procedure
   - create_dl_preprocess_param_from_model.

   We recommend to save this dictionary DLPreprocessParam in order to have access to the preprocessing parameter values later during the inference phase.

Training of the model This part explains the finetuning of the 3D Gripping Point Detection model by retraining it.

1. Set the training parameters and store them in the dictionary TrainParam. This can be done using the procedure

- `create_dl_train_param`.

2. Train the model. This can be done using the procedure
   - `train_dl_model`.

   The procedure expects:
   - the model handle `DLModelHandle`,
   - the dictionary `DLDataset` containing the data information,
   - the dictionary `TrainParam` containing the training parameters.

Evaluation of the retrained model In this part, we evaluate the 3D Gripping Point Detection model.

1. Set the model parameters which may influence the evaluation.
2. The evaluation can be done conveniently using the procedure
   - `evaluate_dl_model`.

   This procedure expects a dictionary `GenParam` with the evaluation parameters.
3. The dictionary `EvaluationResult` holds the evaluation measures. To get a clue on how the retrained model performed against the pretrained model you can compare their evaluation values. To understand the different evaluation measures, see section "Evaluation Measures for 3D Gripping Point Detection Results".

Data

This section gives information on the data that needs to be provided for the model inference or training and evaluation of a 3D Gripping Point Detection model.

As a basic concept, the model handles data by dictionaries, meaning it receives the input data from a dictionary `DLSample` and returns a dictionary `DLResult`. More information on the data handling can be found in the chapter Deep Learning / Model.

3D scenes 3D Gripping Point Detection processes 3D scenes, which consist of regular 2D images and depth information.

In order to adapt these 3D data to the network input requirements, a preprocessing step is necessary for the inference. See the section "Specific Preprocessing Parameters" below for information on certain preprocessing parameters. It is recommended to use a high resolution 3D sensor, in order to ensure the necessary data quality. The following data are needed:

2D image

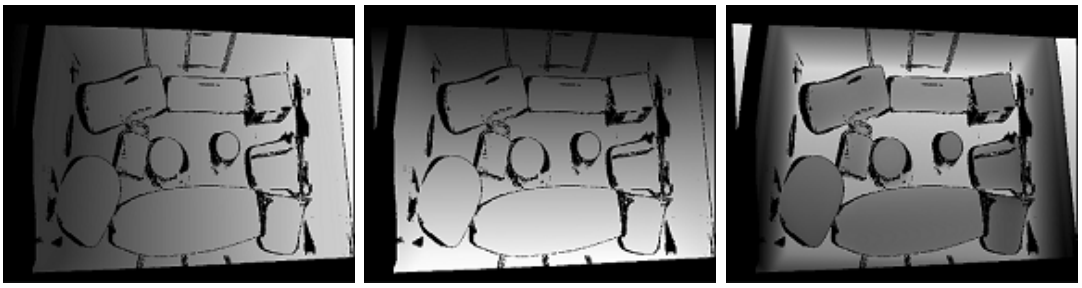
- RGB image, or
- intensity (gray value) image

Intensity image.

Depth information

- X-image (values need to increase from left to right)
- Y-image (values need to increase from top to bottom)

- Z-image (values need to increase from points close to the sensor to far points; this is for example the case if the data is given in the camera coordinate system)

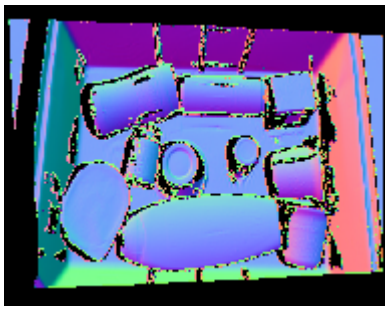


(1) (2) (3)

(1) X-image, (2) Y-image, (3) Z-image.

Normals (optional)

- 2D mappings (3-channel image)



Normals image.

Providing normal images improves the runtime, as this avoids the need for their computation.

In order to restrict the search area, the domain of the RGB/intensity image can be reduced. For details, see the section "Specific Preprocessing Parameters" below. Note that the domain of the XYZ-images and the (optional) normals images need to be identical. Furthermore, for all input data, only valid pixels may be part of the used domain.

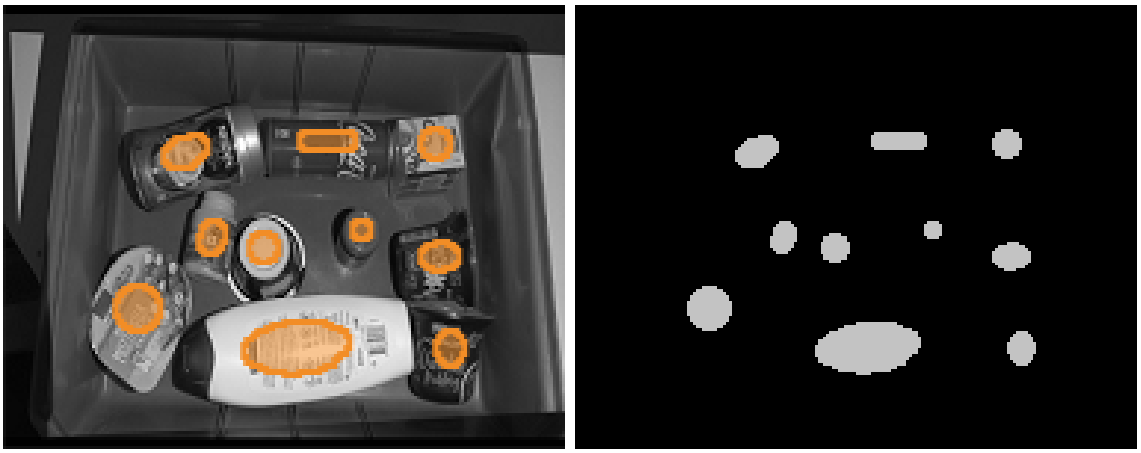
Data for Training and Evaluation The training data is used to train and evaluate a network specifically for your application.

The dataset needed for this consists of 3D scenes and corresponding information on possible gripping surfaces given as segmentation images. They have to be provided in a way the model can process them. Concerning the 3D scene requirements, find more information in the section "3D scenes" above.

How the data has to be formatted in HALCON for a DL model is explained in the chapter Deep Learning / Model. In short, a dictionary `DLDataset` serves as a database for the information needed by the training and evaluation procedures.

The data for `DLDataset` can be read using `read_dl_dataset_3d_gripping_point_detection`. See the reference of `read_dl_dataset_3d_gripping_point_detection` for information on the required contents of a 3D Gripping Point Detection `DLDataset`.

Along with 3D scenes, segmentation images need to be provided, which function as the ground truth. The segmentation images contain two gray values that denote every pixel in the scene to be either a valid gripping point or not. You can label your data using the MVTec Deep Learning Tool, available from the MVTec website.



(1) (2)

(1) Labeling of an intensity image. (2) Segmentation image, denoting gripping points (gray).

Make sure that the whole labeled area provides robust gripping points for the robot. Consider the following aspects when labeling your data:

- Gripping points need to be on a surface that can be accessed by the robot arm without being obstructed.
- Gripping points need to be on a surface that the robot arm can grip with its suction cup. Therefore, consider the object's material, shape, and surface tilt with regard to the ground plane.
- Take the size of the robots suction cup into account.
- Take the strength of the suction cup into account.
- Tend to label gripping points near the object's center of mass (especially for potentially heavier items).
- Gripping points should not be at an object's border.
- Gripping points should not be at the border of visible object regions.

Model output As inference output, the model will return a dictionary `DLResult` for every sample. This dictionary includes the following entries:

- `'gripping_map'`: Binary image, indicating for each pixel of the scene whether the model predicted a gripping point (pixel value = *1.0*) or not (*0.0*).
- `'gripping_confidence'`: Image, containing raw, uncalibrated confidence values for every point in the scene.

Evaluation Measures for 3D Gripping Point Detection Results

For 3D Gripping Point Detection, the following evaluation measures are supported in HALCON:

`mean_pro` Mean overlap of all ground truth regions labeled as gripping class with the predictions (Per-Region Overlap). See the paper referenced below for a detailed description of this evaluation measure.

`mean_precision` Mean pixel-level precision of the predictions for the gripping class. The precision is the proportion of true positives to all positives (true (TP) and false (FP) ones).

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

`mean_iou` Intersection over union (IoU) between the ground truth pixels and the predicted pixels of the gripping class. See Deep Learning / Semantic Segmentation and Edge Extraction for a detailed description of this evaluation measure.

gripping_point_precision Proportion of true positives to all positives (true and false ones).

For this measure, a true positive is a correctly predicted gripping point, meaning the predicted point is located within a ground truth region. However, only one gripping point per region is considered a true positive, additional predictions in the same region are considered false positives.

gripping_point_recall The recall is the proportion of the number of correctly predicted gripping points to the number of all ground truth regions of the gripping class.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

gripping_point_f_score To represent precision and recall with a single number, we provide the F-score, the harmonic mean of precision and recall.

$$\text{F-score} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

Postprocessing

The model results `DLResult` can be postprocessed with `gen_dl_3d_gripping_points_and_poses` in order to generate gripping points. Furthermore, this procedure can be parameterized in order to reject small gripping regions using `min_area_size`, or serve as a template to define custom selection criteria.

The procedure adds the following entry to the dictionary `DLResult`:

- `'gripping_points'`: Tuple of dictionaries containing information on suitable gripping points in a scene:
  - `'region'`: Connected region of potential gripping points. The determined gripping point lies inside this region.
  - `'row'`: Row coordinate of the gripping point in the preprocessed RGB/intensity image.
  - `'column'`: Column coordinate of the gripping point in the preprocessed RGB/intensity image.
  - `'pose'`: 3D pose of the gripping point (relative to the coordinate system of the XYZ-images, i.e., of the camera) which can be used by the robot.

Specific Preprocessing Parameters

In the preprocessing step, along with the data, preprocessing parameters need to be passed to `preprocess_dl_samples`. Two pairs of those preprocessing parameters have particularly significant impact:

- `'image_width'`, `'image_height'`: Determine the image dimensions of the images to be inferred.

  With larger image dimensions and thus a better resolution, smaller gripping surfaces can be detected. However, the runtime and memory consumption of the application increases.

- `'min_z'`, `'max_z'`: Determine the allowed distance from the camera for 3D points based on the Z-image.

  These parameters can therefore help to reduce erroneous outliers and therefore increase the application robustness.

A restriction of the search area can be done by reducing the domain of the input images (using `reduce_domain`). The way `preprocess_dl_samples` handles the domain is set using the preprocessing parameter `'domain_handling'`. The parameter `'domain_handling'` should be used in a way that only essential information is passed on to the network for inference. The following images show how an input image with reduced domain is passed on after the preprocessing step depending on the set `'domain_handling'`.

(1) (2) (3) (4)

(1) Input image with reduced domain (red), (2) image for `'full_domain'`, (3) image for `'keep_domain'`, (4) image for `'crop_domain'`.

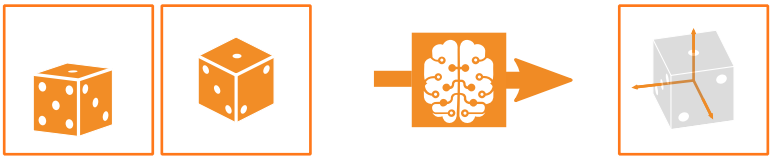
References

Bergmann, P., Batzner, K., Fauser, M., Sattlegger, D. and Steger, C., 2021. The MVTec anomaly detection dataset: a comprehensive real-world dataset for unsupervised anomaly detection. International Journal of Computer Vision, 129(4), pp.1038-1059.

3.3 Deep 3D Matching

This chapter explains how to use Deep 3D Matching.

Deep 3D Matching is used to accurately detect objects in a scene and compute their 3D pose. This approach is particularly effective for complex scenarios where traditional 3D matching techniques (like shape-based 3D matching) may struggle due to variations in object appearance, occlusions, or noisy data. Compared to surface-based matching, Deep 3D Matching works with a calibrated multi-view setup and does not require data from a 3D sensor.



A possible example for a Deep 3D Matching application: Images from different angles are used to detect an object. As a result the 3D pose of the object is computed.

The Deep 3D Matching model consists of two components, which are dedicated to two distinct tasks, the detection, which localizes objects, and the estimation of object poses. For a Deep 3D Matching application, both components need to be trained on the 3D CAD model of the object to be found in the application scenes.

Note: For now only inference is possible in HALCON, the custom training of a model will be available in a future version of HALCON. If you want to use the feature for your applications, please contact your HALCON sales partner for further information.

Once trained, the deep learning model can be used to infer the pose of the object in new application scenes. During the inference process, images from different angles are used as input.

General Inference Workflow

This paragraph describes how to determine a 3D pose using the Deep 3D Matching method. An application scenario can be seen in the HDevelop example `deep_3d_matching_workflow.hdev`.

1. Read the trained Deep 3D Matching model by using
   - `read_deep_matching_3d`.
2. Optimize the deep learning network for the use with -interfaces
   (a) Extract the detection network from the deep 3d matching model using

- get_deep_matching_3d_param.

(b) Optimize the parameter for inference with

- optimize_dl_model_for_inference.

(c) Set the optimized detection network using

- set_deep_matching_3d_param.

(d) Repeat these steps for the 3D pose estimation network.

(e) Save the optimized model using

- write_deep_matching_3d.

Note that the optimization of the model has significant impact on the runtime, if it is done with every inference run. So writing the optimized model saves time in the inference.

3. Set the camera parameters using

- set_deep_matching_3d_param.

4. Apply the model using the operator

- apply_deep_matching_3d.

5. Visualize the resulting 3D poses.

Training and Evaluation of the Model

For now only inference is possible in HALCON, training of a model will be available in a future version. If you want to use the feature for your applications, please contact your HALCON sales partner for further information.

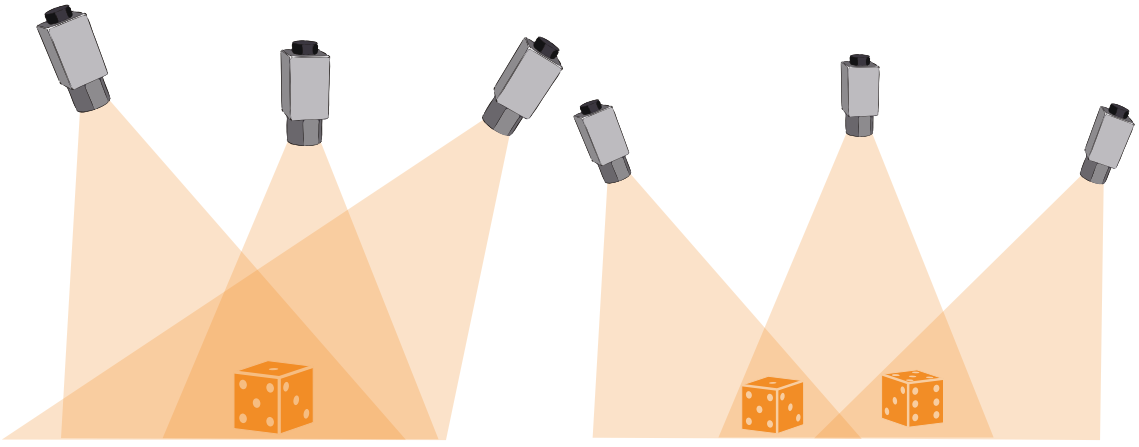
Data

This section gives information on the camera setup and data that needs to be provided for the model inference or training and evaluation of a Deep 3D Matching model.

As a basic concept, the model handles data by dictionaries, meaning it receives the input data from a dictionary `DLSample` and returns a dictionary `DeepMatchingResults`. More information on the data handling can be found in the chapter Deep Learning / Model.

Multi-View Camera Setup In order to use Deep 3D Matching with high accuracy you need a calibrated stereo or multi-view camera setup. In comparison to stereo reconstruction, Deep 3D Matching can deal with more strongly varying camera constellations and distances. Also there is no need to use 3D sensors in the setup. For information how to calibrate the used setup, please refer to the chapter Calibration / Multi-View.

The objects to be detected must be captured from two or more different perspectives in order to calculate the 3D poses.



(1) (2)

Example setups for Deep 3D Matching: Scenes are recorded by several cameras, the objects to be detected do not have to be seen by every single camera (but by at least two cameras).

Data for Training and Evaluation The training data is used to train and evaluate a Deep 3D Matching model specifically for your application.

The required training data is generated using CAD models. Synthetic images of the object are created from various angles, lighting conditions, and backgrounds. Note that there are no real images required, the required data is generated based of the CAD model.

The data needed for this is a CAD model and corresponding information on material, surface finish and color. Information about possible axial and radial symmetries can significantly improve the generated training data.

```
apply_deep_matching_3d (
    Images : : Deep3DMatchingModel : DeepMatchingResults )
```

Find the pose of objects using Deep 3D Matching.

The operator `apply_deep_matching_3d` finds instances of the object defined in `Deep3DMatchingModel` in the images `Images` and returns the detected instances and their 3D poses in `DeepMatchingResults`.

Input Images

`Images` must be an image array with exactly as many images as there are cameras set in the Deep 3D Matching model (see `set_deep_matching_3d_param`). The image resolutions must match the resolution of the corresponding camera parameters. The images must be either of type *'byte'* or *'float'*, and they must have 1 or 3 channels.

Deep Learning Models

`apply_deep_matching_3d` uses deep learning technology for detecting the object instances. For an efficient execution, it is strongly recommended to use appropriate hardware accelerators and to optimize the deep learning models. See `get_deep_matching_3d_param` on how to obtain the deep learning models in order to set the device on which they are executed and `optimize_dl_model_for_inference` for optimizing the models for a particular hardware.

Detection Steps

1. **Object Detection** The object detection deep learning model is used to find instances of the target object in all images.
2. **3D pose estimation** The pose estimation deep learning model is used to estimate the 3D pose of all instances found in the previous step. Poses of the same object found in different images are combined into a single instance.
3. **Pose Refinement** The poses found in the previous step are further refined using edges visible in the image. Additionally, their score is computed.
4. **Filter Results** The detected instances are filtered using the minimum score (*'min_score'*), the minimum number of cameras in which instances must be visible (*'min_num_views'*), as well es the maximum number of instances to return (*'num_matches'*).

Result Format

The results are returned in `DeepMatchingResults` as a dictionary. The dictionary key *'results'* contains all detected results. Each result has the following keys:

'score':
The score of the result instance.

'pose':
The pose of the result instance in world coordinate systems.

'cameras':
A tuple of integers containing the camera indices in which the instance was detected in.

Parameters

▷ **Images** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / real
Input images.

▷ **Deep3DMatchingModel** (input_control) . . . . . deep_matching_3d ⇝ *handle*
Deep 3D matching model.

▷ **DeepMatchingResults** (output_control) . . . . . dict-array ⇝ *handle*
Results.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

3D Metrology

get_deep_matching_3d_param (: : Deep3DMatchingModel, GenParamName : GenParamValue)

Read a parameter from a Deep 3D Matching model.

The operator get_deep_matching_3d_param returns the parameter values of GenParamName for the Deep 3D Matching model Deep3DMatchingModel in GenParamValue.

The following table gives an overview, which parameters can be set using set_deep_matching_3d_param and which can be retrieved using get_deep_matching_3d_param.

GenParamName	set	get
'camera_parameter N'	x	x
'camera_pose N'	x	x
'delete_cameras'	x	
'dl_model_detection'	x	x
'dl_model_pose_estimation'	x	x
'min_num_views'	x	x
'min_score'	x	x
'num_matches'	x	x
'orig_3d_model'		x

In the following the parameters are described:

'camera_parameter N', *'camera_pose N'*, *'delete_cameras'*:
These parameters control the camera setup used for matching, i.e., the camera poses and the camera parameters. *'delete_cameras'* can be set with an empty tuple as value to delete all cameras from a Deep 3D Matching model.

The keys for accessing the parameters of the N'th camera are *'camera_parameter N'* and *'camera_pose N'*, where N is the zero-based index of the camera. For example, to access the parameters of the first camera, use *'camera_parameter 0'* and *'camera_pose 0'*. Note that cameras must be added in order.

Further note that the camera parameters should not contain any distortion. It is recommended to remove any distortion from the camera parameters and images beforehand, using, for example, change_radial_distortion_cam_par in combination with change_radial_distortion_image or gen_radial_distortion_map and map_image.

The camera pose is the pose of the camera in an arbitrary world coordinate system. The poses of detected objects are returned in that world coordinate system. The angles must be passed in radians.

'dl_model_detection', *'dl_model_pose_estimation'*:
The deep learning models used for Deep 3D Matching. Both models are already pre-trained for the target object. They can be obtained and written back in order to, optimize it using `optimize_dl_model_for_inference` or change the device on which they are executed.

'min_num_views':
This parameter determines the minimum number of cameras in which an instance must be visible in order to be returned by `apply_deep_matching_3d`. The parameter can be either an integer larger than zero, or the string *'auto'*. If *'auto'*, instances must be visible in a single camera if only a single camera is used, and in at least two cameras otherwise.

Suggested values: *'auto'*, *2*, *3*

Default: *'auto'*

Value range: ≥ 0.

'min_score':
This parameter determines the minimum score of detected instances. In other words, `apply_deep_matching_3d` ignores all detected instances with a score smaller than this value. The score computed by the Deep 3D Matching model lies between *0* and *1*, where *0* indicates a bad match and *1* is a very good match.

Value range: $[0, \ldots, 1]$

Default: *0.2*

'num_matches':
This parameter determines the maximum number of matches to return by `apply_deep_matching_3d`. If the operator finds more instances than set in *'num_matches'*, only the *'num_matches'* instances with the highest scores are returned. This parameter can be set to zero, in which case all instances above *'min_score'* are returned.

Value range: ≥ 0.

Default: *0*

'orig_3d_model':
This parameter returns the original 3D CAD model used for creating the Deep 3D Matching model. It can be used to, visualize detection results.

Attention

Deep 3D Matching requires images to not have too much of a distortion. It is recommended to remove any distortion from the camera parameters and images beforehand, using, for example, `change_radial_distortion_cam_par` in combination with `change_radial_distortion_image` or `gen_radial_distortion_map` and `map_image`.

Parameters

▷ **`Deep3DMatchingModel`** (input_control) . . . . . deep_matching_3d ⇝ *handle*
Deep 3D Matching model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of parameter.
Default: 'min_score'
Suggested values: `GenParamName` ∈ {'min_score', 'num_matches', 'orig_3d_model', 'min_num_views', 'dl_model_detection', 'dl_model_pose_estimation', 'camera_parameter', 'camera_pose'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer / handle
Obtained value of parameter.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Module

3D Metrology

read_deep_matching_3d (: : FileName : Deep3DMatchingModel)

Read a Deep 3D Matching model from a file.

The operator read_deep_matching_3d reads a Deep 3D Matching model. Such models have to be in the HALCON format. As a result, the handle Deep3DMatchingModel is returned.

The model is loaded from the file FileName. The default HALCON file extension for Deep 3D Matching models is '.dm3'.

Please note that the values of runtime specific parameters are not written to file, see write_deep_matching_3d. As a consequence when reading a model these parameters are initialized with their default value, see get_deep_matching_3d_param.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Filename
File extension: .dm3

▷ **Deep3DMatchingModel** (output_control) ..... deep_matching_3d ⇝ *handle*
Handle of the Deep 3D Matching model.

Result

If the parameters are valid, the operator read_deep_matching_3d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_deep_matching_3d_param, get_deep_matching_3d_param, apply_deep_matching_3d

Module

3D Metrology

set_deep_matching_3d_param (: : Deep3DMatchingModel, GenParamName, GenParamValue :)

Set a parameter of a Deep 3D Matching model.

The operator set_deep_matching_3d_param sets the selected parameters GenParamName in the Deep 3D Matching model Deep3DMatchingModel to the values passed in GenParamValue.

The possible parameters are listed and described in get_deep_matching_3d_param.

Parameters

▷ **Deep3DMatchingModel** (input_control) ..... deep_matching_3d ⇝ *handle*
Deep 3D Matching model.

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Name of parameter.
Default: 'min_score'
Suggested values: GenParamName ∈ {'min_score', 'num_matches', 'min_num_views', 'dl_model_detection', 'dl_model_pose_estimation', 'camera_parameter', 'camera_pose', 'delete_cameras'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *string* / real / integer / handle
Value of parameter.
Default: 0.2

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Module

3D Metrology

write_deep_matching_3d (: : Deep3DMatchingModel, FileName :)

Write a Deep 3D Matching model in a file.

write_deep_matching_3d writes the Deep 3D Matching model Deep3DMatchingModel to the file given by FileName. Please note that the runtime specific parameters *'device'* and *'batch_size'* of the deep learning models are not written.

The default HALCON file extension for Deep 3D Matching models is '.dm3'.

The Deep 3D Matching model can be read with read_deep_matching_3d.

Parameters

▷ **Deep3DMatchingModel** (input_control) ..... deep_matching_3d ⇝ *handle*
Handle of the Deep 3D Matching model.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Filename
File extension: .dm3

Result

If the parameters are valid, the operator write_deep_matching_3d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_deep_matching_3d_param

Possible Successors

clear_handle

Module

3D Metrology

3.4 Deformable Surface-Based

add_deformable_surface_model_reference_point (
: : DeformableSurfaceModel, ReferencePointX, ReferencePointY,
ReferencePointZ : ReferencePointIndex)

Add a reference point to a deformable surface model.

The operator add_deformable_surface_model_reference_point adds one or more reference points to the deformable surface model passed in DeformableSurfaceModel. The 3D coordinates of the reference points is passed in the parameters ReferencePointX, ReferencePointY and ReferencePointZ. The index of the new reference points is returned in ReferencePointIndex.

Reference points are defined in model coordinates, i.e., in the coordinate frame of the model parameter of `create_deformable_surface_model`. The operators `find_deformable_surface_model` and `refine_deformable_surface_model` return the position of all added reference points as found in the scene.

Parameters

▷ **`DeformableSurfaceModel`** (input_control) . . . . . deformable_surface_model ⇝ *handle*
Handle of the deformable surface model.

▷ **`ReferencePointX`** (input_control) . . . . . real(-array) ⇝ *real* / integer
x-coordinates of a reference point.

▷ **`ReferencePointY`** (input_control) . . . . . real(-array) ⇝ *real* / integer
x-coordinates of a reference point.

▷ **`ReferencePointZ`** (input_control) . . . . . real(-array) ⇝ *real* / integer
x-coordinates of a reference point.

▷ **`ReferencePointIndex`** (output_control) . . . . . integer(-array) ⇝ *integer*
Index of the new reference point.

Result

`add_deformable_surface_model_reference_point` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- DeformableSurfaceModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_deformable_surface_model`, `read_deformable_surface_model`

Possible Successors

`find_deformable_surface_model`, `refine_deformable_surface_model`, `write_deformable_surface_model`

See also

`create_deformable_surface_model`, `find_deformable_surface_model`, `refine_deformable_surface_model`

Module

3D Metrology

`add_deformable_surface_model_sample` `( : : DeformableSurfaceModel, ObjectModel3D : )`

Add a sample deformation to a deformable surface model

The operator `add_deformable_surface_model_sample` adds the example deformation passed in `ObjectModel3D` to the deformable surface model `DeformableSurfaceModel`. The point cloud given in `ObjectModel3D` must have exactly as many points as the sampled deformation model, and is usually the result of the operator `find_deformable_surface_model` or `refine_deformable_surface_model`. The deformable surface model must have been created beforehand using, for example, `create_deformable_surface_model`. The operator re-trains the deformable surface model including the passed deformation. This allows `find_deformable_surface_model` to find deformations that are similar to the one given in `ObjectModel3D`.

Parameters

▷ **DeformableSurfaceModel** (input_control) . . . . . deformable_surface_model ⇝ *handle*
Handle of the deformable surface model.

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the deformed 3D object model.

Result

`add_deformable_surface_model_sample` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DeformableSurfaceModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_deformable_surface_model`, `find_deformable_surface_model`, `refine_deformable_surface_model`

Possible Successors

`find_deformable_surface_model`, `refine_deformable_surface_model`, `get_deformable_surface_model_param`, `write_deformable_surface_model`, `clear_deformable_surface_model`

Alternatives

`read_deformable_surface_model`

See also

`find_deformable_surface_model`, `refine_deformable_surface_model`, `read_deformable_surface_model`, `create_deformable_surface_model`, `write_deformable_surface_model`, `clear_deformable_surface_model`

Module

3D Metrology

clear_deformable_surface_matching_result (: : DeformableSurfaceMatchingResult :)

Free the memory of a deformable surface matching result.

The operator `clear_deformable_surface_matching_result` frees the memory of a deformable surface matching result that was created with `find_deformable_surface_model` or `refine_deformable_surface_model`. After calling `clear_deformable_surface_matching_result`, the result can no longer be used. The handle `DeformableSurfaceMatchingResult` becomes invalid.

Parameters

▷ **DeformableSurfaceMatchingResult** (input_control) . . . . . deformable_surface_matching_result(-array) ⇝ *handle*

Handle of the deformable surface matching result.

Result

If the handle of the result is valid, the operator `clear_deformable_surface_matching_result` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DeformableSurfaceMatchingResult

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`find_deformable_surface_model`, `refine_deformable_surface_model`

See also

`find_deformable_surface_model`, `refine_deformable_surface_model`

Module

3D Metrology

`clear_deformable_surface_model` (: : `DeformableSurfaceModel` :)

Free the memory of a deformable surface model.

The operator `clear_deformable_surface_model` frees the memory of a deformable surface model that was created, for example, by `read_deformable_surface_model` or `create_deformable_surface_model`. After calling `clear_deformable_surface_model`, the model can no longer be used. The handle `DeformableSurfaceModel` becomes invalid.

Parameters

▷ **`DeformableSurfaceModel`** (input_control) . . . . . deformable_surface_model(-array) ⇝ *handle*
Handle of the deformable surface model.

Result

If the handle of the model is valid, the operator `clear_deformable_surface_model` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DeformableSurfaceModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`read_deformable_surface_model`, `create_deformable_surface_model`

See also

`read_deformable_surface_model`, `create_deformable_surface_model`

Module

3D Metrology

`create_deformable_surface_model` (: : `ObjectModel3D`, `RelSamplingDistance`, `GenParamName`, `GenParamValue` : `DeformableSurfaceModel`)

Create the data structure needed to perform deformable surface-based matching.

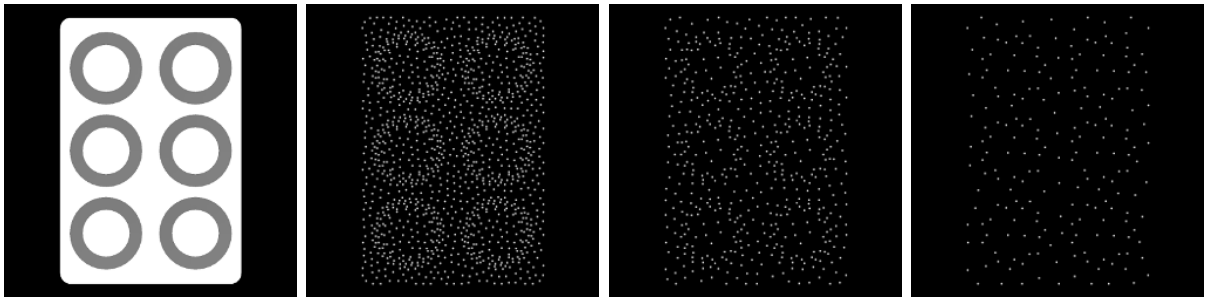
The operator create_deformable_surface_model creates a model for deformable surface-based matching for the 3D object stored in the 3D object model ObjectModel3D. The 3D object model can, for example, have been read previously from a file by using read_object_model_3d or it can have been created by using xyz_to_object_model_3d. The created surface model is returned in DeformableSurfaceModel.

The creation of the deformable surface model requires that the 3D object model contains points and normals. The following combinations are possible:

- points and point normals, e.g., from a call to surface_normals_object_model_3d
- points and a triangular or polygon mesh, e.g., from a CAD file
- points and a 2D-Mapping, e.g., from an XYZ image triple converted with xyz_to_object_model_3d

Note that the direction and orientation (inward or outward) of the normals of the model are important for matching.

The deformable surface model is created by sampling the 3D object model with a certain distance. The sampling distance must be specified in the parameter RelSamplingDistance and is parametrized relative to the diameter of the axis-parallel bounding box of the 3D object model. For example, if RelSamplingDistance is set to 0.05 and the diameter of ObjectModel3D is 10 cm, the points sampled from the object's surface will be approximately 5 mm apart. The sampled points can be obtained with the operator get_deformable_surface_model_param using the value *'sampled_model'*. Note that outlier points in the object model should be avoided, as they would corrupt the diameter. Reducing RelSamplingDistance leads to more points, and in turn to a more stable but slower matching. Increasing RelSamplingDistance leads to less points, and in turn to a less stable but faster matching.



(1) (2) (3) (4)

(1) Original 3D model. (2) 3D model sampled with RelSamplingDistance = *0.02*. (3) RelSamplingDistance = *0.03*. (4) RelSamplingDistance = *0.05*.

By default, deformable surface models created with create_deformable_surface_model can handle a moderate amount of deformation. The operator add_deformable_surface_model_sample can be used to add additional training samples, thus expanding the range of possible deformations. The amount of deformation that can be found can also be controlled with the generic parameters *'scale_min'*, *'scale_max'* and *'bending_max'* (see below).

The generic parameter pair GenParamName and GenParamValue is used to set additional parameters for the model generation. GenParamName contains the tuple of parameter names that shall be set and GenParamValue contains the corresponding values. The following values are possible for GenParamName:

'model_invert_normals': Invert the orientation of the surface normals of the model. The normal orientation needs to be known for the model generation. If both the model and the scene are acquired with the same setup, the normals will already point in the same direction. If the model was loaded from a CAD file, the normals might point into the opposite direction. If you experience the effect that the model is found on the 'outside' of the scene surface and the model was created from a CAD file, try to set this parameter to *'true'*. Also, make sure that the normals in the CAD file all point either outward or inward, i.e., are oriented consistently.

List of values: *'false'*, *'true'*

Default: *'false'*

'scale_min' **and** *'scale_max'*: The minimum and maximum allowed scaling of the model. Note that if you set one of the two parameters, the other one must be set too.

Suggested values: *0.8*, *1*, *1.2*

Default: No scaling

Restriction: *0* < *'scale_min'* < *'scale_max'*

'bending_max': Controls the maximum automatic deformation of the model. The model is deformed automatically by bending it with an angle up to the value of *'bending_max'*. This allows for deformations to be found that are within this bending range. The angle is passed in degrees.

Suggested values: *5*, *10*, *30*

Default: *20*

Restriction: *0 <= 'bending_max' < 90*

'stiffness': Control the stiffness of the model when performing the refinement. Larger values of this parameter lead to a more stiff model that can be less deformed. Smaller values lead to a less stiff model that allows more deformation.

Suggested values: *0.2*, *0.5*, *0.8*

Default: *0.5*

Restriction: *0 < 'stiffness' <= 1*

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the 3D object model.

▷ **RelSamplingDistance** (input_control) . . . . . real ⇝ *real*
Sampling distance relative to the object's diameter
Default: 0.05
Suggested values: RelSamplingDistance ∈ {0.1, 0.05, 0.03, 0.02, 0.01}
Restriction: 0 < RelSamplingDistance < 1

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters.
Default: []
Suggested values: GenParamName ∈ {'model_invert_normals', 'scale_min', 'scale_max', 'bending_max', 'stiffness'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false', 1, 0.9, 1.1, 5, 10, 20, 30, 0.05, 0.1, 0.2}

▷ **DeformableSurfaceModel** (output_control) . . . . . deformable_surface_model ⇝ *handle*
Handle of the deformable surface model.

Result

create_deformable_surface_model returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, get_object_model_3d_params

Possible Successors

add_deformable_surface_model_sample, add_deformable_surface_model_reference_point, find_deformable_surface_model, refine_deformable_surface_model, get_deformable_surface_model_param, write_deformable_surface_model, clear_deformable_surface_model

Alternatives

read_deformable_surface_model

See also

find_deformable_surface_model, refine_deformable_surface_model,

read_deformable_surface_model, add_deformable_surface_model_sample, add_deformable_surface_model_reference_point, write_deformable_surface_model, clear_deformable_surface_model

References

Bertram Drost, Slobodan Ilic: "Graph-Based Deformable 3D Object Matching." Proceedings of the 37th German Conference on Pattern Recognition, pp. 222-233, 2015.

Module

3D Metrology

deserialize_deformable_surface_model (: : SerializedItemHandle : DeformableSurfaceModel)

Deserialize a deformable surface model.

deserialize_deformable_surface_model deserializes a deformable surface model, that was serialized by serialize_deformable_surface_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized deformable surface model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created deformable surface model with the handle DeformableSurfaceModel.

Parameters

- ▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
  Handle of the serialized item.
- ▷ **DeformableSurfaceModel** (output_control) ..... deformable_surface_model ⇝ *handle*
  Handle of the deformable surface model.

Result

If the parameters are valid, the operator deserialize_deformable_surface_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_deformable_surface_model

Possible Successors

find_deformable_surface_model, refine_deformable_surface_model, get_deformable_surface_model_param, clear_deformable_surface_model

Alternatives

create_deformable_surface_model

See also

create_deformable_surface_model, read_deformable_surface_model, write_deformable_surface_model, serialize_deformable_surface_model

Module

3D Metrology

find_deformable_surface_model (: : DeformableSurfaceModel, ObjectModel3D, RelSamplingDistance, MinScore, GenParamName, GenParamValue : Score, DeformableSurfaceMatchingResult)

Find the best match of a deformable surface model in a 3D scene.

The operator find_deformable_surface_model finds the best match of the deformable surface model DeformableSurfaceModel in the 3D scene ObjectModel3D. The deformable surface model must have been created previously with, for example, create_deformable_surface_model.

The matching requires that the 3D object model ObjectModel3D contains points and normals. The scene shall provide one of the following options:

- points and point normals
- points and a 2D-Mapping, e.g., an XYZ image triple converted with xyz_to_object_model_3d

It is important for an accurate pose that the normals of the scene and the model point in the same direction (see *'scene_invert_normals'*). Note that triangles or polygons in the passed scene are ignored. Instead, only the vertices are used for matching. It is thus in general not recommended to use this operator on meshed scenes, such as CAD data. Instead, such a scene must be sampled beforehand using sample_object_model_3d to create points and normals. When using noisy point clouds, e.g., from time-of-flight cameras, the generic parameter *'scene_normal_computation'* should be set to *'mls'* in order to obtain more robust results (see below).

First, points are sampled uniformly from the scene passed in ObjectModel3D. The sampling distance is controlled with the parameter RelSamplingDistance, and is given relative to the diameter of the surface model. Decreasing RelSamplingDistance leads to more sampled points, and in turn to a more stable but slower matching. Increasing RelSamplingDistance reduces the number of sampled scene points, which leads to a less stable but faster matching. For an illustration showing different values for RelSamplingDistance, please refer to the operator create_deformable_surface_model.

The operator get_deformable_surface_matching_result can be used to retrieve the sampled scene points for visual inspection. For a robust matching it is recommended that at least 50-100 scene points are sampled for each object instance.

The method first finds an approximate position of the object. This position is then refined. The generic parameters controlling the deformation are described further down.

If a match was found, the score of the match is returned in Score and a deformable surface matching result handle is returned in DeformableSurfaceMatchingResult. Details of the matching result, such as the deformed model and the position of the reference points, can be queried with the operator get_deformable_surface_matching_result using the result handle.

The score is normalized between *0* and *1* and represents the amount of model surface visible in the scene. A value of *1* represents a perfect match. The parameter MinScore can be used to filter the result. A match is returned only if its score exceeds the value of MinScore.

The parameters GenParamName and GenParamValue are used to set generic parameters. Both get a tuple of equal length, where the tuple passed to GenParamName contains the names of the parameters to set, and the tuple passed to GenParamValue contains the corresponding values. The possible parameter names and values are described below.

'scene_normal_computation': This parameter controls the normal computation of the sampled scene. In the default mode *'fast'*, normals are computed based on a small neighborhood of points. In the mode *'mls'*, normals are computed based on a larger neighborhood and using the more complex but more accurate *'mls'* method. A more detailed description of the *'mls'* method can be found in the description of the operator surface_normals_object_model_3d. The *'mls'* mode is intended for noisy data, such as images from time-of-flight cameras.

List of values: *'fast'*, *'mls'*

Default: *'fast'*

'scene_invert_normals': Invert the orientation of the surface normals of the scene. The orientation of surface normals of the scene have to match with the orientation of the model. If both the model and the scene are acquired with the same setup, the normals will already point in the same direction. If you experience the effect that the model is found on the 'outside' of the scene surface, try to set this parameter to *'true'*. Also, make sure that the normals in the scene all point either outward or inward, i.e., are oriented consistently.

List of values: *'false'*, *'true'*

Default: *'false'*

'pose_ref_num_steps': Number of iterations for the refinement. Increasing the number of iteration leads to a more accurate position at the expense of runtime. However, once convergence is reached, the accuracy can no longer be increased, even if the number of steps is increased.

Suggested values: *1*, *10*, *25*, *50*

Default: *25*

Restriction: *'pose_ref_num_steps' > 0*

'pose_ref_dist_threshold_rel': Set the distance threshold for refinement relative to the diameter of the surface model. Only scene points that are closer to the object than this distance are used for the optimization. Scene points further away are ignored.

Suggested values: *0.05*, *0.1*, *0.25*, *0.3*

Default: *0.25*

Restriction: *0 < 'pose_ref_dist_threshold_rel'*

'pose_ref_dist_threshold_abs': Set the distance threshold for dense pose refinement as absolute value. See *'pose_ref_dist_threshold_rel'* for a detailed description. Only one of the parameters *'pose_ref_dist_threshold_rel'* and *'pose_ref_dist_threshold_abs'* can be set. If both are set, only the value of the last parameter is used.

Restriction: *0 < 'pose_ref_dist_threshold_abs'*

'pose_ref_scoring_dist_rel': Set the distance threshold for scoring relative to the diameter of the surface model. See the following *'pose_ref_scoring_dist_abs'* for a detailed description. Only one of the parameters *'pose_ref_scoring_dist_rel'* and *'pose_ref_scoring_dist_abs'* can be set. If both are set, only the value of the last parameter is used.

Suggested values: *0.1*, *0.05*, *0.03*, *0.005*

Default: *0.03*

Restriction: *0 < 'pose_ref_scoring_dist_rel'*

'pose_ref_scoring_dist_abs': Set the distance threshold for scoring. Only scene points that are closer to the object than this distance are considered to be 'on the model' when computing the score after the refinement. All other scene points are considered not to be on the model. The value should correspond to the amount of noise on the coordinates of the scene points. Only one of the parameters *'pose_ref_scoring_dist_rel'* and *'pose_ref_scoring_dist_abs'* can be set. If both are set, only the value of the last parameter is used.

Restriction: *0 < 'pose_ref_scoring_dist_abs'*

Parameters

▷ **`DeformableSurfaceModel`** (input_control) . . . . . deformable_surface_model ⇝ *handle*
Handle of the deformable surface model.

▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the 3D object model containing the scene.

▷ **`RelSamplingDistance`** (input_control) . . . . . real ⇝ *real*
Scene sampling distance relative to the diameter of the surface model.
Default: 0.05
Suggested values: `RelSamplingDistance` ∈ {0.1, 0.07, 0.05, 0.04, 0.03}
Restriction: 0 < RelSamplingDistance < 1

▷ **`MinScore`** (input_control) . . . . . real ⇝ *real* / integer
Minimum score of the returned match.
Default: 0
Restriction: MinScore >= 0

▷ **`GenParamName`** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: `GenParamName` ∈ {'scene_normal_computation', 'scene_invert_normals', 'pose_ref_num_steps', 'pose_ref_dist_threshold_rel', 'pose_ref_dist_threshold_abs', 'pose_ref_scoring_dist_abs', 'pose_ref_scoring_dist_rel'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value-array ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: `GenParamValue` ∈ {'fast', 'mls', 0, 1, 10, 25, 50, 0.05, 0.1, 0.25, 0.3, 0.05, 0.03, 0.005}

▷ **`Score`** (output_control) . . . . . real(-array) ⇝ *real*
Score of the found instance of the surface model.

▷ **DeformableSurfaceMatchingResult** (output_control) .....
deformable_surface_matching_result(-array) ⇝ *handle*

Handle of the matching result.

Result

`find_deformable_surface_model` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`read_object_model_3d`, `xyz_to_object_model_3d`, `get_object_model_3d_params`, `read_deformable_surface_model`, `create_deformable_surface_model`, `get_deformable_surface_model_param`, `add_deformable_surface_model_reference_point`, `add_deformable_surface_model_sample`

Possible Successors

`refine_deformable_surface_model`, `get_deformable_surface_matching_result`, `clear_deformable_surface_matching_result`, `clear_object_model_3d`

Alternatives

`refine_deformable_surface_model`

See also

`refine_deformable_surface_model`

Module

3D Metrology

get_deformable_surface_matching_result

```
get_deformable_surface_matching_result (
    : : DeformableSurfaceMatchingResult, ResultName,
    ResultIndex : ResultValue )
```

Get details of a result from deformable surface based matching.

The operator `get_deformable_surface_matching_result` returns details about the results of deformable surface based matching or the deformable surface refinement. The results are stored in `DeformableSurfaceMatchingResult`, which must have been created by `find_deformable_surface_model` or `refine_deformable_surface_model`.

The parameter `ResultName` is used to select which result detail shall be returned. For some result details, `ResultIndex` selects the index of the result detail. `ResultIndex` is ignored for certain values of `ResultName`.

The following values are possible for `ResultName`:

'sampled_scene': A 3D object model handle is returned that contains the sampled scene points that were used in the matching or refinement. This is helpful for tuning the sampling distance of the scene (see parameter RelSamplingDistance of operators `find_deformable_surface_model` and `refine_deformable_surface_model`). The parameter `ResultIndex` is ignored.

'rigid_pose': If `DeformableSurfaceMatchingResult` was created by `find_deformable_surface_model`, a rigid pose is returned that approximates the deformable matching result. The parameter `ResultIndex` is ignored. This parameter is not available if `DeformableSurfaceMatchingResult` was created by `refine_deformable_surface_model`.

'reference_point_x':

'reference_point_y':

'reference_point_z': Returns the x-, y- or z-coordinates of a transformed reference point. The reference point must have been added to the deformable surface model using the operator `add_deformable_surface_model_reference_point`. The indices of the reference points to be returned are passed in `ResultIndex`. If *'all'* is passed in `ResultIndex`, the position of all reference points is returned.

'deformed_model': Returns a deformed variant of the 3D object model that was originally passed to `create_deformable_surface_model`. The 3D object model is deformed with the reconstructed deformation. Triangles, polygons and extended attributes contained in the original 3D object model are maintained. The parameter `ResultIndex` is ignored.

'deformed_sampled_model': Returns a deformed variant of the 3D object model that was sampled by `create_deformable_surface_model`. The returned 3D object model has the same number of points as the original, undeformed sampled model, and the points are in the same order. Details about the sampling are described in `create_deformable_surface_model`. The original, undeformed sampled model can be obtained with `get_deformable_surface_model_param`. The parameter `ResultIndex` is ignored.

Parameters

▷ **`DeformableSurfaceMatchingResult`** (input_control) ..... deformable_surface_matching_result ⇝ *handle*

Handle of the deformable surface matching result.

▷ **`ResultName`** (input_control) ..... string(-array) ⇝ *string*
Name of the result property.
Default: 'sampled_scene'
List of values: `ResultName` ∈ {'sampled_scene', 'rigid_pose', 'reference_point_x', 'reference_point_y', 'reference_point_z', 'deformed_model', 'deformed_sampled_model'}

▷ **`ResultIndex`** (input_control) ..... integer(-array) ⇝ *integer* / string
Index of the result property.
Default: 0
Suggested values: `ResultIndex` ∈ {0, 1, 2, 3, 'all'}
Restriction: ResultIndex >= 0

▷ **`ResultValue`** (output_control) ..... integer(-array) ⇝ *integer* / string / real / handle
Value of the result property.

Result

If the handle of the result is valid, the operator `get_deformable_surface_matching_result` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`find_deformable_surface_model`, `refine_deformable_surface_model`

Possible Successors

`clear_deformable_surface_model`

See also

`find_deformable_surface_model`, `refine_deformable_surface_model`, `read_deformable_surface_model`, `write_deformable_surface_model`, `clear_deformable_surface_model`

Module

3D Metrology

get_deformable_surface_model_param (: : DeformableSurfaceModel, GenParamName : GenParamValue)

Return the parameters and properties of a deformable surface model.

The operator get_deformable_surface_model_param returns parameters and properties of the surface model DeformableSurfaceModel. The surface model must have been created with, for example, create_deformable_surface_model.

The following values are possible for GenParamName:

'diameter': Diameter of the model point cloud. The diameter is the length of the diagonal of the axis-parallel bounding box.

'sampled_model': The 3D points sampled from the model for matching. This returns a 3D object model that contains all points sampled from the model surface for matching.

'training_models': This returns all 3D object models that were used for the training of the deformable surface model. This includes the 3D object model passed to and sampled by create_deformable_surface_model, and the 3D object models added with add_deformable_surface_model_sample.

'reference_points_x':

'reference_points_y':

'reference_points_z': Returns the x-, y- or z-coordinates of all reference points added with the operator add_deformable_surface_model_reference_point.

Parameters

▷ **DeformableSurfaceModel** (input_control) . . . . . deformable_surface_model ⇝ *handle*
Handle of the deformable surface model.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the parameter.
Default: 'sampled_model'
List of values: GenParamName ∈ {'diameter', 'sampled_model', 'sampled_pose_refinement', 'training_models', 'reference_points_x', 'reference_points_y', 'reference_points_z', 'original_model'}

▷ **GenParamValue** (output_control) . . . . . attribute.value(-array) ⇝ *real* / string / integer
Value of the parameter.

Result

get_deformable_surface_model_param returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_deformable_surface_model, read_deformable_surface_model, add_deformable_surface_model_reference_point

Possible Successors

find_deformable_surface_model, refine_deformable_surface_model, write_deformable_surface_model

See also

create_deformable_surface_model

Module

3D Metrology

read_deformable_surface_model (: : FileName : DeformableSurfaceModel)

Read a deformable surface model from a file.

The operator read_deformable_surface_model reads the deformable surface model, which has been written with write_deformable_surface_model, from the file FileName. The handle of the deformable surface model is returned in DeformableSurfaceModel. If no absolute path is given in FileName, the file is searched in the current directory of the HALCON process. The default HALCON file extension for the deformable surface model file is 'dsfm'. If no file named FileName exists, the default file extension is appended to FileName.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the file to read.
File extension: .dsfm

▷ **DeformableSurfaceModel** (output_control) ..... deformable_surface_model ⇝ *handle*
Handle of the read deformable surface model.

Result

read_deformable_surface_model returns 2 (H_MSG_TRUE) if all parameters are correct and the file can be read. If the file is not a deformable surface model file, the error 9506 is raised. If the file has a version that can not be read by this version of HALCON, the error 9507 is raised. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

write_deformable_surface_model

Possible Successors

find_deformable_surface_model, refine_deformable_surface_model, get_deformable_surface_model_param, clear_deformable_surface_model

Alternatives

create_deformable_surface_model

See also

create_deformable_surface_model, write_deformable_surface_model

Module

3D Metrology

refine_deformable_surface_model (: : DeformableSurfaceModel, ObjectModel3D, RelSamplingDistance, InitialDeformationObjectModel3D, GenParamName, GenParamValue : Score, DeformableSurfaceMatchingResult)

Refine the position and deformation of a deformable surface model in a 3D scene.

The operator refine_deformable_surface_model refines the initial position and deformation given in InitialDeformationObjectModel3D of the surface model DeformableSurfaceModel in the 3D scene ObjectModel3D. The deformable surface model DeformableSurfaceModel must have been created previously with, for example, create_deformable_surface_model.

refine_deformable_surface_model is useful if the position and deformation of an object in a scene is approximately known and only needs to be refined. Additional information about the output parameters can be found in find_deformable_surface_model.

InitialDeformationObjectModel3D must contain as many points as the sampled deformation model obtained by get_deformable_surface_model_param and in the same order.

The score of the refined result is returned in Score and a deformable surface matching result handle is returned in DeformableSurfaceMatchingResult. Details of the result, such as the deformed model and the position of the reference points, can be queried with the operator get_deformable_surface_matching_result using the result handle.

The score is normalized between *0* and *1* and represents the amount of model surface visible in the scene. A value of *1* represents a perfect match.

The parameters GenParamName and GenParamValue are used to set generic parameters. Details about the generic parameters are described in the documentation of find_deformable_surface_model.

Parameters

▷ **DeformableSurfaceModel** (input_control) . . . . . deformable_surface_model ⇝ *handle*
Handle of the deformable surface model.

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the 3D object model containing the scene.

▷ **RelSamplingDistance** (input_control) . . . . . real ⇝ *real*
Relative sampling distance of the scene.
Default: 0.05
Suggested values: RelSamplingDistance ∈ {0.1, 0.07, 0.05, 0.04, 0.03}
Restriction: 0 < RelSamplingDistance < 1

▷ **InitialDeformationObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
Initial deformation of the 3D object model

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'scene_normal_computation', 'pose_ref_num_steps', 'pose_ref_dist_threshold_rel', 'pose_ref_dist_threshold_abs', 'pose_ref_scoring_dist_abs', 'pose_ref_scoring_dist_rel'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {'fast', 'mls', 0, 1, 10, 25, 50, 0.05, 0.1, 0.25, 0.3, 0.05, 0.03, 0.005}

▷ **Score** (output_control) . . . . . real(-array) ⇝ *real*
Score of the refined model.

▷ **DeformableSurfaceMatchingResult** (output_control) . . . . . deformable_surface_matching_result(-array) ⇝ *handle*
Handle of the matching result.

Result

refine_deformable_surface_model returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, get_object_model_3d_params, read_deformable_surface_model, create_deformable_surface_model, get_deformable_surface_model_param, find_deformable_surface_model

Possible Successors

get_deformable_surface_matching_result, clear_deformable_surface_matching_result, clear_object_model_3d

Alternatives

find_deformable_surface_model

See also

create_deformable_surface_model, find_deformable_surface_model

Module

3D Metrology

serialize_deformable_surface_model (: : DeformableSurfaceModel : SerializedItemHandle)

Serialize a deformable surface_model.

serialize_deformable_surface_model serializes the data of a deformable surface model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_deformable_surface_model is converted to a serialized item. The deformable surface model is defined by the handle DeformableSurfaceModel. The serialized deformable surface model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_deformable_surface_model.

Parameters

▷ **DeformableSurfaceModel** (input_control) ..... deformable_surface_model ⇝ *handle*
Handle of the deformable surface model.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_deformable_surface_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_deformable_surface_model, create_deformable_surface_model

Possible Successors

clear_deformable_surface_model, fwrite_serialized_item, send_serialized_item, deserialize_deformable_surface_model

See also

create_deformable_surface_model, read_deformable_surface_model, write_deformable_surface_model, deserialize_deformable_surface_model

Module

3D Metrology

write_deformable_surface_model (: : DeformableSurfaceModel, FileName :)

Write a deformable surface model to a file.

The operator write_deformable_surface_model writes a deformable surface model to the file FileName. The file can be read again with read_deformable_surface_model. The default HALCON file extension for the deformable surface model file is 'dsfm'.

Parameters

▷ **DeformableSurfaceModel** (input_control) . . . . . deformable_surface_model ⇝ *handle*
Handle of the deformable surface model to write.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name to write to.
File extension: .dsfm

Result

write_deformable_surface_model returns 2 (H_MSG_TRUE) if all parameters are correct and the HALCON process has write permission to the file. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_deformable_surface_model, create_deformable_surface_model, get_deformable_surface_model_param

Possible Successors

clear_deformable_surface_model

See also

create_deformable_surface_model, read_deformable_surface_model

Module

3D Metrology

3.5 Shape-Based

clear_shape_model_3d (: : ShapeModel3DID :)

Free the memory of a 3D shape model.

The operator clear_shape_model_3d frees the memory of a 3D shape model that was created by create_shape_model_3d. After calling clear_shape_model_3d, the model can no longer be used. The handle ShapeModel3DID becomes invalid.

Parameters

▷ **ShapeModel3DID** (input_control) . . . . . shape_model_3d(-array) ⇝ *handle*
Handle of the 3D shape model.

Result

If the handle of the model is valid, the operator clear_shape_model_3d returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ShapeModel3DID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_shape_model_3d, read_shape_model_3d, write_shape_model_3d

Module

3D Metrology

create_cam_pose_look_at_point (: : CamPosX, CamPosY, CamPosZ, LookAtX, LookAtY, LookAtZ, RefPlaneNormal, CamRoll : CamPose)

Create a 3D camera pose from camera center and viewing direction.

The operator create_cam_pose_look_at_point creates a 3D camera pose with respect to a world coordinate system based on two points and the camera roll angle.

The first of the two points defines the position of the optical center of the camera in the world coordinate system, i.e., the origin of the camera coordinate system. It is given by its three coordinates CamPosX, CamPosY, and CamPosZ. The second of the two points defines the viewing direction of the camera. It represents the point in the world coordinate system at which the camera is to look. It is also specified by its three coordinates LookAtX, LookAtY, and LookAtZ. Consequently, the second point lies on the z axis of the camera coordinate system.

Finally, the remaining degree of freedom to be specified is a rotation of the camera around its z axis, i.e., the roll angle of the camera. To determine this rotation, the normal of a reference plane can be specified in RefPlaneNormal, which defines the reference orientation of the camera. Finally, the camera roll angle can be specified in CamRoll, which describes a rotation of the camera around its z axis with respect to its reference orientation.

The reference plane can be seen as a plane in the world coordinate system that is parallel to the x axis of the camera (in its reference orientation, i.e., with a roll angle of 0). In an alternative interpretation, the normal vector of the reference plane projected onto the image plane points upwards, i.e., it is mapped to the negative y axis of the camera coordinate system. The parameter RefPlaneNormal may take one of the following values:

'x': The reference plane is the yz plane of the world coordinate system. The projected x axis of the world coordinate system points *upwards* in the image plane.

'-x': The reference plane is the yz plane of the world coordinate system. The projected x axis of the world coordinate system points *downwards* in the image plane.

'y': The reference plane is the xz plane of the world coordinate system. The projected y axis of the world coordinate system points *upwards* in the image plane.

'-y': The reference plane is the xz plane of the world coordinate system. The projected y axis of the world coordinate system points *downwards* in the image plane.

'z': The reference plane is the xy plane of the world coordinate system. The projected z axis of the world coordinate system points *upwards* in the image plane.

'-z': The reference plane is the xy plane of the world coordinate system. The projected z axis of the world coordinate system points *downwards* in the image plane.

Alternatively to the above values, an arbitrary normal vector can be specified in RefPlaneNormal, which is not restricted to the coordinate axes. For this, a tuple of three values representing the three components of the normal vector must be passed.

Note that the position of the optical center and the point at which the camera looks must differ from each other. Furthermore, the normal vector of the reference plane and the z axis of the camera must not be parallel. Otherwise, the camera pose is not well-defined.

create_cam_pose_look_at_point is particularly useful if a 3D object model or a 3D shape model should be visualized from a certain camera position. In this case, the pose that is created by create_cam_pose_look_at_point can be passed to project_object_model_3d or project_shape_model_3d, respectively.

It is also possible to pass tuples of different length for different input parameters. In this case, internally the maximum number of parameter values over all input control parameters is computed. This number is taken as

the number of output camera poses. Then, all input parameters can contain a single value or the same number of values as output camera poses. In the first case, the single value is used for the computation of all camera poses, while in the second case the respective value of the element in the parameter is used for the computation of the corresponding camera pose.

Parameters

- ▷ **CamPosX** (input_control) . . . . . real(-array) ⇝ *real*
  X coordinate of the optical center of the camera.
- ▷ **CamPosY** (input_control) . . . . . real(-array) ⇝ *real*
  Y coordinate of the optical center of the camera.
- ▷ **CamPosZ** (input_control) . . . . . real(-array) ⇝ *real*
  Z coordinate of the optical center of the camera.
- ▷ **LookAtX** (input_control) . . . . . real(-array) ⇝ *real*
  X coordinate of the 3D point to which the camera is directed.
- ▷ **LookAtY** (input_control) . . . . . real(-array) ⇝ *real*
  Y coordinate of the 3D point to which the camera is directed.
- ▷ **LookAtZ** (input_control) . . . . . real(-array) ⇝ *real*
  Z coordinate of the 3D point to which the camera is directed.
- ▷ **RefPlaneNormal** (input_control) . . . . . string-array ⇝ *string* / real
  Normal vector of the reference plane (points up).
  Default: '-y'
  List of values: RefPlaneNormal ∈ {'x', 'y', 'z', '-x', '-y', '-z'}
- ▷ **CamRoll** (input_control) . . . . . angle.rad(-array) ⇝ *real*
  Camera roll angle.
  Default: 0
- ▷ **CamPose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
  3D camera pose.

Result

If the parameters are valid, the operator create_cam_pose_look_at_point returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameters are chosen such that the pose is not well defined, the error 8940 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

convert_point_3d_spher_to_cart

Alternatives

create_pose

Module

3D Metrology

create_shape_model_3d (: : ObjectModel3D, CamParam, RefRotX, RefRotY, RefRotZ, OrderOfRotation, LongitudeMin, LongitudeMax, LatitudeMin, LatitudeMax, CamRollMin, CamRollMax, DistMin, DistMax, MinContrast, GenParamName, GenParamValue : ShapeModel3DID)

Prepare a 3D object model for matching.

The operator create_shape_model_3d prepares a 3D object model, which is passed in ObjectModel3D, as a 3D shape model used for matching. The 3D object model must previously been read from a file by using read_object_model_3d.

The 3D shape model is generated by computing different views of the 3D object model within a user-specified pose range. The views are automatically generated by placing virtual cameras around the 3D object model and projecting the 3D object model into the image plane of each virtual camera position. For each such obtained view a 2D shape representation is computed. Thus, for the generation of the 3D shape model, no images of the object are used but only the 3D object model, which is passed in `ObjectModel3D`. The shape representations of all views are stored in the 3D shape model, which is returned in `ShapeModel3DID`. During the matching process with `find_shape_model_3d`, the shape representations are used to find out the best-matching view, from which the pose is subsequently refined and returned.

In order to create the model views correctly, the camera parameters of the camera that will be used for the matching must be passed in `CamParam`. The camera parameters are necessary, for example, to determine the scale of the projections by using the actual focal length of the camera. Furthermore, they are used to treat radial distortions of the lens correctly. Consequently, it is essential to calibrate the camera by using `calibrate_cameras` before creating the 3D shape model. On the one hand, this is necessary to obtain accurate poses from `find_shape_model_3d`. On the other hand, this makes the 3D matching applicable even when using lenses with significant radial distortions.

The pose range within which the model views are generated can be specified by the parameters `RefRotX`, `RefRotY`, `RefRotZ`, `OrderOfRotation`, `LongitudeMin`, `LongitudeMax`, `LatitudeMin`, `LatitudeMax`, `CamRollMin`, `CamRollMax`, `DistMin`, and `DistMax`. Note that the model will only be recognized during the matching if it appears within the specified pose range. The parameters are described in the following:

Before computing the views, the origin of the coordinate system of the 3D object model is moved to the reference point of the 3D object model, which is the center of the smallest enclosing axis-parallel cuboid and can be queried by using `get_object_model_3d_params`. The virtual cameras, which are used to create the views, are arranged around the 3D object model in such a way that they all look at the origin of the coordinate system, i.e., the z axes of the cameras pass through the origin. The pose range can then be specified by restricting the views to a certain quadrilateral on the sphere around the origin. This naturally leads to the use of the spherical coordinates longitude, latitude, and radius. The definition of the spherical coordinate system is chosen such that the equatorial plane corresponds to the xz plane of the Cartesian coordinate system with the y axis pointing to the south pole (negative latitude) and the negative z axis pointing in the direction of the zero meridian (see `convert_point_3d_spher_to_cart` or `convert_point_3d_cart_to_spher` for further details about the conversion between Cartesian and spherical coordinates). The advantage of this definition is that a camera with the pose [0,0,z,0,0,0,0] has its optical center at longitude=0, latitude=0, and radius=z. In this case, the radius represents the distance of the optical center of the camera to the reference point of the 3D object model.

The longitude range, for which views are to be generated, can be specified by `LongitudeMin` and `LongitudeMax`, both given in radians. Accordingly, the latitude range can be specified by `LatitudeMin` and `LatitudeMax`, also given in radians. `LongitudeMin` and `LongitudeMax` are adjusted to maintain a range of 360° (2π). If an adjustment is possible, `LongitudeMin` and the range are preserved. The minimum and maximum distance between the camera center and the model reference point is specified by `DistMin` and `DistMax`. Thereby, the model origin is in the center of the smallest enclosing cuboid and does not necessarily coincide with the origin of the CAD coordinate system. Note that the unit of the distance must be meters (assuming that the parameter `Scale` has been correctly set when reading the CAD file with `read_object_model_3d`). Finally, the minimum and the maximum camera roll angle can be specified in `CamRollMin` and `CamRollMax`. This interval specifies the allowable camera rotation around its z axis with respect to the 3D object model. If the image plane is parallel to the plane on which the objects reside and if it is known that the object may rotate in this plane only in a restricted range, then it is reasonable to specify this range in `CamRollMin` and `CamRollMax`. In all other cases the interpretation of the camera roll angle is difficult, and hence, it is recommended to set this interval to $[-\pi, +\pi]$. Note that the larger the specified pose range is chosen the more memory the model will consume (except from the range of the camera roll angle) and the slower the matching will be.

The orientation of the coordinate system of the 3D object model is defined by the coordinates within the CAD file that was read by using `read_object_model_3d`. Therefore, it is reasonable to previously rotate the 3D object model into a reference orientation such that the view that corresponds to longitude=0 and latitude=0 is approximately at the center of the pose range. This can be achieved by passing appropriate values for the reference orientation in `RefRotX`, `RefRotY`, `RefRotZ`, and `OrderOfRotation`. The rotation is performed around the axes of the 3D object model, which origin was set to the reference point. The longitude and latitude range can then be interpreted as a variation of the 3D object model pose around the reference orientation. There are two possible ways to specify the reference orientation. The first possibility is to specify three rotation angles in `RefRotX`, `RefRotY`, and `RefRotZ` and the order in which the three rotations are to be applied in `OrderOfRotation`, which can either be *'gba'* or *'abg'*. The second possibility is to specify the three components of the Rodriguez

rotation vector in RefRotX, RefRotY, and RefRotZ. In this case, OrderOfRotation must be set to *'rodriguez'* (see create_pose for detailed information about the order of the rotations and the definition of the Rodriguez vector).

Thus, two transformations are applied to the 3D object model before computing the model views within the pose range. The first transformation is the translation of the origin of the coordinate systems to the reference point. The second transformation is the rotation of the 3D object model to the desired reference orientation around the axes of the reference coordinate system. By combining both transformations one obtains the reference pose of the 3D shape model. The reference pose of the 3D shape model thus describes the pose of the reference coordinate system with respect to the coordinate system of the 3D object model defined by the CAD file. Let $t = (x, y, z)'$ be the coordinates of the reference point of the 3D object model and R be the rotation matrix containing the reference orientation. Then, a point p_m given in the 3D object model coordinate system can be transformed to a point p_r in the reference coordinate system of the 3D shape model by applying the following formula:

$$p_r = R \cdot (p_m - t)$$

This transformation can be expressed by a homogeneous 3D transformation matrix or alternatively in terms of a 3D pose. The latter can be queried by passing *'reference_pose'* for the parameter GenParamName of the operator get_shape_model_3d_params. The above formula can be best imagined as a pose of pose type 8, 10, or 12, depending on the value that was chosen for OrderOfRotation (see create_pose for detailed information about the different pose types). Note, however, that get_shape_model_3d_params always returns the pose using the pose type 0. Finally, poses that are given in one of the two coordinate systems can be transformed to the other coordinate system by using trans_pose_shape_model_3d.

Furthermore, it should be noted that the reference coordinate system is introduced only to specify the pose range in a convenient way. The pose resulting from the 3D matching that is performed with find_shape_model_3d always refers to the original 3D object model coordinate system used in the CAD file.

With MinContrast, it can be determined which edge contrast the model must at least have in the recognition performed by find_shape_model_3d. In other words, this parameter separates the model from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, MinContrast should be set to 10. If multichannel images are used for the search images, the noise in one channel must be multiplied by the square root of the number of channels to determine MinContrast. If, for example, the gray values fluctuate within a range of 10 gray levels in a single channel and the image is a three-channel image, MinContrast should be set to 17. If the model should be recognized in very low contrast images, MinContrast must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, MinContrast should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the pose of the model is extracted robustly and accurately by find_shape_model_3d.

The parameters described above are application-dependent and must be always specified when creating a 3D shape model. In addition, there are some generic parameters that can optionally be used to influence the model creation. For most applications these parameters need not to be specified but can be left at their default values. If desired, these parameters and their corresponding values can be specified by using GenParamName and GenParamValue, respectively. The following values for GenParamName are possible:

'num_levels': For efficiency reasons the model views are generated on multiple pyramid levels. On higher levels fewer views are generated than on lower levels. With the parameter *'num_levels'* the number of pyramid levels on which model views are generated can be specified. It should be chosen as large as possible because by this the time necessary to find the model is significantly reduced. On the other hand, the number of levels must be chosen such that the shape representations of the views on the highest pyramid level are still recognizable and contain a sufficient number of points (at least four). If not enough model points are generated for a certain view, the view is deleted from the model and replaced by a view on a lower pyramid level. If for all views on a pyramid level not enough model points are generated, the number of levels is reduced internally until for at least one view enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is too small for all views already on the lowest pyramid level, create_shape_model_3d returns an error message. If *'num_levels'* is set to *'auto'* (default value), create_shape_model_3d determines the number of pyramid levels automatically. In this case all model views on all pyramid levels are automatically checked whether their shape representations are still recognizable. If the shape representation of a certain view is found to be not recognizable, the view is deleted from the model and replaced by a view on a lower pyramid level. Note that if *'num_levels'* is set to *'auto'*, the number of pyramid levels can be different for different views. In rare cases, it might happen that create_shape_model_3d determines a value for the number of pyramid levels that

is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for `MinScore` or `Greediness` in `find_shape_model_3d` in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in `find_shape_model_3d` may increase. In these cases, the views on the pyramid levels should be checked by using the output of `get_shape_model_3d_contours`.

Suggested values: *'auto'*, *3*, *4*, *5*, *6*

Default: *'auto'*

'fast_pose_refinement': The parameter specifies whether the pose refinement during the search with `find_shape_model_3d` is sped up. If *'fast_pose_refinement'* is set to *'false'*, for complex models with a large number of faces the pose refinement step might amount to a significant part of the overall computation time. If *'fast_pose_refinement'* is set to *'true'*, some of the calculations that are necessary during the pose refinement are already performed during the model generation and stored in the model. Consequently, the pose refinement during the search will be faster. Please note, however, that in this case the memory consumption of the model may increase significantly (typically by less than 30 percent). Further note that the resulting poses that are returned by `find_shape_model_3d` might slightly differ depending on the value of *'fast_pose_refinement'*, because internally the pose refinement is approximated if the parameter is set to *'true'*.

List of values: *'true'*, *'false'*

Default: *'true'*

'lowest_model_level': In some cases the model generation process might be very time consuming and the memory consumption of the model might be very high. The reason for this is that in these cases the number of views, which must be computed and stored in the model, is very high. The larger the pose range is chosen and the larger the objects appear in the image (measured in pixels) the more views are necessary. Consequently, especially the use of large images (e.g., images exceeding a size of 640×480) can result in very large models. Because the number of views is highest on lower pyramid levels, the parameter *'lowest_model_level'* can be used to exclude the lower pyramid levels from the generation of views. The value that is passed for *'lowest_model_level'* determines the lowest pyramid level down to which views are generated and stored in the 3d shape model. If, for example, a value of *2* is passed for large models, the time to generate the model as well as the size of the resulting model is reduced to approximately one third of the original values. If *'lowest_model_level'* is not passed, views are generated for all pyramid levels, which corresponds to the behavior when passing a value of *1* for *'lowest_model_level'*. If for *'lowest_model_level'* a value larger than *1* is passed, in `find_shape_model_3d` the tracking of matches through the pyramid will be stopped at this level. However, if in `find_shape_model_3d` a least-squares adjustment is chosen for pose refinement, the matches are refined on the lowest pyramid level using the least-squares adjustment. Note that for different values for *'lowest_model_level'* different matches might be found during the search. Furthermore, the score of the matches depends on the chosen method for pose refinement. Also note that the higher *'lowest_model_level'* is chosen the higher the portion of the refinement step with respect to the overall run-time of `find_shape_model_3d` will be. As a consequence for higher values of *'lowest_model_level'* the influence of the generic parameter *'fast_pose_refinement'* (see above) on the runtime will increase. A large value for *'lowest_model_level'* on the one hand may lead to long computation times of `find_shape_model_3d` if *'fast_pose_refinement'* is switches off (*'false'*). On the other hand it may lead to a decreased accuracy if *'fast_pose_refinement'* is switches on (*'true'*) because in this mode the pose refinement is only approximated. Therefore, the value for *'lowest_model_level'* should be chosen as small as possible. Furthermore, *'lowest_model_level'* should be chosen small enough such that the edges of the 3D object model are still observable on this level.

Suggested values: *1*, *2*, *3*

Default: *1*

'optimization': For models with particularly large model views, it may be useful to reduce the number of model points by setting *'optimization'* to a value different from *'none'*. If *'optimization'* = *'none'*, all model points are stored. In all other cases, the number of points is reduced according to the value of *'optimization'*. If the number of points is reduced, it may be necessary in `find_shape_model_3d` to set the parameter `Greediness` to a smaller value, e.g., 0.7 or 0.8. For models with small model views, the reduction of the number of model points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined. If *'optimization'* is set to *'auto'*, `create_shape_model_3d` automatically determines the reduction of the number of model points for each model view.

List of values: *'auto'*, *'none'*, *'point_reduction_low'*, *'point_reduction_medium'*, *'point_reduction_high'*

Default: *'auto'*

'metric': This parameter determines the conditions under which the model is recognized in the image. If *'metric'* = *'ignore_part_polarity'*, the contrast polarity is allowed to change only between different parts of the model, whereas the polarity of model points that are within the same model part must not change. Please note that the term *'ignore_part_polarity'* is capable of being misunderstood. It means that polarity changes between neighboring model parts do not influence the score, and hence are ignored. Appropriate model parts are automatically determined. The size of the parts can be controlled by the generic parameter *'part_size'*, which is described below. Note that this metric only works for one-channel images. Consequently, if the model is created by using this metric and searched in a multi-channel image by using `find_shape_model_3d` an error will be returned. If *'metric'* = *'ignore_local_polarity'*, the model is found even if the contrast polarity changes for each individual model point. This metric works for one-channel images as well as for multi-channel images. The metric *'ignore_part_polarity'* should be used if the images contain strongly textured backgrounds or clutter objects, which might result in wrong matches. Note that in general the scores of the matches that are returned by `find_shape_model_3d` are lower for *'ignore_part_polarity'* than for *'ignore_local_polarity'*. This should be kept in mind when choosing the right value for the parameter `MinScore` of `find_shape_model_3d`.

List of values: *'ignore_local_polarity'*, *'ignore_part_polarity'*

Default: *'ignore_local_polarity'*

'part_size': This parameter determines the size of the model parts that is used when *'metric'* is set to *'ignore_part_polarity'* (see above). The size must be specified in pixels and should be approximately twice as large as the size of the background texture in the image. For example, if an object should be found in front of a chessboard with black and white squares of size 5×5 pixels, *'part_size'* should be set to *10*. Note that higher values of *'part_size'* might also decrease the scores of correct instances especially when searching for objects with shiny or reflective surfaces. Therefore, the risk of missing correct instances might increase if *'part_size'* is set to a higher value. If *'metric'* is set to *'ignore_local_polarity'*, the value of *'part_size'* is ignored.

Suggested values: *2*, *3*, *4*, *6*, *8*, *10*

Default: *4*

'min_face_angle': 3D edges are only included in the shape representations of the views if the angle between the two 3D faces that are incident with the 3D object model edge is at least *'min_face_angle'*. If *'min_face_angle'* is set to *0.0*, all edges are included. If *'min_face_angle'* is set to π (equivalent to 180 degrees), only the silhouette of the 3D object model is included. This parameter can be used to suppress edges within curved surfaces, e.g., the surface of a cylinder or cone. Curved surfaces are approximated by multiple planar faces. The edges between such neighboring planar faces should not be included in the shape representation because they also do not appear in real images of the model. Thus, *'min_face_angle'* should be set sufficiently high to suppress these edges. The effect of different values for *'min_face_angle'* can be inspected by using `project_object_model_3d` before calling `create_shape_model_3d`. Note that if edges that are not visible in the search image are included in the shape representation, the performance (robustness and speed) of the matching may decrease considerably.

Suggested values: *'rad(10)'*, *'rad(20)'*, *'rad(30)'*, *'rad(45)'*

Default: *'rad(30)'*

'min_size': This value determines a threshold for the selection of significant model components based on the size of the components, i.e., connected components that have fewer points than the specified minimum size are suppressed. This threshold for the minimum size is divided by two for each successive pyramid level.

Suggested values: *'auto'*, *0*, *3*, *5*, *10*, *20*

Default: *'auto'*

'model_tolerance': The parameter specifies the tolerance of the projected 3D object model edges in the image, given in pixels. The higher the value is chosen, the fewer views need to be generated. Consequently, a higher value results in models that are less memory consuming and faster to find with `find_shape_model_3d`. On the other hand, if the value is chosen too high, the robustness of the matching will decrease. Therefore, this parameter should only be modified with care. For most applications, a good compromise between speed and robustness is obtained when setting *'model_tolerance'* to *1*.

Suggested values: *0*, *1*, *2*

Default: *1*

'union_adjacent_contours': This parameter specifies if adjacent projected contours should be joined by the operator `project_shape_model_3d` or not. Activating this option is equivalent to calling `union_adjacent_contours_xld` afterwards, but significantly faster.

List of values: *'true'*, *'false'*
Default: *'false'*

If the system variable (see `set_system`) *'opengl_hidden_surface_removal_enable'* is set to *'true'* (which is default if it is available) the graphics card is used to accelerate the computation of the visible faces in the model views. Depending on the graphics card this is significantly faster than the analytic visibility computation. If *'fast_pose_refinement'* is set to *'true'*, the precomputations necessary for the pose refinement step in `find_shape_model_3d` are also performed on the graphics card. Be aware that the results of the OpenGL projection are slightly different compared to the analytic projection.

Parameters

- ▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d ⇝ *handle*
  Handle of the 3D object model.
- ▷ **`CamParam`** (input_control) . . . . . campar ⇝ *real* / integer / string
  Internal camera parameters.
- ▷ **`RefRotX`** (input_control) . . . . . angle.rad ⇝ *real*
  Reference orientation: Rotation around x-axis or x component of the Rodriguez vector (in radians or without unit).
  Default: 0
  Suggested values: `RefRotX` ∈ {-1.57, -0.78, -0.17, 0., 0.17, 0.78, 1.57}
- ▷ **`RefRotY`** (input_control) . . . . . angle.rad ⇝ *real*
  Reference orientation: Rotation around y-axis or y component of the Rodriguez vector (in radians or without unit).
  Default: 0
  Suggested values: `RefRotY` ∈ {-1.57, -0.78, -0.17, 0., 0.17, 0.78, 1.57}
- ▷ **`RefRotZ`** (input_control) . . . . . angle.rad ⇝ *real*
  Reference orientation: Rotation around z-axis or z component of the Rodriguez vector (in radians or without unit).
  Default: 0
  Suggested values: `RefRotZ` ∈ {-1.57, -0.78, -0.17, 0., 0.17, 0.78, 1.57}
- ▷ **`OrderOfRotation`** (input_control) . . . . . string ⇝ *string*
  Meaning of the rotation values of the reference orientation.
  Default: 'gba'
  List of values: `OrderOfRotation` ∈ {'gba', 'abg', 'rodriguez'}
- ▷ **`LongitudeMin`** (input_control) . . . . . angle.rad ⇝ *real*
  Minimum longitude of the model views.
  Default: -0.35
  Suggested values: `LongitudeMin` ∈ {-0.78, -0.35, -0.17}
- ▷ **`LongitudeMax`** (input_control) . . . . . angle.rad ⇝ *real*
  Maximum longitude of the model views.
  Default: 0.35
  Suggested values: `LongitudeMax` ∈ {0.17, 0.35, 0.78}
  Restriction: LongitudeMax >= LongitudeMin
- ▷ **`LatitudeMin`** (input_control) . . . . . angle.rad ⇝ *real*
  Minimum latitude of the model views.
  Default: -0.35
  Suggested values: `LatitudeMin` ∈ {-0.78, -0.35, -0.17}
  Restriction: - pi / 2 <= LatitudeMin && LatitudeMin <= pi / 2
- ▷ **`LatitudeMax`** (input_control) . . . . . angle.rad ⇝ *real*
  Maximum latitude of the model views.
  Default: 0.35
  Suggested values: `LatitudeMax` ∈ {0.17, 0.35, 0.78}
  Restriction: - pi / 2 <= LatitudeMax && LatitudeMax <= pi / 2 && LatitudeMax >= LatitudeMin
- ▷ **`CamRollMin`** (input_control) . . . . . angle.rad ⇝ *real*
  Minimum camera roll angle of the model views.
  Default: -3.1416
  Suggested values: `CamRollMin` ∈ {-3.14, -1.57, -0.39, 0.0, 0.39, 1.57, 3.14}

▷ **CamRollMax** (input_control) . . . . . angle.rad ⇝ *real*
Maximum camera roll angle of the model views.
Default: 3.1416
Suggested values: CamRollMax ∈ {-3.14, -1.57, -0.39, 0.0, 0.39, 1.57, 3.14}
Restriction: CamRollMax >= CamRollMin

▷ **DistMin** (input_control) . . . . . number ⇝ *real*
Minimum camera-object-distance of the model views.
Default: 0.3
Suggested values: DistMin ∈ {0.05, 0.1, 0.2, 0.5}
Restriction: DistMin > 0

▷ **DistMax** (input_control) . . . . . number ⇝ *real*
Maximum camera-object-distance of the model views.
Default: 0.4
Suggested values: DistMax ∈ {0.1, 0.2, 0.5, 1.0}
Restriction: DistMax >= DistMin

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer*
Minimum contrast of the objects in the search images.
Default: 10
Suggested values: MinContrast ∈ {1, 2, 3, 5, 7, 10, 20, 30, 1000, 2000, 5000}

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of (optional) parameters for controlling the behavior of the operator.
Default: []
List of values: GenParamName ∈ {'num_levels', 'fast_pose_refinement', 'lowest_model_level', 'optimization', 'metric', 'part_size', 'min_face_angle', 'min_size', 'model_tolerance', 'union_adjacent_contours'}

▷ **GenParamValue** (input_control) . . . . . attribute.name(-array) ⇝ *integer* / real / string
Values of the optional generic parameters.
Default: []
Suggested values: GenParamValue ∈ {0, 1, 2, 3, 4, 6, 8, 10, 'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high', 0.1, 0.2, 0.3, 'ignore_local_polarity', 'ignore_part_polarity', 'true', 'false'}

▷ **ShapeModel3DID** (output_control) . . . . . shape_model_3d ⇝ *handle*
Handle of the 3D shape model.

Result

If the parameters are valid, the operator create_shape_model_3d returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameters are chosen such that all model views contain too few points, the error 8510 is raised. In the case that the projected model is bigger than twice the image size in at least one model view, the error 8910 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_object_model_3d, project_object_model_3d, get_object_model_3d_params

Possible Successors

find_shape_model_3d, write_shape_model_3d, project_shape_model_3d, get_shape_model_3d_params, get_shape_model_3d_contours

See also

convert_point_3d_cart_to_spher, convert_point_3d_spher_to_cart, create_cam_pose_look_at_point, trans_pose_shape_model_3d

References

Markus Ulrich, Christian Wiedemann, Carsten Steger, "Combining Scale-Space and Similarity-Based Aspect

Graphs for Fast 3D Object Recognition," IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 1902-1914, Oct., 2012.

Module

3D Metrology

deserialize_shape_model_3d (: : SerializedItemHandle : ShapeModel3DID)

Deserialize a serialized 3D shape model.

deserialize_shape_model_3d deserializes a 3D shape model, that was serialized by serialize_shape_model_3d (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized 3D shape model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created 3D shape model with the handle ShapeModel3DID.

Parameters

- ▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
  Handle of the serialized item.
- ▷ **ShapeModel3DID** (output_control) ..... shape_model_3d ⇝ *handle*
  Handle of the 3D shape model.

Result

If the parameters are valid, the operator deserialize_shape_model_3d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_shape_model_3d

Possible Successors

find_shape_model_3d, get_shape_model_3d_params

See also

create_shape_model_3d, clear_shape_model_3d

Module

3D Metrology

find_shape_model_3d (Image : : ShapeModel3DID, MinScore, Greediness, NumLevels, GenParamName, GenParamValue : Pose, CovPose, Score)

Find the best matches of a 3D shape model in an image.

The operator find_shape_model_3d finds the best matches of the 3D shape model ShapeModel3DID in the input Image. The 3D shape model must have been created previously by calling create_shape_model_3d or read_shape_model_3d.

The 3D pose of the found instances of the model is returned in Pose. The pose is in the form ${}^{ccs}\mathbf{P}_{mcs}$, where *ccs* denotes the camera coordinate system and *mcs* the model coordinate system (which is a 3D world coordinate system), see Transformations / Poses and "Solution Guide III-C - 3D Vision". Hence, it describes the pose of the 3D object model in camera coordinates. It should be noted that the resulting Pose does not refer to reference coordinate system that is introduced in create_shape_model_3d but to the original 3D object model coordinate system used in the CAD file. If a pose refinement was applied (see below), additionally the accuracy of the six pose parameters are returned in CovPose. By default, CovPose contains the 6 standard

deviations of the pose parameters for each match. In contrast, if the generic parameter *'cov_pose_mode'* (see below) was set to *'covariances'*, CovPose contains the 36 values of the complete 6×6 covariance matrix of the 6 pose parameters. Note that this reflects only an inner accuracy from which the real accuracy of the pose may differ. Finally, the score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

Input parameters in detail

Image and its domain: The domain of the image Image determines the search space for the reference point of the 3D object model. There is no need to correct any distortions in Image as the calibration data has already been provided during the model creation.

MinScore: The parameter MinScore determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger MinScore is chosen, the faster the search is. If the model can be expected never to be occluded in the images, MinScore may be set as high as 0.8 or even 0.9. Note that in images with a high degree of clutter or strong background texture, MinScore should be set to a value not much lower than *0.7* since otherwise false matches could be found.

Greediness: The parameter Greediness determines how "greedily" the search should be carried out. If Greediness $= 0$, a safe search heuristic is used, which always finds the model if it is visible in the image. However, the search will be relatively time consuming in this case. If Greediness $= 1$, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For Greediness $= 1$, the maximum search speed is achieved. In almost all cases, the 3D shape model will always be found for Greediness $= 0.9$.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the 3D shape model was created with create_shape_model_3d. If NumLevels is set to *0*, the number of pyramid levels specified in create_shape_model_3d is used. Optionally, NumLevels can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of *[4,2]* for NumLevels means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

GenParamName and GenParamValue: In addition to the parameters described above, there are some generic parameters that can optionally be used to influence the matching. For most applications these parameters need not to be specified but can be left at their default values. If desired, these parameters and their corresponding values can be specified by using GenParamName and GenParamValue, respectively. The following values for GenParamName are possible:

- If the pose range in which the model is to be searched is smaller than the pose range that was specified during the model creation with create_shape_model_3d, the pose range can be restricted appropriately with the following parameters. If the values lie outside the pose range of the model, the values are automatically clipped to the pose range of the model.

  'longitude_min': Sets the minimum longitude of the pose range.
  Suggested values: *'rad(-45)'*, *'rad(-30)'*, *'rad(-15)'*
  Default: *'rad(-180)'*

  'longitude_max': Sets the maximum longitude of the pose range.
  Suggested values: *'rad(15)'*, *'rad(30)'*, *'rad(45)'*
  Default: *'rad(180)'*

  'latitude_min': Sets the minimum latitude of the pose range.
  Suggested values: *'rad(-45)'*, *'rad(-30)'*, *'rad(-15)'*
  Default: *'rad(-90)'*

  'latitude_max': Sets the maximum latitude of the pose range.
  Suggested values: *'rad(15)'*, *'rad(30)'*, *'rad(45)'*
  Default: *'rad(90)'*

  'cam_roll_min': Sets the minimum camera roll angle of the pose range.
  Suggested values: *'rad(-45)'*, *'rad(-30)'*, *'rad(-15)'*
  Default: *'rad(-180)'*

'cam_roll_max': Sets the maximum camera roll angle of the pose range.
Suggested values: *'rad(15)'*, *'rad(30)'*, *'rad(45)'*
Default: *'rad(180)'*

'dist_min': Sets the minimum camera-object-distance of the pose range.
Suggested values: *0.05*, *0.1*, *0.5*, *1.0*
Default: *0*

'dist_max': Sets the maximum camera-object-distance of the pose range.
Suggested values: *0.05*, *0.1*, *0.5*, *1.0*
Default: (∞)

- Further generic parameters that do not concern the pose range can be specified:

'num_matches': With this parameter the maximum number of instances to be found can be determined. If more than the specified number of instances with a score greater than `MinScore` are found in the image, only the best *'num_matches'* instances are returned. If fewer than *'num_matches'* are found, only that number is returned, i.e., the parameter `MinScore` takes precedence over *'num_matches'*. If *'num_matches'* is set to *0*, all matches that satisfy the score criterion are returned. Note that the more matches should be found the slower the matching will be.
Suggested values: *0*, *1*, *2*, *3*
Default: *1*

'max_overlap': It may happen that multiple instances with similar positions but with different orientations are found in the image. The parameter *'max_overlap'* determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than the specified value only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see `smallest_rectangle2`) of the found instances. If in `create_shape_model_3d` for *'lowest_model_level'* a value larger than *1* was passed, the overlap calculation is based on the projection of the smallest enclosing axis-parallel cuboid of the 3D object model. Because in this case the overlap might be overestimated, in some cases it could be necessary to increase the value for *'max_overlap'*. If $'max\_overlap' = 0$, the found instances may not overlap at all, while for $'max\_overlap' = 1$ all instances are returned.
Suggested values: *0.0*, *0.2*, *0.4*, *0.6*, *0.8*, *1.0*
Default: *0.5*

'pose_refinement': This parameter determines whether the poses of the instances should be refined after the matching. If *'pose_refinement'* is set to *'none'* the model's pose is only determined with a limited accuracy. In this case, the accuracy depends on several sampling steps that are used inside the matching process and, therefore cannot be predicted very well. Therefore, *'pose_refinement'* should only be set to *'none'* when the computation time is of primary concern and an approximate pose is sufficient. In all other cases the pose should be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In order to achieve a high accuracy, this refinement is directly performed in 3D. Therefore, the refinement requires additional computation time. If the system variable (see `set_system`) *'opengl_hidden_surface_removal_enable'* is set to *'true'* (which is default if it is available) and the model `ShapeModel3DID` was created with *'fast_pose_refinement'* set to *'false'*, the projection of the model in the pose refinement step is accelerated using the graphics card. Depending on the graphics card this is significantly faster than the non accelerated algorithm. Be aware that the results of the OpenGL projection are slightly different compared to the analytic projection. The different modes for least-squares adjustment (*'least_squares'*, *'least_squares_high'*, and *'least_squares_very_high'*) can be used to determine the accuracy with which the minimum distance is searched for. The higher the accuracy is chosen, the longer the pose refinement will take, however. For most applications *'least_squares_high'* should be chosen because this results in the best trade-off between runtime and accuracy. Note that the pose refinement can be sped up by passing *'fast_pose_refinement'* for the parameter `GenParamName` of the operator `create_shape_model_3d`.
List of values: *'none'*, *'least_squares'*, *'least_squares_high'*, *'least_squares_very_high'*
Default: *'least_squares_high'*

'recompute_score': This parameter determines whether the score of the matches is recomputed after the pose refinement. If *'recompute_score'* is set to *'false'*, the score is returned that was computed before the pose refinement. In some cases, however, the pose refinement changes the object pose by more than one pixel in the image. Consequently, the original score does not appropriately describe the refined match any longer. This could result in wrong matches obtaining high scores or perfect

matches obtaining low scores. To obtain a more meaningful score that reflects the pose changes due to the pose refinement, the score can be recomputed after the pose refinement by setting *'recompute_score'* to *'true'*. Note that this might change the order of the matches as well as the selection of matches that is returned. Also note that the recomputation of the score values needs additional computation time. This increase of the run-time can be reduced by setting the generic parameter *'fast_pose_refinement'* of the operator `create_shape_model_3d` to *'true'*.
List of values: *'false'*, *'true'*
Default: *'false'*

'outlier_suppression': This parameter only takes effect if *'pose_refinement'* is set to a value other than *'none'*, and hence, a least-squares adjustment is performed. Then, in some cases it might be useful to apply a robust outlier suppression during the least-squares adjustment. This might be necessary, for example, if a high degree of clutter is present in the image, which prevents the least-squares adjustment from finding the optimum pose. In this case, *'outlier_suppression'* should be set to either *'medium'* (eliminates a medium proportion of outliers) or *'high'* (eliminates a high proportion of outliers). However, in most applications, no robust outlier suppression is necessary, and hence, *'outlier_suppression'* can be set to *'none'*. It should be noted that activating the outlier suppression comes along with a significantly increasing computation time.
List of values: *'none'*, *'medium'*, *'high'*
Default: *'none'*

'cov_pose_mode': This parameter only takes effect if *'pose_refinement'* is set to a value other than *'none'*, and hence, a least-squares adjustment is performed. *'cov_pose_mode'* determines the mode in which the accuracies that are computed during the least-squares adjustment are returned in `CovPose`. If *'cov_pose_mode'* is set to *'standard_deviations'*, the 6 standard deviations of the 6 pose parameters are returned for each match. In contrast, if *'cov_pose_mode'* is set to *'covariances'*, `CovPose` contains the 36 values of the complete 6×6 covariance matrix of the 6 pose parameters.
List of values: *'standard_deviations'*, *'covariances'*
Default: *'standard_deviations'*

'border_model': The model is searched within those points of the domain of the image in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than `MinScore`. Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed by setting *'border_model'* to *'true'*, which will cause models that extend beyond the image border to be found if they achieve a score greater than `MinScore`. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the search will increase in this mode. Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels the model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., `dilation_circle`.
List of values: *'false'*, *'true'*
Default: *'false'*

Parameters

▷ **`Image`** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image in which the model should be found.

▷ **`ShapeModel3DID`** (input_control) . . . . . shape_model_3d ⇝ *handle*
Handle of the 3D shape model.

▷ **`MinScore`** (input_control) . . . . . real ⇝ *real*
Minimum score of the instances of the model to be found.
Default: 0.7
Suggested values: `MinScore` ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: $0 \leq$ `MinScore` ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **Greediness** (input_control) . . . . . real ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumLevels** (input_control) . . . . . integer-array ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of (optional) parameters for controlling the behavior of the operator.
Default: []
List of values: GenParamName ∈ {'longitude_min', 'longitude_max', 'latitude_min', 'latitude_max', 'cam_roll_min', 'cam_roll_max', 'dist_min', 'dist_max', 'num_matches', 'max_overlap', 'pose_refinement', 'cov_pose_mode', 'outlier_suppression', 'border_model', 'recompute_score'}

▷ **GenParamValue** (input_control) . . . . . attribute.name-array ⇝ *integer* / real / string
Values of the optional generic parameters.
Default: []
Suggested values: GenParamValue ∈ {-0.78, -0.35, -0.17, 0.0, 0.17, 0.35, 0.78, 0.1, 0.2, 0.5, 'none', 'false', 'true', 'least_squares', 'least_squares_high', 'least_squares_very_high', 'standard_deviations', 'covariances', 'medium', 'high'}

▷ **Pose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
3D pose of the 3D shape model.

▷ **CovPose** (output_control) . . . . . real-array ⇝ *real*
6 standard deviations or 36 covariances of the pose parameters.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the 3D shape model.

Example

```
read_object_model_3d (DXFModelFileName, 'm', [], [], ObjectModel3D, \
                      DxfStatus)
CamParam := ['area_scan_division',0.01221,2791,7.3958e-6,7.4e-6,\
             308.21,245.92,640,480]
create_shape_model_3d (ObjectModel3D, CamParam, 0, 0, 0, 'gba', \
                       -rad(20), rad(20), -rad(20), rad(20), 0, \
                       rad(360), 0.15, 0.2, 10, [], [], ShapeModel3DID)
grab_image_async (Image, AcqHandle, -1)
find_shape_model_3d (Image, ShapeModel3DID, 0.6, 0.9, 0, [], [], \
                     Pose, CovPose, Score)
project_shape_model_3d (ModelContours, ShapeModel3DID, CamParam, \
                        Pose, 'true', rad(30))
```

Result

If the parameter values are correct, the operator find_shape_model_3d returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised. If the model was created with find_shape_model_3d by setting *'metric'* to *'ignore_part_polarity'* and a multi-channel input image is passed in Image, the error 3359 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

create_shape_model_3d, read_shape_model_3d

Possible Successors

project_shape_model_3d

See also

convert_point_3d_cart_to_spher, convert_point_3d_spher_to_cart, create_cam_pose_look_at_point, trans_pose_shape_model_3d

Module

3D Metrology

get_shape_model_3d_contours (: ModelContours : ShapeModel3DID, Level, View : ViewPose)

Return the contour representation of a 3D shape model view.

The operator get_shape_model_3d_contours returns a representation of a single model view of the 3D shape model ShapeModel3DID as XLD contours in ModelContours. The parameters Level and View determine for which model view the contour representation should be returned, where Level denotes the pyramid level and View denotes the model view on this pyramid level.

The permitted range of values for Level and View can previously be determined by using the operator get_shape_model_3d_params and passing *'num_views_per_level'* for GenParamName.

The contours can be used to visualize and rate the 3D shape model that was created with create_shape_model_3d. With this it is possible, for example, to decide whether the number of pyramid levels in the model is appropriate or not. If the contours on the highest pyramid do not show enough details to be representative for the model view, the number of pyramid levels that are used during the search with find_shape_model_3d should be adjusted downwards. In contrast, if the contours show too many details even on the highest pyramid level, a higher number of pyramid levels should be chosen already during the creation of the 3D shape model by using create_shape_model_3d.

Additionally, the pose of the selected view is returned in ViewPose. It can be used, for example, to project the 3D shape model according to the view pose by using project_shape_model_3d. The rating of the model contours that was described above can then be performed by comparing the ModelContours to the projected model. Note that the position of the contours of the projection and the position of the model contours may slightly differ because of radial distortions.

Parameters

- ▷ **ModelContours** (output_object) ..... xld_cont-array ⇝ *object*
  Contour representation of the model view.
- ▷ **ShapeModel3DID** (input_control) ..... shape_model_3d ⇝ *handle*
  Handle of the 3D shape model.
- ▷ **Level** (input_control) ..... integer ⇝ *integer*
  Pyramid level for which the contour representation should be returned.
  Default: 1
  Suggested values: Level ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
  Restriction: Level >= 1
- ▷ **View** (input_control) ..... integer ⇝ *integer*
  View for which the contour representation should be returned.
  Default: 1
  Suggested values: View ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
  Restriction: View >= 1
- ▷ **ViewPose** (output_control) ..... pose ⇝ *real* / integer
  3D pose of the 3D shape model at the current view.

Result

If the parameters are valid, the operator get_shape_model_3d_contours returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_shape_model_3d`, `read_shape_model_3d`, `get_shape_model_3d_params`

Possible Successors

`create_shape_model_3d`

Module

3D Metrology

get_shape_model_3d_params (: : ShapeModel3DID, GenParamName : GenParamValue)

Return the parameters of a 3D shape model.

The operator `get_shape_model_3d_params` allows to query parameters of the 3D shape model. The names of the desired parameters are passed in the generic parameter `GenParamName`, the corresponding values are returned in `GenParamValue`.

The following parameters can be queried:

'cam_param': Internal parameters of the camera that is used for the matching.

'ref_rot_x': Reference orientation: Rotation around x-axis or x component of the Rodriguez vector (in radians or without unit).

'ref_rot_y': Reference orientation: Rotation around y-axis or y component of the Rodriguez vector (in radians or without unit).

'ref_rot_z': Reference orientation: Rotation around z-axis or z component of the Rodriguez vector (in radians or without unit).

'order_of_rotation': Meaning of the rotation values of the reference orientation.

'longitude_min': Minimum longitude of the model views.

'longitude_max': Maximum longitude of the model views.

'latitude_min': Minimum latitude of the model views.

'latitude_max': Maximum latitude of the model views.

'cam_roll_min': Minimum camera roll angle of the model views.

'cam_roll_max': Maximum camera roll angle of the model views.

'dist_min': Minimum camera-object-distance of the model views.

'dist_max': Maximum camera-object-distance of the model views.

'min_contrast': Minimum contrast of the objects in the search images.

'num_levels': User-specified number of pyramid levels.

'num_levels_max': Maximum number of used pyramid levels over all model views.

'optimization': Kind of optimization by reducing the number of model points.

'metric': Match metric.

'part_size': Size of the model parts that is used when *'metric'* is set to *'ignore_part_polarity'*.

'min_face_angle': Minimum 3D face angle for which 3D object model edges are included in the 3D shape model.

'min_size': Minimum size of the projected 3D object model edge (in number of pixels) to include the projected edge in the 3D shape model.

'model_tolerance': Maximum acceptable tolerance of the projected 3D object model edges (in pixels).

'num_views_per_level': Number of model views per pyramid level. For each pyramid level the number of views that are stored in the 3D shape model are returned. Thus, the number of returned elements corresponds to the number of used pyramid levels, which can be queried with *'num_levels_max'*. Note that for pyramid levels below *'lowest_model_level'* (see documentation of `create_shape_model_3d`), the value *0* is returned.

'reference_pose': Reference position and orientation of the 3d shape model. The returned pose is in the form ${}^{rcs}\mathbf{P}_{mcs}$, where *rcs* denotes the reference coordinates system and *mcs* the model coordinate system (which is a 3D world coordinate system), see Transformations / Poses and "Solution Guide III-C - 3D Vision". Hence, it describes the pose of the coordinate system that is used in the underlying 3D object model relative to the internally used reference coordinate system of the 3D shape model. With this pose, points given in the object coordinate system can be transformed into the reference coordinate system.

'reference_point': 3D coordinates of the reference point of the underlying 3D object model.

'bounding_box1': Smallest enclosing axis-parallel cuboid of the underlying 3D object model in the following order: [min_x, min_y, min_z, max_x, max_y, max_z].

'fast_pose_refinement': Describes whether the pose refinement during the search is performed in a sped up mode (*'true'*) or in the conventional mode (*'false'*).

'lowest_model_level': Lowest pyramid level down to which views are stored in the model.

'union_adjacent_contours': Describes whether in project_shape_model_3d adjacent contours should be joined or not.

A detailed description of the parameters can be looked up with the operator create_shape_model_3d.

It is possible to query the values of several parameters with a single operator call by passing a tuple containing the names of all desired parameters to GenParamName. As a result a tuple of the same length with the corresponding values is returned in GenParamValue. Note that this is solely possible for parameters that return only a single value.

Parameters

▷ **ShapeModel3DID** (input_control) . . . . . shape_model_3d ⇝ *handle*
Handle of the 3D shape model.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that are to be queried for the 3D shape model.
Default: 'num_levels_max'
List of values: GenParamName ∈ {'cam_param', 'ref_rot_x', 'ref_rot_y', 'ref_rot_z', 'order_of_rotation', 'longitude_min', 'longitude_max', 'latitude_min', 'latitude_max', 'cam_roll_min', 'cam_roll_max', 'dist_min', 'dist_max', 'min_contrast', 'num_levels', 'num_levels_max', 'optimization', 'metric', 'part_size', 'min_face_angle', 'min_size', 'model_tolerance', 'num_views_per_level', 'reference_pose', 'reference_point', 'bounding_box1', 'fast_pose_refinement', 'lowest_model_level', 'union_adjacent_contours'}

▷ **GenParamValue** (output_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
Values of the generic parameters.

Result

If the parameters are valid, the operator get_shape_model_3d_params returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_shape_model_3d, read_shape_model_3d

Possible Successors

find_shape_model_3d

See also

convert_point_3d_cart_to_spher, convert_point_3d_spher_to_cart, create_cam_pose_look_at_point, trans_pose_shape_model_3d

Module

3D Metrology

project_shape_model_3d (: ModelContours : ShapeModel3DID, CamParam, Pose, HiddenSurfaceRemoval, MinFaceAngle :)

Project the edges of a 3D shape model into image coordinates.

The operator project_shape_model_3d projects the edges of the 3D object model that was used to create the 3D shape model ShapeModel3DID into the image coordinate system and returns the projected edges in ModelContours. The coordinates of the 3D object model are given in the 3D world coordinate system (*mcs*). First, they are transformed into the camera coordinate system (*ccs*) using the external camera parameters given in Pose. Then, these coordinates are projected into the image coordinate system based on the internal camera parameters CamParam.

The internal camera parameters CamParam describe the projection characteristics of the camera (see Calibration). The Pose is in the form ${}^{ccs}\mathbf{P}_{mcs}$, see Transformations / Poses and "Solution Guide III-C - 3D Vision". Hence, it describes the position and orientation of the model coordinate system defined by the 3D object model relative to the camera coordinate system.

The parameter HiddenSurfaceRemoval can be used to switch on or to switch off the removal of hidden surfaces. If HiddenSurfaceRemoval is set to *'true'*, only those projected edges are returned that are not hidden by faces of the 3D object model. If HiddenSurfaceRemoval is set to *'false'*, all projected edges are returned. This is faster than a projection with HiddenSurfaceRemoval set to *'true'*.

If the system variable (see set_system) *'opengl_hidden_surface_removal_enable'* is set to *'true'* (which is default if it is available) and HiddenSurfaceRemoval is set to *'true'*, the projection of the model is accelerated using the graphics card. Depending on the graphics card this is significantly faster than the non accelerated algorithm. Be aware that the results of the OpenGL projection are slightly different compared to the analytic projection. Notable, only the contours visible through CamParam are projected in this mode.

3D edges are only projected if the angle between the two 3D faces that are incident with the 3D edge is at least MinFaceAngle. If MinFaceAngle is set to *0.0*, all edges are projected. If MinFaceAngle is set to π (equivalent to 180 degrees), only the silhouette of the 3D object model is returned. This parameter can be used to suppress edges within curved surfaces, e.g., the surface of a cylinder.

If for the model creation with create_shape_model_3d the parameter *'union_adjacent_contours'* was activated, adjacent contours are joined.

project_shape_model_3d and project_object_model_3d return the same result if the 3D object model that was used to create the 3D shape model is passed to project_object_model_3d.

project_shape_model_3d is especially useful in order to visualize the matches that are returned by find_shape_model_3d in the case that the underlying 3D object model is no longer available.

Parameters

- ▷ **ModelContours** (output_object) ..... xld_cont-array ⇝ *object*
  Contour representation of the model view.
- ▷ **ShapeModel3DID** (input_control) ..... shape_model_3d ⇝ *handle*
  Handle of the 3D shape model.
- ▷ **CamParam** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameters.
- ▷ **Pose** (input_control) ..... pose ⇝ *real* / integer
  3D pose of the 3D shape model in the world coordinate system.
- ▷ **HiddenSurfaceRemoval** (input_control) ..... string ⇝ *string*
  Remove hidden surfaces?
  Default: 'true'
  List of values: HiddenSurfaceRemoval ∈ {'true', 'false'}
- ▷ **MinFaceAngle** (input_control) ..... angle.rad ⇝ *real* / integer
  Smallest face angle for which the edge is displayed
  Default: 0.523599
  Suggested values: MinFaceAngle ∈ {0.17, 0.26, 0.35, 0.52}

Result

If the parameters are valid, the operator project_shape_model_3d returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_shape_model_3d, read_shape_model_3d, get_shape_model_3d_params, find_shape_model_3d

Alternatives

project_object_model_3d

See also

convert_point_3d_cart_to_spher, convert_point_3d_spher_to_cart, create_cam_pose_look_at_point, trans_pose_shape_model_3d

Module

3D Metrology

read_shape_model_3d (: : FileName : ShapeModel3DID)

Read a 3D shape model from a file.

The operator read_shape_model_3d reads a 3D shape model, which has been written with write_shape_model_3d, from the file FileName. The default HALCON file extension for the 3D shape model is 'sm3'.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
File name.
File extension: .sm3

▷ **ShapeModel3DID** (output_control) ..... shape_model_3d ⇝ *handle*
Handle of the 3D shape model.

Result

If the file name is valid, the operator read_shape_model_3d returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

find_shape_model_3d, get_shape_model_3d_params

See also

create_shape_model_3d, clear_shape_model_3d

Module

3D Metrology

serialize_shape_model_3d (
: : ShapeModel3DID : SerializedItemHandle)

Serialize a 3D shape model.

serialize_shape_model_3d serializes the data of a 3D shape model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_shape_model_3d is converted to a serialized item. The 3D shape model is defined by the handle ShapeModel3DID. The serialized 3D shape model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_shape_model_3d.

Parameters

▷ **ShapeModel3DID** (input_control) ..... shape_model_3d ⇝ *handle*
Handle of the 3D shape model.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_shape_model_3d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_shape_model_3d

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_shape_model_3d

Module

3D Metrology

trans_pose_shape_model_3d (: : ShapeModel3DID, PoseIn, Transformation : PoseOut)

Transform a pose that refers to the coordinate system of a 3D object model to a pose that refers to the reference coordinate system of a 3D shape model and vice versa.

The operator trans_pose_shape_model_3d transforms the pose PoseIn into the pose PoseOut by using the transformation direction specified in Transformation. In the majority of cases, the operator will be used to transform a camera pose that is given relative to the source coordinate system to a camera pose that refers to the target coordinate system.

The pose can be transformed between two coordinate systems. The first coordinate system is the reference coordinate system of the 3D shape model (*ref*) that is passed in ShapeModel3DID. The origin of the reference coordinate system lies at the reference point of the underlying 3D object model. The orientation of the reference coordinate system is determined by the reference orientation that was specified when creating the 3D shape model with create_shape_model_3d.

The second coordinate system is the world coordinate system, i.e., the coordinate system of the 3D object model (*mcs*) that underlies the 3D shape model. This coordinate system is implicitly determined by the coordinates that are stored in the CAD file that was read by using read_object_model_3d.

If Transformation is set to *'ref_to_model'*, it is assumed that PoseIn refers to the reference coordinate system of the 3D shape model. Thus, PoseIn is ${}^{cs}\mathbf{P}_{rcs}$, where *cs* denotes the coordinate system the input pose transforms into (e.g., the camera coordinate system). For further information we refer to Transformations / Poses and "Solution Guide III-C - 3D Vision". The resulting output pose PoseOut in this case refers to the coordinate system of the 3D object model, thus ${}^{cs}\mathbf{P}_{mcs}$.

If Transformation is set to *'model_to_ref'*, it is assumed that PoseIn refers to the coordinate system of the 3D object model, ${}^{cs}\mathbf{P}_{mcs}$. The resulting output pose PoseOut in this case refers to the reference coordinate system of the 3D shape model, thus ${}^{cs}\mathbf{P}_{rcs}$.

The relative pose of the two coordinate systems can be queried by passing *'reference_pose'* for GenParamName in the operator get_shape_model_3d_params.

Parameters

▷ **ShapeModel3DID** (input_control) . . . . . shape_model_3d ⇝ *handle*
Handle of the 3D shape model.

▷ **PoseIn** (input_control) . . . . . pose ⇝ *real* / integer
Pose to be transformed in the source system.

▷ **Transformation** (input_control) . . . . . string ⇝ *string*
Direction of the transformation.
Default: 'ref_to_model'
List of values: `Transformation` ∈ {'ref_to_model', 'model_to_ref'}

▷ **PoseOut** (output_control) . . . . . pose ⇝ *real* / integer
Transformed 3D pose in the target system.

Result

If the parameters are valid, the operator `trans_pose_shape_model_3d` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`find_shape_model_3d`

Alternatives

`hom_mat3d_translate`, `hom_mat3d_rotate`

Module

3D Metrology

write_shape_model_3d (: : ShapeModel3DID, FileName :)

Write a 3D shape model to a file.

The operator `write_shape_model_3d` writes a 3D shape model to the file `FileName`. The model can be read again with `read_shape_model_3d`. The default HALCON file extension for the 3D shape model is 'sm3'.

Parameters

▷ **ShapeModel3DID** (input_control) . . . . . shape_model_3d ⇝ *handle*
Handle of the 3D shape model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: `.sm3`

Result

If the file name is valid (write permission), the operator `write_shape_model_3d` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_shape_model_3d`

Module

3D Metrology

3.6 Surface-Based

clear_surface_matching_result (: : SurfaceMatchingResultID :)

Free the memory of a surface matching result.

The operator clear_surface_matching_result frees the memory of a surface matching result that was created by find_surface_model or refine_surface_model_pose. After calling clear_surface_matching_result, the result can no longer be used. The handle SurfaceMatchingResultID becomes invalid.

Parameters

▷ **SurfaceMatchingResultID** (input_control) . . . . . surface_matching_result(-array) ⇝ *handle*
Handle of the surface matching result.

Result

If the handle of the result is valid, the operator clear_surface_matching_result returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SurfaceMatchingResultID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

find_surface_model, refine_surface_model_pose

See also

find_surface_model, refine_surface_model_pose

Module

3D Metrology

clear_surface_model (: : SurfaceModelID :)

Free the memory of a surface model.

The operator clear_surface_model frees the memory of a surface model that was created by read_surface_model or create_surface_model. After calling clear_surface_model, the model can no longer be used. The handle SurfaceModelID becomes invalid.

Parameters

▷ **SurfaceModelID** (input_control) . . . . . surface_model(-array) ⇝ *handle*
Handle of the surface model.

Result

If the handle of the model is valid, the operator clear_surface_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SurfaceModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

read_surface_model, create_surface_model

See also

read_surface_model, create_surface_model

Module

3D Metrology

create_surface_model (: : ObjectModel3D, RelSamplingDistance, GenParamName, GenParamValue : SurfaceModelID)

Create the data structure needed to perform surface-based matching.

The operator create_surface_model creates a model for surface-based matching for the 3D object model ObjectModel3D. The 3D object model can, for example, have been read previously from a file by using read_object_model_3d, or been created by using xyz_to_object_model_3d. The created surface model is returned in SurfaceModelID.

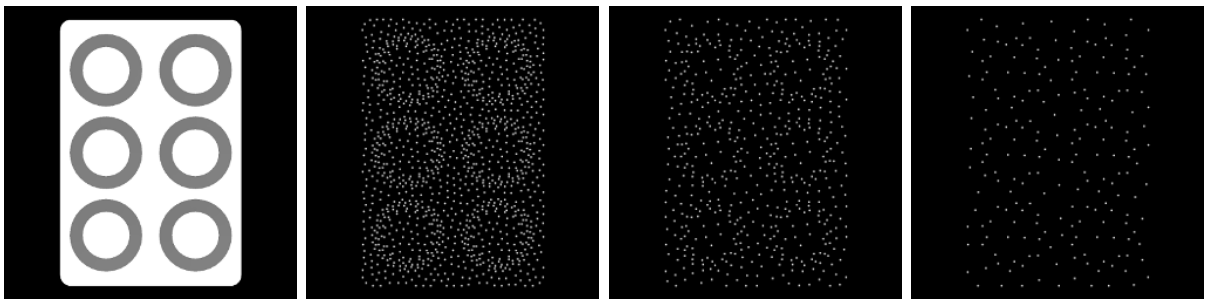
Additional parameters of the surface model can be set with set_surface_model_param after the model was created.

The creation of the surface model requires that the 3D object model contains points and normals. The following combinations are possible:

- points and point normals;
- points and a triangular or polygon mesh, e.g., from a CAD file;
- points and a 2D-Mapping, e.g., an XYZ image triple converted with xyz_to_object_model_3d.

Note that the direction and orientation (inward or outward) of the normals of the model are important for matching. For edge-supported surface-based matching the normals need to point inwards and further the model must contain a triangular or polygon mesh (see below).

The surface model is created by sampling the 3D object model with a certain distance. The sampling distance must be specified in the parameter RelSamplingDistance and is parametrized relative to the diameter of the axis-parallel bounding box of the 3D object model. For example, if RelSamplingDistance is set to *0.05* and the diameter of ObjectModel3D is *10 cm*, the points sampled from the object's surface will be approximately *5 mm* apart. The sampled points are used for the approximate matching in the operator find_surface_model (see below). The sampled points can be obtained with the operator get_surface_model_param using the value *'sampled_model'*. Note that outlier points in the object model should be avoided, as they would corrupt the diameter. Reducing RelSamplingDistance leads to more points, and in turn to a more stable but slower matching. Increasing RelSamplingDistance leads to less points, and in turn to a less stable but faster matching.



(1) (2) (3) (4)

(1) Original 3D model. (2) 3D model sampled with RelSamplingDistance = *0.02*. (3) RelSamplingDistance = *0.03*. (4) RelSamplingDistance = *0.05*.

The sampled points are used for finding the object model in a scene by using the operator find_surface_model. For this, all possible pairs of points from the point set are examined, and the distance and relative surface orientation of each pair is computed. Both values are discretized and stored for matching. The generic parameters *'feat_step_size_rel'* and *'feat_angle_resolution'* can be used to set the discretization of the distance and the orientation angles, respectively (see below).

The 3D object model is sampled a second time for the pose refinement. The second sampling is done with a smaller sampling distance, leading to more points. The generic parameter *'pose_ref_rel_sampling_distance'* sets the sampling distance relative to the object's diameter. Decreasing the value results in a more accurate pose refinement but a larger model and a slower model generation and matching. Increasing the value leads to a less accurate pose refinement but a smaller model and faster model generation and matching (see below).

Surface-based matching can additionally use 3D edges to improve the alignment. This is particularly helpful for objects that are planar or contain larger planar sides, such that they are found in incorrect rotations or in a background plane. In order to allow find_surface_model to also align edges, the surface model must be trained by setting the generic parameter *'train_3d_edges'* to *'true'*. In this case, the model must contain a triangular or polygon mesh where the order of the points results in normals that point inwards. Also, the training for edge-supported surface-based matching requires OpenGL 2.1, GLSL 1.2, and the OpenGL extensions GL_EXT_framebuffer_object and GL_EXT_framebuffer_blit. Note that the training can take significantly longer than without edge-support.

Additionally, the model can be prepared to support view-based score computation. This is particularly helpful for models where only a small part of the 3D object model is visible, which results in low scores if the ratio to the total number of points is used. Accordingly, the view-based score is computed using the ratio of the matched points to the maximum number of potentially visible model points from a certain viewpoint. In order to allow find_surface_model to compute a view-based score, the surface model must be trained by setting the generic parameter *'train_view_based'* to *'true'*. Similar to *'train_3d_edges'*, the model must contain a triangular or polygon mesh where the order of the points results in normals that point inwards.

Note that using noisy data for the creation of your 3D object model results in the computation of deficient surface normals. Especially when the model is prepared for the use with 3D edges or the support of view-based score, this can lead to unreliable scores. In order to reduce noisy 3D data you can, e.g., use smooth_object_model_3d or simplify_object_model_3d.

The generic parameter pair GenParamName and GenParamValue are used to set additional parameters for the model generation. GenParamName contains the tuple of parameter names that shall be set and GenParamValue contains the corresponding values. The following values are possible for GenParamName:

'model_invert_normals': Invert the orientation of the surface normals of the model. The normal orientation needs to be known for the model generation. If both the model and the scene are acquired with the same setup, the normals will already point in the same direction. If the model was loaded from a CAD file, the normals might point into the opposite direction. If you experience the effect that the model is found on the 'outside' of the scene surface and the model was created from a CAD file, try to set this parameter to *'true'*. Also, make sure that the normals in the CAD file all point either outward or inward, i.e., are oriented consistently. The normal direction is irrelevant for the pose refinement of the surface model. Therefore, if the object model is only used with the operator refine_surface_model_pose, the value of *'model_invert_normals'* has no effect on the result.

List of values: *'false'*, *'true'*

Default: *'false'*

'pose_ref_rel_sampling_distance': Set the sampling distance for the pose refinement relative to the object's diameter. Decreasing this value leads to a more accurate pose refinement but a larger model and slower model generation and refinement. Increasing the value leads to a less accurate pose refinement but a smaller model and faster model generation and matching.

Suggested values: *0.05*, *0.02*, *0.01*, *0.005*

Default: *0.01*

Restriction: *0 < 'pose_ref_rel_sampling_distance' < 1*

'feat_step_size_rel': Set the discretization distance of the point pair distance relative to the object's diameter. This value defaults to the value of RelSamplingDistance. It is not recommended to change this value. For very noisy scenes, the value can be increased to improve the robustness of the matching against noisy points.

Suggested values: *0.1*, *0.05*, *0.03*

Default: Value of RelSamplingDistance

Restriction: *0 < 'feat_step_size_rel' < 1*

'feat_angle_resolution': Set the discretization of the point pair orientation as the number of subdivisions of the angle. It is recommended to not change this value. Increasing the value increases the precision of the matching but decreases the robustness against incorrect normal directions. Decreasing the value decreases the precision of the matching but increases the robustness against incorrect normal directions. For very noisy scenes where the normal directions can not be computed accurately, the value can be set to *25* or *20*.

Suggested values: *20*, *25*, *30*

Default: *30*

Restriction: *'feat_angle_resolution'* > *1*

'train_3d_edges': Enable the training for edge-supported surface-based matching and refinement. In this case the model must contain a mesh, i.e. triangles or polygons. Also, it is important that the computed normal vectors point inwards. This parameter requires OpenGL.

List of values: *'false'*, *'true'*

Default: *'false'*

'train_view_based': Enable the training for view-based score computation for surface-based matching and refinement. In this case the model must contain a mesh, i.e. triangles or polygons. Also, it is important that the computed normal vectors point inwards. This parameter requires OpenGL.

List of values: *'false'*, *'true'*

Default: *'false'*

'train_self_similar_poses': Prepares the surface model for optimizations regarding self-similar, almost symmetric poses. For this, poses are found under which the model is very similar to itself, i.e., poses that can be distinguished only by very small properties of the model (such as boreholes) and that can be confused by `find_surface_model`. When calling `find_surface_model`, it will automatically be determined which of those self-similar poses are correct.

List of values: *'false'*, *'true'*

Default: *'false'*

Parameters

▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the 3D object model.

▷ **`RelSamplingDistance`** (input_control) . . . . . real ⇝ *real*
Sampling distance relative to the object's diameter
Default: 0.03
Suggested values: `RelSamplingDistance` ∈ {0.1, 0.05, 0.03, 0.02, 0.01}
Restriction: 0 < RelSamplingDistance < 1

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters.
Default: []
Suggested values: `GenParamName` ∈ {'model_invert_normals', 'pose_ref_rel_sampling_distance', 'feat_step_size_rel', 'feat_angle_resolution', 'train_3d_edges', 'train_view_based', 'train_self_similar_poses'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: `GenParamValue` ∈ {0, 1, 'true', 'false', 0.005, 0.01, 0.02, 0.05, 0.1}

▷ **`SurfaceModelID`** (output_control) . . . . . surface_model ⇝ *handle*
Handle of the surface model.

Result

`create_surface_model` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, get_object_model_3d_params, surface_normals_object_model_3d

Possible Successors

find_surface_model, refine_surface_model_pose, get_surface_model_param, write_surface_model, clear_surface_model, set_surface_model_param

Alternatives

read_surface_model

See also

find_surface_model, refine_surface_model_pose, read_surface_model, write_surface_model, clear_surface_model, set_surface_model_param

References

Bertram Drost, Markus Ulrich, Nassir Navab, Slobodan Ilic: "Model Globally, Match Locally: Efficient and Robust 3D Object Recognition." Computer Vision and Pattern Recognition, pp. 998-1005, 2010.

Module

3D Metrology

deserialize_surface_model (: : SerializedItemHandle : SurfaceModelID)

Deserialize a surface model.

deserialize_surface_model deserializes a surface model, that was serialized by serialize_surface_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized surface model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created surface model with the handle SurfaceModelID.

Parameters

- ▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
  Handle of the serialized item.
- ▷ **SurfaceModelID** (output_control) ..... surface_model ⇝ *handle*
  Handle of the surface model.

Result

If the parameters are valid, the operator deserialize_surface_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, fread_serialized_item, receive_serialized_item, serialize_surface_model

Possible Successors

find_surface_model, refine_surface_model_pose, get_surface_model_param, clear_surface_model, find_surface_model_image, refine_surface_model_pose_image

Alternatives

create_surface_model

See also

create_surface_model, read_surface_model, write_surface_model

Module

3D Metrology

```
find_surface_model ( : : SurfaceModelID, ObjectModel3D,
    RelSamplingDistance, KeyPointFraction, MinScore,
    ReturnResultHandle, GenParamName, GenParamValue : Pose, Score,
    SurfaceMatchingResultID )
```

Find the best matches of a surface model in a 3D scene.

The operator `find_surface_model` finds the best matches of the surface model `SurfaceModelID` in the 3D scene `ObjectModel3D` and returns their pose in `Pose`.

The matching is divided in three steps:

1. `Approximate matching`
2. `Sparse pose refinement`
3. `Dense pose refinement`

These steps are described in more detail in the technical note `Surface-Based Matching`. The generic parameters used to control these steps are described in the respective sections below. The further paragraphs describe the parameters and mention further points to note.

The matching process and the parameters can be visualized and inspected using the HDevelop procedure `debug_find_surface_model`.

Points to Note

Matching the surface model uses points and normals of the 3D scene `ObjectModel3D`. The scene shall provide one of the following options:

- points and point normals.
- points and a 2D-Mapping, e.g., an XYZ image triple converted with `xyz_to_object_model_3d`. In this case the normals are calculated using the 2D-Mapping.
- points only. The normals are estimated based on the 3D neighborhood. Note, this option is not recommended, since it generally leads to a longer processing time and additionally the computed normals are usually less accurate, leading to less accurate results.

It is important for an accurate `Pose` that the normals of the scene and the model point in the same direction (see *'scene_invert_normals'*).

If the model was trained for edge-supported surface-based matching and the edge-supported matching has not been turned off via *'use_3d_edges'*, only the second combination is possible, i.e., the scene must contain a 2D mapping.

If the model was trained for edge-supported surface-based matching and the scene contains a mapping, normals contained in the input point cloud are not used (see *'scene_normal_computation'* below).

Further, for models which were trained for edge-supported surface-based matching it is necessary that the normal vectors point inwards.

Note that triangles or polygons in the passed scene are ignored. Instead, only the vertices are used for matching. It is thus in general not recommended to use this operator on meshed scenes, such as CAD data. Instead, such a scene must be sampled beforehand using `sample_object_model_3d` to create points and normals (e.g., using the method *'fast_compute_normals'*).

When using noisy point clouds, e.g., from time-of-flight cameras, the generic parameter *'scene_normal_computation'* could be set to *'mls'* in order to obtain more robust results (see below).

Parameter Description

`SurfaceModelID` is the handle of the surface model. The model must have been created previously with `create_surface_model` or read in with `read_surface_model`, respectively. Certain surface model parameters influencing the matching can be set using `set_surface_model_param`, such as *'pose_restriction_max_angle_diff'* restricting the allowed range of rotations.

`ObjectModel3D` is the handle of the 3D object model containing the scene in which the matches are searched. Note that in most cases, it is assumed the scene was observed from a camera looking along the z-axis. This is important to align the scene normals if they are re-computed (see *'scene_normal_computation'* below). In contrast, when the model was trained for edge-supported surface-based matching and the scene contains a mapping, normals are automatically aligned consistently.

The parameter RelSamplingDistance controls the sampling distance during the step Approximate matching and the Score calculation during the step Sparse pose refinement. Its value is given relative to the diameter of the surface model. Decreasing RelSamplingDistance leads to more sampled points, and in turn to a more stable but slower matching. Increasing RelSamplingDistance reduces the number of sampled scene points, which leads to a less stable but faster matching. For an illustration showing different values for RelSamplingDistance, please refer to the operator create_surface_model. The sampled scene points can be retrieved for a visual inspection using the operator get_surface_matching_result. For a robust matching it is recommended that at least 50-100 scene points are sampled for each object instance.

The parameter KeyPointFraction controls how many points out of the sampled scene points are selected as key points. For example, if the value is set to *0.1*, 10% of the sampled scene points are used as key points. For stable results it is important that each instance of the object is covered by several key points. Increasing KeyPointFraction means that more key points are selected from the scene, resulting in a slower but more stable matching. Decreasing KeyPointFraction has the inverse effect and results in a faster but less stable matching. The operator get_surface_matching_result can be used to retrieve the selected key points for visual inspection.

The parameter MinScore can be used to filter the results. Only matches with a score exceeding the value of MinScore are returned. If MinScore is set to zero, all matches are returned.

For edged-supported surface-based matching (see create_surface_model) four different sub-scores are determined (see their explanation below). For surface-based matching models where view-based score computation is trained (see create_surface_model), an additional fifth sub-score is determined. As a consequence, you can filter the results based on each of them by passing a tuple with up to five threshold values to MinScore. These threshold values are sorted in the order of the scores (see below) and missing entries are regarded as *0*, meaning no filtering based on this sub-score. To find suitable values for the thresholds, the corresponding sub-scores of found object instances can be obtained using get_surface_matching_result. Depending on the settings, not all sub-scores might be available. The thresholds for unavailable sub-scores are ignored. The five sub-scores, whose threshold values have to be passed in exactly this order in MinScore, are:

1. The overall score as returned in Score and through *'score'* by get_surface_matching_result,
2. the surface fraction of the score, i.e., how much of the object's surface was detected in the scene, returned through *'score_surface'* by get_surface_matching_result,
3. the 3D edge fraction of the score, i.e., how well the 3D edges of the object silhouette are aligned with the 3D edges detected in the scene returned through *'score_3d_edges'* by get_surface_matching_result,
4. the 2D edge fraction of the score, i.e., how well the object silhouette projected into the images aligns with edges detected in the images (available only for the operators find_surface_model_image and refine_surface_model_pose_image), returned through *'score_2d_edges'* by get_surface_matching_result, and
5. the view-based score, i.e., how many model points were detected in the scene, in relation to how many of the object points are potentially visible from the determined viewpoint, returned through *'score_view_based'* by get_surface_matching_result.

The parameter ReturnResultHandle determines if a surface matching result handle is returned or not. If the parameter is set to *'true'*, the handle is returned in the parameter SurfaceMatchingResultID. Additional details of the matching process can be queried with the operator get_surface_matching_result using that handle.

The parameters GenParamName and GenParamValue are used to set generic parameters. Both get a tuple of equal length, where the tuple passed to GenParamName contains the names of the parameters to set, and the tuple passed to GenParamValue contains the corresponding values. The possible parameter names and values are described in the paragraph The three steps of the matching.

The output parameter Pose gives the 3D poses of the found object instances. For every found instance of the surface model its pose is given in the scene coordinate system, thus the pose is in the form ${}^{scs}\mathbf{P}_{mcs}$, where *scs* denote the coordinate system of the scene (which often is identical with the coordinate system of the sensor, the camera coordinate system) and *mcs* the model coordinate system (which is a 3D world coordinate system), see Transformations / Poses and "Solution Guide III-C - 3D Vision". Thereby, the pose refers to the original coordinate system of the 3D object model that was passed to create_surface_model.

The output parameter Score returns a score for each match. Its value and interpretation differs for the cases distinguished below.

- **With pose refinement**
  For a matching with pose refinement, the score depends on whether edge-support was activated:
  - Without edge-support, compute the surface fraction, i.e. the approximate fraction of the object's surface that is visible in the scene. This is done by counting the number of model points that have a corresponding scene point and dividing this number either by:
    * the total number of points on the model, if the surface-based model is not prepared for view-based score computation
      or by:
    * the maximum number of potentially visible model points based on the current viewpoint, if the surface-based model is prepared for view-based score computation.

    $0 \leq Score \leq 1$
  - With edge-support, compute the geometric mean of the surface fraction and the edge fraction. The surface fraction is affected by whether the surface-based model is prepared for view-based score computation or not, as explained above. The edge fraction is the number of points from the sampled model edges that are aligned with edges of the scene, divided by the maximum number of potentially visible points of edges on the model. Note that if the edges are extracted from multiple viewpoints, this might lead to score greater than 1.
    $0 \leq Score \leq 1$ (if the scene was acquired from one single viewpoint)
    $0 \leq Score \leq N$ (if the scene was merged from scenes that were acquired from N different viewpoints)

  Note that for the computation of the score after the sparse pose refinement, the sampled scene points are used. For the computation of the score after the dense pose refinement, all scene points are used. Therefore, after the dense pose refinement, the score values does not depend on the sampling distance of the scene.
- **Without pose refinement**
  If only the first step, `Approximate Matching`, out of the three steps described in `The three steps of the matching` takes place, the possible score value and interpretation only differs whether there is edge-support or not:
  - Without edge-support:
    The score is the approximate number of points from the subsampled scene that lie on the found object.
    $Score \geq 0$
  - With edge-support:
    The score is the approximate number of points from the subsampled scene that lie on the found object multiplied with the number of points from the sampled scene edges that are aligned with edges of the model.
    $Score \geq 0$

The output parameter `SurfaceMatchingResultID` returns a handle for the surface matching result. Using this handle, additional details of the matching process can be queried with the operator `get_surface_matching_result`. Note, that in order to return the handle, `ReturnResultHandle` has to be set to *'true'*.

The Three Steps of the Matching

The matching is divided into three steps:

1. Approximate matching The approximate poses of the instances of the surface model in the scene are searched.
The following generic parameters control the approximate matching and can be set with `GenParamName` and `GenParamValue`:

'num_matches': Sets the maximum number of matches that are returned.
Suggested values: *1*, *2*, *5*
Default: *1*
Restriction: *'num_matches' > 0*

'max_overlap_dist_rel': For efficiency reasons, the maximum overlap can not be defined in 3D. Instead, only the minimum distance between the centers of the axis-aligned bounding boxes of two matches can be specified with *'max_overlap_dist_rel'*. The value is set relative to the diameter of the object. Once an object with a high `Score` is found, all other matches are suppressed if the centers of their bounding boxes lie too close to the center of the first object. If the resulting matches must not overlap, the value for *'max_overlap_dist_rel'* should be set to *1.0*.

Note that only one of the parameters *'max_overlap_dist_rel'* and *'max_overlap_dist_abs'* should be set. If both are set, only the value of the last modified parameter is used.
Suggested values: *0.1*, *0.5*, *1*
Default: *0.5*
Restriction: *'max_overlap_dist_rel'* >= *0*

'max_overlap_dist_abs': This parameter has the same effect as the parameter *'max_overlap_dist_rel'*. Note that in contrast to *'max_overlap_dist_rel'*, the value for *'max_overlap_dist_abs'* is set as an absolute value. See *'max_overlap_dist_rel'* above, for a description of the effect of this parameter.
Note that only one of the parameters *'max_overlap_dist_rel'* and *'max_overlap_dist_abs'* should be set. If both are set, only the value of the last modified parameter is used.
Suggested values: *1*, *2*, *3*
Restriction: *'max_overlap_dist_abs'* >= *0*

'scene_normal_computation': This parameter controls the normal computation of the sampled scene.
In the default mode *'fast'*, in most cases normals from the 3D scene are used (if it already contains normals) or computed based on a small neighborhood of points (if not). The computed normals n are then oriented such that $n_z \geq 0$ in case no original normals exist. This orientation of $n_z \geq 0$ implies the assumption that the scene was observed from a camera looking along the z-axis.
In the default mode *'fast'*, in case the model was trained for edge-supported surface-based matching and the scene contains a mapping, input normals are not used and normals are always computed from the mapping contained in the 3D scene. Further, the computed normals are oriented inwards consistently with respect to the mapping.
In the mode *'mls'*, normals are recomputed based on a larger neighborhood and using the more complex but often more accurate *'mls'* method. A more detailed description of the *'mls'* method can be found in the description of the operator `surface_normals_object_model_3d`. The *'mls'* mode is intended for noisy data, such as images from time-of-flight cameras. The recomputed normals are oriented as the normals in mode *'fast'*.
List of values: *'fast'*, *'mls'*
Default: *'fast'*

'scene_invert_normals': Invert the orientation of the surface normals of the scene. The orientation of surface normals of the scene have to match with the orientation of the model. If both the model and the scene are acquired with the same setup, the normals will already point in the same direction. If you experience the effect that the model is found on the 'outside' of the scene surface, try to set this parameter to *'true'*. Also, make sure that the normals in the scene all point either outward or inward, i.e., are oriented consistently. For edge-supported surface-based matching, the normal vectors have to point inwards, but typically are automatically generated flipped inwards consistently with respect to the mapping. The orientation of the normals can be inspected using the procedure `debug_find_surface_model`.
List of values: *'false'*, *'true'*
Default: *'false'*

'3d_edges': Allows to manually set the 3D scene edges for edge-supported surface-based matching, i.e. if the surface model was created with *'train_3d_edges'* enabled. The parameter must be a 3D object model handle. The edges are usually a result of the operator `edges_object_model_3d` but can further be filtered in order to remove outliers. If this parameter is not given, `find_surface_model` will internally extract the edges similar to the operator `edges_object_model_3d`.

'3d_edge_min_amplitude_rel': Sets the threshold when extracting 3D edges for edge-supported surface-based matching, i.e. if the surface model was created with *'train_3d_edges'* enabled. The threshold is set relative to the diameter of the object. Note that if edges were passed manually with the generic parameter *'3d_edges'*, this parameter is ignored. Otherwise, it behaves identically to the parameter `MinAmplitude` of operator `edges_object_model_3d`.
Suggested values: *0.05*, *0.1*, *0.5*
Default: *0.05*
Restriction: *'3d_edge_min_amplitude_rel'* >= *0*

'3d_edge_min_amplitude_abs': Similar to *'3d_edge_min_amplitude_rel'*, however, the value is given as absolute distance and not relative to the object diameter.
Restriction: *'3d_edge_min_amplitude_abs'* >= *0*

'viewpoint': This parameter specifies the viewpoint from which the 3D data is seen. It is used for surface models that are prepared for view-based score computation (i.e. with *'train_view_based'* enabled) to get the maximum number of potentially visible points of the model based on the current viewpoint. For this, `GenParamValue` must contain a string consisting of the three coordinates (x, y, and z) of the view-

point, separated by spaces. The viewpoint is defined in the same coordinate frame as ObjectModel3D and should roughly correspond to the position the scene was acquired from. A visualization of the viewpoint can be created using the procedure debug_find_surface_model in order to inspect its position.
Default: *'0 0 0'*

'max_gap': Gaps in the 3D data are closed, as far as they do not exceed the maximum gap size *'max_gap'* [pixels] and the surface model was created with *'train_3d_edges'* enabled. Larger gaps will contain edges at their boundary, while gaps smaller than this value will not. This suppresses edges around smaller patches that were not reconstructed by the sensor as well as edges at the more distant part of a discontinuity. For sensors with very large resolutions, the value should be increased to avoid spurious edges. Note that if edges were passed manually with the generic parameter *'3d_edges'*, this parameter is ignored. Otherwise, it behaves identically to the parameter GenParamName of the operator edges_object_model_3d when *'max_gap'* is set.
The influence of *'max_gap'* can be inspected using the procedure debug_find_surface_model.
Default: *30*

'use_3d_edges': Turns the edge-supported matching on or off. This can be used to perform matching without 3D edges, even though the model was created for edge-supported matching. If the model was not created for edge-supported surface-based matching, an error is returned.
List of values: *'true'*, *'false'*
Default: *'true'*

2. Sparse pose refinement In this second step, the approximate poses found in the previous step are further refined. This increases the accuracy of the poses and the significance of the score value.

The following generic parameters control the sparse pose refinement and can be set with GenParamName and GenParamValue:

'sparse_pose_refinement': Enables or disables the sparse pose refinement.
List of values: *'true'*, *'false'*
Default: *'true'*

'pose_ref_use_scene_normals': Enables or disables the usage of scene normals for the pose refinement. If this parameter is enabled, and if the scene contains point normals, then those normals are used to increase the accuracy of the pose refinement. For this, the influence of scene points whose normal points in a different direction than the model normal is decreased. Note that the scene must contain point normals. Otherwise, this parameter is ignored.
List of values: *'true'*, *'false'*
Default: *'false'*

'use_view_based': Turns the view-based score computation for surface-based matching on or off. This can be used to perform matching without using the view-based score, even though the model was prepared for view-based score computation. The influence of *'use_view_based'* on the score is explained in the documentation of Score above.
If the model was not prepared for view-based score computation, an error is returned.
List of values: *'true'*, *'false'*
Default: *'false'*, if *'train_view_based'* was disabled when creating the model, otherwise *'true'*.

3. Dense pose refinement Accurately refines the poses found in the previous steps.

The following generic parameters influence the accuracy and speed of the dense pose refinement and can be set with GenParamName and GenParamValue:

'dense_pose_refinement': Enables or disables the dense pose refinement.
List of values: *'true'*, *'false'*
Default: *'true'*

'pose_ref_num_steps': Number of iterations for the dense pose refinement. Increasing the number of iteration leads to a more accurate pose at the expense of runtime. However, once convergence is reached, the accuracy can no longer be increased, even if the number of steps is increased. Note that this parameter is ignored if the dense pose refinement is disabled.
Suggested values: *1*, *3*, *5*, *20*
Default: *5*
Restriction: *'pose_ref_num_steps'* > *0*

'pose_ref_sub_sampling': Set the rate of scene points to be used for the dense pose refinement. For example, if this value is set to *5*, every 5th point from the scene is used for pose refinement. This parameter allows an easy trade-off between speed and accuracy of the pose refinement: Increasing the value leads to less

points being used and in turn to a faster but less accurate pose refinement. Decreasing the value has the inverse effect. Note that this parameter is ignored if the dense pose refinement is disabled.
Suggested values: *1*, *2*, *5*, *10*
Default: *2*
Restriction: *'pose_ref_sub_sampling' > 0*

'pose_ref_dist_threshold_rel': Set the distance threshold for dense pose refinement relative to the diameter of the surface model. Only scene points that are closer to the object than this distance are used for the optimization. Scene points further away are ignored.
Note that only one of the parameters *'pose_ref_dist_threshold_rel'* and *'pose_ref_dist_threshold_abs'* should be set. If both are set, only the value of the last modified parameter is used. Note that this parameter is ignored if the dense pose refinement is disabled.
Suggested values: *0.03*, *0.05*, *0.1*, *0.2*
Default: *0.1*
Restriction: *'pose_ref_dist_threshold_rel' > 0*

'pose_ref_dist_threshold_abs': Set the distance threshold for dense pose refinement as an absolute value. See *'pose_ref_dist_threshold_rel'* for a detailed description.
Note that only one of the parameters *'pose_ref_dist_threshold_rel'* and *'pose_ref_dist_threshold_abs'* should be set. If both are set, only the value of the modified last parameter is used.
Restriction: *'pose_ref_dist_threshold_abs' > 0*

'pose_ref_scoring_dist_rel': Set the distance threshold for scoring relative to the diameter of the surface model. See the following *'pose_ref_scoring_dist_abs'* for a detailed description.
Note that only one of the parameters *'pose_ref_scoring_dist_rel'* and *'pose_ref_scoring_dist_abs'* should be set. If both are set, only the value of the last modified parameter is used. Note that this parameter is ignored if the dense pose refinement is disabled.
Suggested values: *0.2*, *0.01*, *0.005*, *0.0001*
Default: *0.005*
Restriction: *'pose_ref_scoring_dist_rel' > 0*

'pose_ref_scoring_dist_abs': Set the distance threshold for scoring. Only scene points that are closer to the object than this distance are considered to be 'on the model' when computing the score after the pose refinement. All other scene points are considered not to be on the model. The value should correspond to the amount of noise on the coordinates of the scene points. Note that this parameter is ignored if the dense pose refinement is disabled.
Note that only one of the parameters *'pose_ref_scoring_dist_rel'* and *'pose_ref_scoring_dist_abs'* should be set. If both are set, only the value of the last modified parameter is used.

'pose_ref_use_scene_normals': Enables or disables the usage of scene normals for the pose refinement. This parameter is explained in more details in the section `Sparse pose refinement` above.
List of values: *'true'*, *'false'*
Default: *'false'*

'pose_ref_dist_threshold_edges_rel': Set the distance threshold of edges for dense pose refinement relative to the diameter of the surface model. Only scene edges that are closer to the object edges than this distance are used for the optimization. Scene edges further away are ignored.
Note that only one of the parameters *'pose_ref_dist_threshold_edges_rel'* and *'pose_ref_dist_threshold_edges_abs'* should be set. If both are set, only the value of the last modified parameter is used. Note that this parameter is ignored if the dense pose refinement is disabled or if no edge-supported surface-based matching is used.
Suggested values: *0.03*, *0.05*, *0.1*, *0.2*
Default: *0.1*
Restriction: *'pose_ref_dist_threshold_edges_rel' > 0*

'pose_ref_dist_threshold_edges_abs': Set the distance threshold of edges for dense pose refinement as an absolute value. See *'pose_ref_dist_threshold_edges_rel'* for a detailed description.
Note that only one of the parameters *'pose_ref_dist_threshold_edges_rel'* and *'pose_ref_dist_threshold_edges_abs'* should be set. If both are set, only the value of the last modified parameter is used. Note that this parameter is ignored if the dense pose refinement is disabled or if no edge-supported surface-based matching is used.
Restriction: *'pose_ref_dist_threshold_edges_abs' > 0*

'pose_ref_scoring_dist_edges_rel': Set the distance threshold of edges for scoring relative to the diameter of the surface model. See the following *'pose_ref_scoring_dist_edges_abs'* for a detailed description.
Note that only one of the parameters *'pose_ref_scoring_dist_edges_rel'* and

'pose_ref_scoring_dist_edges_abs' should be set. If both are set, only the value of the last modified parameter is used. Note that this parameter is ignored if the dense pose refinement is disabled or if no edge-supported surface-based matching is used.
Suggested values: *0.2*, *0.01*, *0.005*, *0.0001*
Default: *0.005*
Restriction: *'pose_ref_scoring_dist_edges_rel'* > *0*

'pose_ref_scoring_dist_edges_abs': Set the distance threshold of edges for scoring as an absolute value. Only scene edges that are closer to the object edges than this distance are considered to be 'on the model' when computing the score after the pose refinement. All other scene edges are considered not to be on the model. The value should correspond to the expected inaccuracy of the extracted scene edges and the inaccuracy of the refined pose.
Note that only one of the parameters *'pose_ref_scoring_dist_edges_rel'* and *'pose_ref_scoring_dist_edges_abs'* should be set. If both are set, only the value of the last modified parameter is used. Note that this parameter is ignored if the dense pose refinement is disabled or if no edge-supported surface-based matching is used.
Restriction: *'pose_ref_scoring_dist_edges_abs'* > *0*

'use_view_based': Turns the view-based score computation for surface-based matching on or off. For further details, see the respective description in the section about the sparse pose refinement above.
If the model was not prepared for view-based score computation, an error is returned.
List of values: *'true'*, *'false'*
Default: *'false'*, if *'train_view_based'* was disabled when creating the model, otherwise *'true'*.

'use_self_similar_poses': Turns the optimization regarding self-similar, almost symmetric poses on or off.
If the model was not created with activated parameter *'train_self_similar_poses'*, an error is returned when setting *'use_self_similar_poses'* to *'true'*.
List of values: *'true'*, *'false'*
Default: *'false'*, if *'train_self_similar_poses'* was disabled when creating the model, otherwise *'true'*.

Parameters

▷ **SurfaceModelID** (input_control) . . . . . surface_model ⇝ *handle*
Handle of the surface model.

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the 3D object model containing the scene.

▷ **RelSamplingDistance** (input_control) . . . . . real ⇝ *real*
Scene sampling distance relative to the diameter of the surface model.
Default: 0.05
Suggested values: RelSamplingDistance ∈ {0.1, 0.07, 0.05, 0.04, 0.03}
Restriction: 0 < RelSamplingDistance < 1

▷ **KeyPointFraction** (input_control) . . . . . real ⇝ *real*
Fraction of sampled scene points used as key points.
Default: 0.2
Suggested values: KeyPointFraction ∈ {0.3, 0.2, 0.1, 0.05}
Restriction: 0 < KeyPointFraction <= 1

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real* / integer
Minimum score of the returned poses.
Default: 0
Restriction: MinScore >= 0

▷ **ReturnResultHandle** (input_control) . . . . . string ⇝ *string*
Enable returning a result handle in SurfaceMatchingResultID.
Default: 'false'
Suggested values: ReturnResultHandle ∈ {'true', 'false'}

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'num_matches', 'max_overlap_dist_rel', 'max_overlap_dist_abs', 'sparse_pose_refinement', 'dense_pose_refinement', 'pose_ref_num_steps', 'pose_ref_sub_sampling', 'pose_ref_dist_threshold_rel', 'pose_ref_dist_threshold_abs', 'pose_ref_scoring_dist_rel', 'pose_ref_scoring_dist_abs', 'pose_ref_use_scene_normals', 'scene_normal_computation', 'scene_invert_normals', '3d_edge_min_amplitude_rel', '3d_edge_min_amplitude_abs', 'viewpoint',

'max_gap', '3d_edges', 'pose_ref_dist_threshold_edges_rel', 'pose_ref_dist_threshold_edges_abs', 'pose_ref_scoring_dist_edges_rel', 'pose_ref_scoring_dist_edges_abs', 'use_3d_edges', 'use_view_based', 'use_self_similar_poses'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {0, 1, 'true', 'false', 0.005, 0.01, 0.03, 0.05, 0.1, 'num_scene_points', 'model_point_fraction', 'num_model_points', 'fast', 'mls'}

▷ **Pose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
3D pose of the surface model in the scene.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the surface model.

▷ **SurfaceMatchingResultID** (output_control) . . . . . surface_matching_result(-array) ⇝ *handle*
Handle of the matching result, if enabled in ReturnResultHandle.

Result

find_surface_model returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, get_object_model_3d_params, read_surface_model, create_surface_model, get_surface_model_param, edges_object_model_3d

Possible Successors

refine_surface_model_pose, get_surface_matching_result, clear_surface_matching_result, clear_object_model_3d

Alternatives

refine_surface_model_pose, find_surface_model_image, refine_surface_model_pose_image

See also

refine_surface_model_pose, find_surface_model_image

Module

3D Metrology

find_surface_model_image (Image : : SurfaceModelID, ObjectModel3D, RelSamplingDistance, KeyPointFraction, MinScore, ReturnResultHandle, GenParamName, GenParamValue : Pose, Score, SurfaceMatchingResultID)

Find the best matches of a surface model in a 3D scene and images.

The operator find_surface_model_image finds the best matches of the surface model SurfaceModelID in the scene that is comprised of the 3D surface in ObjectModel3D and the images of the scene in Image. Note that the number of images passed in Image must correspond to the number of cameras set with set_surface_model_param. Note also that the surface model must have been created by create_surface_model with the parameter *'train_3d_edges'* enabled.

The images are used only in the sparse and dense refinement step. For this, the refinement simultaneously optimizes the alignment of the model with the 3D scene as well as the alignment of the reprojected edges of the model silhouette with edges in the passed images. The domain of the images is ignored.

In addition to the parameters documented in `find_surface_model`, `find_surface_model_image` also supports the following generic parameters:

'min_contrast': Sets the minimum contrast of the object in the search images. Edges with a contrast below this threshold are ignored in the refinement.

Suggested values: *5*, *10*, *20*

Default: *10*

Restriction: *'min_contrast'* >= *0*

'max_deformation': Sets the search range in pixels for corresponding edges in the image. This parameter can be used if the shape of the object is slightly deformed compared to the original 3D model used in `create_surface_model`. Note that increasing this parameter can have a significant impact on the runtime of the refinement.

Suggested values: *0*, *1*, *5*

Default: *1*

Restriction: *'max_deformation'* >= *0*

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2
Images of the scene.

▷ **SurfaceModelID** (input_control) ..... surface_model ⇝ *handle*
Handle of the surface model.

▷ **ObjectModel3D** (input_control) ..... object_model_3d ⇝ *handle*
Handle of the 3D object model containing the scene.

▷ **RelSamplingDistance** (input_control) ..... real ⇝ *real*
Scene sampling distance relative to the diameter of the surface model.
Default: 0.05
Suggested values: `RelSamplingDistance` ∈ {0.1, 0.07, 0.05, 0.04, 0.03}
Restriction: 0 < RelSamplingDistance < 1

▷ **KeyPointFraction** (input_control) ..... real ⇝ *real*
Fraction of sampled scene points used as key points.
Default: 0.2
Suggested values: `KeyPointFraction` ∈ {0.3, 0.2, 0.1, 0.05}
Restriction: 0 < KeyPointFraction <= 1

▷ **MinScore** (input_control) ..... real(-array) ⇝ *real* / integer
Minimum score of the returned poses.
Default: 0
Restriction: MinScore >= 0

▷ **ReturnResultHandle** (input_control) ..... string ⇝ *string*
Enable returning a result handle in `SurfaceMatchingResultID`.
Default: 'false'
Suggested values: `ReturnResultHandle` ∈ {'true', 'false'}

▷ **GenParamName** (input_control) ..... attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: `GenParamName` ∈ {'num_matches', 'max_overlap_dist_rel', 'max_overlap_dist_abs', 'sparse_pose_refinement', 'dense_pose_refinement', 'pose_ref_num_steps', 'pose_ref_sub_sampling', 'pose_ref_dist_threshold_rel', 'pose_ref_dist_threshold_abs', 'pose_ref_scoring_dist_rel', 'pose_ref_scoring_dist_abs', 'pose_ref_use_scene_normals', 'scene_normal_computation', 'scene_invert_normals', '3d_edge_min_amplitude_rel', '3d_edge_min_amplitude_abs', 'viewpoint', 'max_gap', '3d_edges', 'max_deformation', 'min_contrast', 'use_3d_edges', 'use_view_based', 'use_self_similar_poses'}

▷ **GenParamValue** (input_control) ..... attribute.value-array ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: `GenParamValue` ∈ {0, 1, 'true', 'false', 0.005, 0.01, 0.03, 0.05, 0.1, 'num_scene_points', 'model_point_fraction', 'num_model_points', 'fast', 'mls'}

▷ **Pose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
3D pose of the surface model in the scene.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the surface model.

▷ **SurfaceMatchingResultID** (output_control) . . . . . surface_matching_result(-array) ⇝ *handle*
Handle of the matching result, if enabled in ReturnResultHandle.

Result

find_surface_model_image returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, get_object_model_3d_params, read_surface_model, create_surface_model, get_surface_model_param, edges_object_model_3d

Possible Successors

refine_surface_model_pose, get_surface_matching_result, clear_surface_matching_result, clear_object_model_3d

Alternatives

refine_surface_model_pose, find_surface_model, refine_surface_model_pose_image

See also

refine_surface_model_pose, find_surface_model

Module

3D Metrology

get_surface_matching_result (: : SurfaceMatchingResultID, ResultName, ResultIndex : ResultValue)

Get details of a result from surface based matching.

The operator get_surface_matching_result returns details about the results of surface based matching or the surface pose refinement. The results are stored in SurfaceMatchingResultID, which must have been created by find_surface_model or refine_surface_model_pose.

The parameter ResultName is used to select which result detail shall be returned. If details about one of the results shall be retrieved, ResultIndex selects the result index, where *0* selects the first result. ResultIndex is ignored for certain values of ResultName.

The following values are possible for ResultName if SurfaceMatchingResultID was created by find_surface_model or find_surface_model_image:

'sampled_scene': A 3D object model handle is returned that contains the sampled scene points that were used in the approximate matching step. This is helpful for tuning the sampling distance for the matching (see parameter RelSamplingDistance of operator find_surface_model). The parameter ResultIndex is ignored.

'key_points': A 3D object model handle is returned that contains all points from the 3D scene that were used as key points in the matching process. This is helpful for tuning the sampling distance and key point rate for the matching (see parameter KeyPointFraction of operator find_surface_model). The parameter ResultIndex is ignored. At least 10 key points should be on the object of interest for stable results.

'score_unrefined': The score of the result before the dense pose refinement is returned. If the sparse pose refinement was disabled, this is the score of the approximate matching. Otherwise the score of the sparse pose refinement is returned. See `find_surface_model` for details about the score. In `ResultIndex` the index of the result must be specified. If `SurfaceMatchingResultID` was created by `refine_surface_model_pose`, 0 is returned.

'sampled_3d_edges': If the surface model was trained with *'train_3d_edges'* enabled, a 3D object model handle is returned that contains the sampled 3D edge points that were used in the approximate matching step and in the sparse refinement step. The parameter `ResultIndex` is ignored.

The following values are always possible for `ResultName`, regardless the operator `SurfaceMatchingResultID` was created with:

'pose': Returns the pose of the matching or refinement result. In `ResultIndex` the index of the result must be specified.

'score_refined': Returns the score of the result after the dense pose refinement. See `find_surface_model` for details about this score. In `ResultIndex` the index of the result must be specified. If `SurfaceMatchingResultID` was created by `find_surface_model` and dense pose refinement was disabled, *0* is returned.

'score': Returns the combined score of the result indexed in `ResultIndex`, thus this parameter is equal to `Score` returned in `find_surface_model`.

'score_surface': Returns the surface-based score of the result indexed in `ResultIndex`. If not specifically set otherwise, this score is equal to *'score_refined'*.

'score_3d_edges': Returns the 3D edge score of the result indexed in `ResultIndex`. This score is only applicable for edged-supported surface-based matching.

'score_2d_edges': Returns the 2D edge score of the result indexed in `ResultIndex`. This score is only applicable for edged-supported surface-based matching.

'score_view_based': Returns the view-based score of the result indexed in `ResultIndex`. This score is only applicable if the surface model supports view-based score computation.

'all_scores': Returns for the result indexed in `ResultIndex` the values of the five scores *'score'*, *'score_surface'*, *'score_3d_edges'*, *'score_2d_edges'*, and *'score_view_based'*. Thereby the scores have the same order as the thresholds given through the parameter `MinScore` in the matching and refinement operators.

Parameters

▷ **`SurfaceMatchingResultID`** (input_control) ..... surface_matching_result ⇝ *handle*
Handle of the surface matching result.

▷ **`ResultName`** (input_control) ..... string(-array) ⇝ *string*
Name of the result property.
Default: 'pose'
List of values: `ResultName` ∈ {'sampled_scene', 'key_points', 'pose', 'score_unrefined', 'score_refined', 'sampled_3d_edges', 'score', 'score_surface', 'score_3d_edges', 'score_2d_edges', 'score_view_based', 'all_scores'}

▷ **`ResultIndex`** (input_control) ..... integer ⇝ *integer*
Index of the matching result, starting with 0.
Default: 0
Suggested values: `ResultIndex` ∈ {0, 1, 2, 3}
Restriction: ResultIndex >= 0

▷ **`ResultValue`** (output_control) ..... integer(-array) ⇝ *integer* / string / real / handle
Value of the result property.

Result

If the handle of the result is valid, the operator `get_surface_matching_result` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

find_surface_model, refine_surface_model_pose

Possible Successors

clear_surface_model

See also

find_surface_model, refine_surface_model_pose, read_surface_model, write_surface_model, clear_surface_model

Module

3D Metrology

get_surface_model_param (: : SurfaceModelID, GenParamName : GenParamValue)

Return the parameters and properties of a surface model.

The operator get_surface_model_param returns parameters and properties of the surface model SurfaceModelID. The surface model must have been created by create_surface_model or read_surface_model. The names of the desired properties are passed in the generic parameter GenParamName, the corresponding values are returned in GenParamValue.

The following values are possible for GenParamName:

'diameter': Diameter of the model point cloud. The diameter is the length of the diagonal of the axis-parallel bounding box (see parameter *'bounding_box1'*).

'center': Center point of the model. The center point is the center of the axis-parallel bounding box (see parameter *'bounding_box1'*).

'bounding_box1': Smallest enclosing axis-parallel cuboid (min_x, min_y, min_z, max_x, max_y, max_z).

'sampled_model': The 3D points sampled from the model for matching. This returns an ObjectModel3D that contains all points sampled from the model surface for matching.

'sampled_pose_refinement': The 3D model points subsampled from the model for the pose refinement. This returns an ObjectModel3D that contains all points sampled from the model surface for pose refinement.

'3d_edges_trained': Returns if the surface model was prepared for edge-supported surface-based matching, i.e., if the parameter *'train_3d_edges'* was enabled in create_surface_model. The returned value is either *'true'* or *'false'*.

'view_based_trained': Returns if the surface model was prepared to support view-based score computation for surface-based matching, i.e., if the parameter *'train_view_based'* was enabled in create_surface_model. The returned value is either *'true'* or *'false'*.

'camera_parameter':

'camera_parameter X': Returns the camera parameters for camera number X, where X is a zero-based index for the cameras. If not given, X defaults zero (first camera). The camera parameters must previously have been set by set_surface_model_param.

'camera_pose':

'camera_pose X': Returns the camera pose for camera number X, where X is a zero-based index for the cameras. If not given, X defaults zero (first camera).

'symmetry_axis_direction':

'symmetry_axis_origin': Returns the symmetry axis or origin, respectively, as set with set_surface_model_param. If no axis is set, an empty tuple is returned.

'symmetry_poses': Returns the symmetry poses as set with set_surface_model_param.

'symmetry_poses_all': Returns all symmetry poses created by set_surface_model_param based on the symmetry poses set with set_surface_model_param.

'pose_restriction_reference_pose': Returns the reference pose as set with `set_surface_model_param`, or an empty tuple if not set.

'pose_restriction_max_angle_diff': Returns the maximum angular difference between the reference pose and found poses, in radians, or an empty tuple if not set.

'pose_restriction_allowed_axis_direction':

'pose_restriction_allowed_axis_origin': Returns the allowed rotation axis and origin, respectively, as set with `set_surface_model_param`. If no axis is set, an empty tuple is returned.

'pose_restriction_filter_final_poses_only': Returns *'true'* if only the final poses are filtered, or *'false'* if the poses are filtered during the matching process (default).

'self_similar_poses_trained': Returns if the surface model was prepared for optimizations regarding self-similar, almost symmetric poses, i.e., if the parameter *'train_self_similar_poses'* was enabled in `create_surface_model`. The returned value is either *'true'* or *'false'*.

'sampled_self_similarity': Returns an ObjectModel3D that contains those 3D points of the model that were sampled for the search of self-similar poses.

'self_similar_poses': Returns the poses under which the object is self-similar, i.e., almost symmetric. If the parameter *'train_self_similar_poses'* was not enabled in `create_surface_model`, an empty tuple is returned.

'self_similar_poses_models': Returns a tuple of ObjectModel3Ds that contains a copy of the original model, transformed into the poses returned by *'self_similar_poses'*. This allows for a visual inspection of the self-similar poses. This parameter is only available if the surface model was created with activated parameter *'train_self_similar_poses'*.

Parameters

▷ **SurfaceModelID** (input_control) . . . . . surface_model ⇝ *handle*
Handle of the surface model.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the parameter.
Default: 'diameter'
List of values: `GenParamName` ∈ {'diameter', 'center', 'bounding_box1', 'sampled_model', 'sampled_pose_refinement', '3d_edges_trained', 'camera_parameter', 'camera_pose', 'symmetry_axis_direction', 'symmetry_axis_origin', 'symmetry_poses', 'symmetry_poses_all', 'pose_restriction_reference_pose', 'pose_restriction_max_angle_diff', 'pose_restriction_allowed_axis_direction', 'pose_restriction_allowed_axis_origin', 'pose_restriction_filter_final_poses_only', 'view_based_trained', 'self_similar_poses_trained', 'sampled_self_similarity', 'self_similar_poses', 'self_similar_poses_models'}

▷ **GenParamValue** (output_control) . . . . . attribute.value(-array) ⇝ *real* / string / integer / handle
Value of the parameter.

Result

`get_surface_model_param` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_surface_model`, `read_surface_model`

Possible Successors

`find_surface_model`, `refine_surface_model_pose`, `write_surface_model`

See also

`create_surface_model`, `set_surface_model_param`

Module

3D Metrology

read_surface_model (: : FileName : SurfaceModelID)

Read a surface model from a file.

The operator read_surface_model reads the surface model, which has been written with write_surface_model, from the file FileName. The handle of the surface model is returned in SurfaceModelID. If no absolute path is given in FileName, the file is searched in the current directory of the HALCON process. The default HALCON file extension for the surface model (SFM) file is 'sfm'. If no file named FileName exists, the default file extension is appended to FileName.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the SFM file.
File extension: .sfm

▷ **SurfaceModelID** (output_control) ..... surface_model ⇝ *handle*
Handle of the read surface model.

Result

read_surface_model returns 2 (H_MSG_TRUE) if all parameters are correct and the file can be read. If the file is not a surface model file, the error 9506 is raised. If the file has a version that can not be read by this version of HALCON, the error 9507 is raised. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d

Possible Successors

find_surface_model, refine_surface_model_pose, get_surface_model_param, clear_surface_model, find_surface_model_image, refine_surface_model_pose_image

Alternatives

create_surface_model

See also

create_surface_model, write_surface_model

Module

3D Metrology

refine_surface_model_pose (: : SurfaceModelID, ObjectModel3D, InitialPose, MinScore, ReturnResultHandle, GenParamName, GenParamValue : Pose, Score, SurfaceMatchingResultID)

Refine the pose of a surface model in a 3D scene.

The operator refine_surface_model_pose refines the approximate pose InitialPose of the surface model SurfaceModelID in the 3D scene ObjectModel3D. The surface model SurfaceModelID must have been created previously with create_surface_model or read_surface_model. Additionally, set_surface_model_param can be used to set certain parameters that influence the refinement, such as restricting the allowed range of rotations.

refine_surface_model_pose is useful if the pose of an object in a scene is approximately known and only needs to be refined. The refined pose is returned in Pose, along with a score in Score. It is possible to pass multiple poses for refinement. Note that, contrary to find_surface_model, the returned poses are not sorted by their score but are returned in the same order as the input poses.

The maximum possible error in the approximate pose that can still be refined depends on the type of object, the amount of clutter in the scene and the visible parts of the objects. In general, differences in the orientation of up to 15° and differences in the position of up to 10% can be refined.

The accuracy of the pose refinement is limited to around 0.1% of the model's size due to numerical reasons. The accuracy further depends on the noise of the scene points, the number of scene points and the shape of the model.

Details about the pose refinement and the parameters are described in the documentation of `find_surface_model` in the section about the dense pose refinement step. The following generic parameters can be set for `refine_surface_model_pose`, and are also documented in `find_surface_model`: *'pose_ref_num_steps'*, *'pose_ref_sub_sampling'*, *'pose_ref_dist_threshold_rel'*, *'pose_ref_dist_threshold_abs'*, *'pose_ref_scoring_dist_rel'*, *'pose_ref_scoring_dist_abs'*, *'pose_ref_use_scene_normals'*, *'3d_edge_min_amplitude_rel'*, *'3d_edge_min_amplitude_abs'*, *'3d_edges'*, *'use_3d_edges'*, *'use_view_based'*, *'use_self_similar_poses'*, *'pose_ref_dist_threshold_edges_rel'*, *'pose_ref_dist_threshold_edges_abs'*, *'pose_ref_scoring_dist_edges_rel'*, and *'pose_ref_scoring_dist_edges_abs'*.

Parameters

- ▷ **SurfaceModelID** (input_control) . . . . . surface_model ⇝ *handle*
  Handle of the surface model.
- ▷ **ObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
  Handle of the 3D object model containing the scene.
- ▷ **InitialPose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
  Initial pose of the surface model in the scene.
- ▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real* / integer
  Minimum score of the returned poses.
  Default: 0
  Restriction: MinScore >= 0
- ▷ **ReturnResultHandle** (input_control) . . . . . string ⇝ *string*
  Enable returning a result handle in `SurfaceMatchingResultID`.
  Default: 'false'
  List of values: `ReturnResultHandle` ∈ {'true', 'false'}
- ▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
  Names of the generic parameters.
  Default: []
  List of values: `GenParamName` ∈ {'pose_ref_num_steps', 'pose_ref_sub_sampling', 'pose_ref_dist_threshold_rel', 'pose_ref_dist_threshold_abs', 'pose_ref_scoring_dist_rel', 'pose_ref_scoring_dist_abs', 'pose_ref_use_scene_normals', '3d_edge_min_amplitude_rel', '3d_edge_min_amplitude_abs', 'viewpoint', '3d_edges', 'use_3d_edges', 'use_view_based', 'use_self_similar_poses'}
- ▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / real / integer
  Values of the generic parameters.
  Default: []
  Suggested values: `GenParamValue` ∈ {0, 1, 'true', 'false', 0.005, 0.01, 0.03, 0.05, 0.1}
- ▷ **Pose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
  3D pose of the surface model in the scene.
- ▷ **Score** (output_control) . . . . . real-array ⇝ *real*
  Score of the found instances of the model.
- ▷ **SurfaceMatchingResultID** (output_control) . . . . . surface_matching_result(-array) ⇝ *handle*
  Handle of the matching result, if enabled in `ReturnResultHandle`.

Result

`refine_surface_model_pose` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, get_object_model_3d_params, read_surface_model, create_surface_model, get_surface_model_param, find_surface_model, edges_object_model_3d

Possible Successors

get_surface_matching_result, clear_surface_matching_result, clear_object_model_3d

Alternatives

find_surface_model, refine_surface_model_pose_image, find_surface_model_image

See also

create_surface_model, find_surface_model, refine_surface_model_pose_image

Module

3D Metrology

refine_surface_model_pose_image (Image : : SurfaceModelID, ObjectModel3D, InitialPose, MinScore, ReturnResultHandle, GenParamName, GenParamValue : Pose, Score, SurfaceMatchingResultID)

Refine the pose of a surface model in a 3D scene and in images.

The operator refine_surface_model_pose_image refines the approximate pose InitialPose of the surface model SurfaceModelID in the 3D scene comprised of the 3D surface in ObjectModel3D and the images of the scene in Image. Note that the number of images passed in Image must correspond to the number of cameras set with set_surface_model_param. Note also that the surface model must have been created by create_surface_model with the parameter *'train_3d_edges'* enabled.

The refinement simultaneously optimizes the alignment of the model with the 3D scene as well as the alignment of the reprojected edges with edges in the passed images. The domain of the images is ignored.

In addition to the parameters documented in refine_surface_model_pose, refine_surface_model_pose_image also supports the generic parameters *'min_contrast'* and *'max_deformation'*, documented in find_surface_model_image.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Images of the scene.

▷ **SurfaceModelID** (input_control) . . . . . surface_model ⇝ *handle*
Handle of the surface model.

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the 3D object model containing the scene.

▷ **InitialPose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
Initial pose of the surface model in the scene.

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real* / integer
Minimum score of the returned poses.
Default: 0
Restriction: MinScore >= 0

▷ **ReturnResultHandle** (input_control) . . . . . string ⇝ *string*
Enable returning a result handle in SurfaceMatchingResultID.
Default: 'false'
List of values: ReturnResultHandle ∈ {'true', 'false'}

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'pose_ref_num_steps', 'pose_ref_sub_sampling', 'pose_ref_dist_threshold_rel', 'pose_ref_dist_threshold_abs', 'pose_ref_scoring_dist_rel',

'pose_ref_scoring_dist_abs', 'pose_ref_use_scene_normals', 'max_deformation', 'min_contrast', '3d_edge_min_amplitude_rel', '3d_edge_min_amplitude_abs', 'viewpoint', '3d_edges', 'use_3d_edges', 'use_view_based', 'use_self_similar_poses'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {0, 1, 'true', 'false', 0.005, 0.01, 0.03, 0.05, 0.1}

▷ **Pose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
3D pose of the surface model in the scene.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the model.

▷ **SurfaceMatchingResultID** (output_control) . . . . . surface_matching_result(-array) ⇝ *handle*
Handle of the matching result, if enabled in ReturnResultHandle.

Result

refine_surface_model_pose_image returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, get_object_model_3d_params, read_surface_model, create_surface_model, get_surface_model_param, find_surface_model, edges_object_model_3d

Possible Successors

get_surface_matching_result, clear_surface_matching_result, clear_object_model_3d

Alternatives

find_surface_model, refine_surface_model_pose, find_surface_model_image

See also

create_surface_model, find_surface_model, refine_surface_model_pose

Module

3D Metrology

serialize_surface_model (
: : SurfaceModelID : SerializedItemHandle)

Serialize a surface_model.

serialize_surface_model serializes the data of a surface model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_surface_model is converted to a serialized item. The surface model is defined by the handle SurfaceModelID. The serialized surface model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_surface_model.

Parameters

▷ **SurfaceModelID** (input_control) . . . . . surface_model ⇝ *handle*
Handle of the surface model.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_surface_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_surface_model`, `create_surface_model`, `get_surface_model_param`

Possible Successors

`clear_surface_model`, `fwrite_serialized_item`, `send_serialized_item`, `deserialize_surface_model`

See also

`create_surface_model`, `read_surface_model`, `write_surface_model`

Module

3D Metrology

`set_surface_model_param` (: : `SurfaceModelID`, `GenParamName`, `GenParamValue` :)

Set parameters and properties of a surface model.

The operator `set_surface_model_param` sets parameters and properties of the surface model `SurfaceModelID`. The surface model must have been created by `create_surface_model` or `read_surface_model`. The names of the desired properties are passed in the generic parameter `GenParamName`, the corresponding values are passed in `GenParamValue`.

The possible values for `GenParamName` are listed below.

- **Defining cameras for image-based refinement.** The following parameters allow to set and clear camera parameters and poses. Those are used by the operators `find_surface_model_image` and `refine_surface_model_pose_image` to project the surface model into the passed image.

  Note that the camera parameters must be set before the camera pose.

  'camera_parameter':

  'camera_parameter X': Sets the camera parameters for camera number X, where X is a zero-based index for the cameras. If not given, X defaults zero (first camera). The camera parameters are used by the operators `find_surface_model_image` and `refine_surface_model_pose_image`, which use the images corresponding to the camera for the 3D pose refinement. Cameras must be added in increasing order.

  'camera_pose':

  'camera_pose X': Sets the camera pose for camera number X, where X is a zero-based index for the cameras. If not given, X defaults zero (first camera). The pose defaults to the zero-pose *[0,0,0,0,0,0,0]* when adding a new camera with *'camera_parameter'*. This usually means that camera and 3D sensor have the same point of origin.

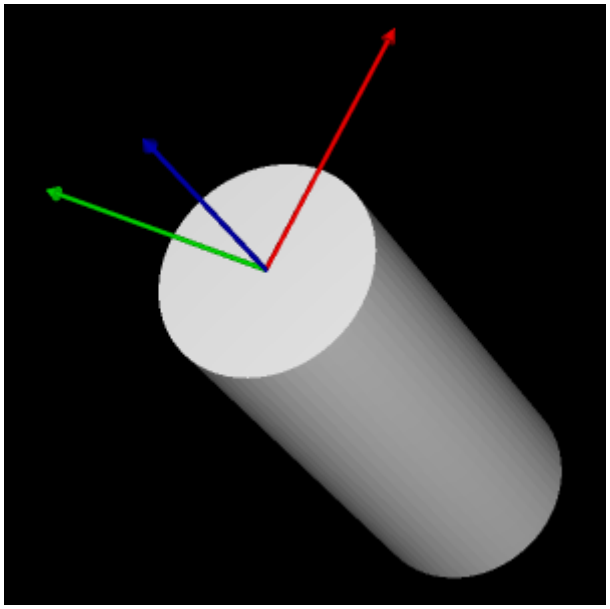
  'clear_cameras': Removes all previously set cameras from the surface model.

- **Defining Object Symmetries.** The following parameters can be used to define symmetries of the 3D object which was used for the creation of the surface model. If the 3D object is symmetric, that information can be used to speed up the surface-based matching. Note that for surface models created with the *'train_3d_edges'* parameter enabled, no symmetries can be set.

  By default, no symmetry is active.

  Note that for performance reasons, when changing the symmetry with any of the parameters below, certain internal data structures of the surface model are re-created, which can take a few seconds.

'symmetry_axis_direction': Set the direction of the symmetry axis of the model. GenParamValue must be a tuple with three numbers, containing the x-, y- and z-value of the axis direction. The model is modified to use this symmetry information for speeding up the matching process.
To remove the symmetry information, pass an empty tuple in GenParamValue. Note that either a symmetry axis or symmetry poses can be set, but not both.

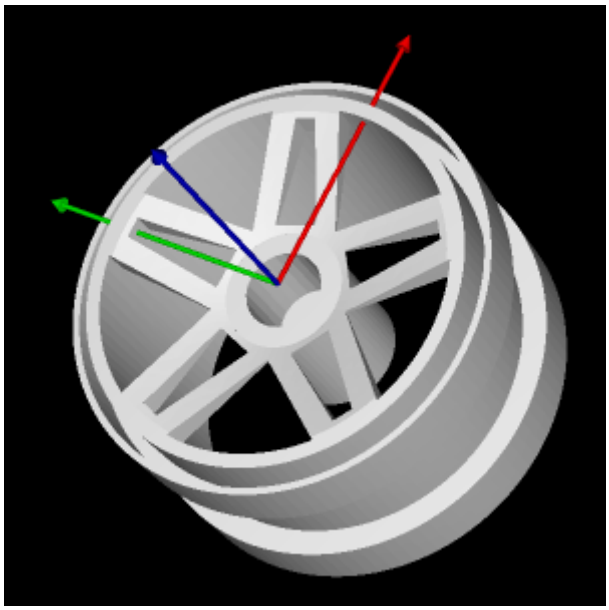


An object (cylinder) with the symmetry axis direction *[0,0,1]*.

In case that *'symmetry_axis_direction'* is used in combination with a restriction of the pose range as described below, the value of *'symmetry_axis_direction'* is also used as if set with the parameter *'pose_restriction_allowed_axis_direction'*.

'symmetry_axis_origin': Set a point on the symmetry axis of the model. GenParamValue must be a tuple with three numbers, which represent a point in model coordinates that lies on the symmetry axis of the model. This parameter is optional and defaults to the center of the model as returned by get_surface_model_param.
In case that *'symmetry_axis_origin'* is used in combination with a restriction of the pose range as described below, the value of *'symmetry_axis_origin'* is also used as if set with the parameter *'pose_restriction_allowed_axis_origin'*.

'symmetry_poses': Set one or more symmetry poses of the model (see create_pose). The model must be identical when transformed with any of those poses. The model is modified to use this symmetry information for speeding up the matching process.
When setting one or more symmetry poses, set_surface_model_param will internally create all poses that can be created by chaining and inverting the passed poses. To obtain all internally created poses, use get_surface_model_param with the argument *'symmetry_poses_all'*. If more than 100 poses are created internally, an error is returned, which indicates that the passed symmetry poses are invalid.
To remove the symmetry poses, pass an empty tuple in GenParamValue. Note that either a symmetry axis or symmetry poses can be set, but not both.



An object with a discontinuous symmetry. The symmetry pose for this object is *[0,0,0, 0,0,360.0/5, 0]*.

- **Restrict the pose range.** The following parameters can be used to restrict the range of rotations in which the surface model is searched for by `find_surface_model`, or the allowed range of rotations for the refinement with `refine_surface_model_pose`.

  By default, no pose range restriction is active.

  Note that for performance reasons, when changing the pose range with any of the parameters below, certain internal data structures of the surface model are re-created, which can take a few seconds.

  'pose_restriction_reference_pose': Set a reference pose of the model. The reference pose can be used along with *'pose_restriction_max_angle_diff'*, to restrict the allowed range of rotations of the model.
  If `GenParamValue` is an empty tuple, any previously set reference pose is cleared and no pose range restriction will be active for the model.
  Otherwise, `GenParamValue` must be a pose (see `create_pose`). Note that the translation part of the pose is ignored. Also note that both *'pose_restriction_reference_pose'* and *'pose_restriction_max_angle_diff'* must be set in order for the pose restriction to be active.

  'pose_restriction_max_angle_diff': Set by how much the rotation of a pose found with `find_surface_model` or refined with `refine_surface_model_pose` may deviate from the rotation set with *'pose_restriction_reference_pose'*, in radians.
  If `GenParamValue` is an empty tuple, any previously set maximum deviation angle is cleared and no pose range restriction will be active for the model.
  Otherwise, `GenParamValue` must be an angle, which indicates by how much the rotations of a detected pose *'P'* and the reference pose *'R'* set with *'pose_restriction_reference_pose'* may differ. The comparison is performed for every model point using the formula $\angle(R\mathbf{v}, P\mathbf{v}) \leq \text{max\_angle\_diff}$, where $\mathbf{v}$ is the 3D point vector.

  'pose_restriction_allowed_axis_direction': Set an axis for which rotations are ignored when evaluating the pose range (see *'pose_restriction_reference_pose'* and *'pose_restriction_max_angle_diff'*). If `GenParamValue` is an empty tuple, any previously set axis is cleared.
  Otherwise, `GenParamValue` must contain a tuple of three numbers which are the direction of the axis in model coordinates.
  If such an axis is set, then a pose is considered to be within the allowed range if the angle between the axis in the reference pose and the compared pose is smaller than the allowed angle, using $\angle(R\,\text{axis}, P\,\text{axis}) \leq \text{max\_angle\_diff}$.

  'pose_restriction_allowed_axis_origin': Set a point on the allowed rotation axis of the model. `GenParamValue` must be a tuple with three numbers, which represent a point in model coordinates that lies on the symmetry axis of the model. This parameter is optional and defaults to the center of the model as returned by `get_surface_model_param`.

  'pose_restriction_filter_final_poses_only': This flag allows to switch between two different modes for the pose range restriction.
  If `GenParamValue` is *'false'* (default), poses outside the defined pose range are removed early in the matching process. Use this setting if the object pose in the scene is always within the de-

fined rotation range, but the object is sometimes found with incorrect rotations. Note that with this setting, find_surface_model might return poses that the algorithm considers to be locally suboptimal, because the locally more optimal poses are outside the allowed pose range. Also note that with this setting, the pose restriction is observed strictly. When passing an input pose to refine_surface_model_pose that is outside the allowed pose range, it will be transformed to be within the allowed pose range.
If GenParamValue is *'true'*, only the final poses are filtered before returning them. This allows removing poses that are valid object poses, but are not needed by the application because, for example, the object cannot be picked up by the robot in a certain orientation. Note that in this setting, less poses than requested might be returned by find_surface_model if one or more of the final poses are outside the allowed pose range.

- **Modifying self-similarities.** The following parameters can be used to adapt the optimization regarding self-similar poses, i.e., poses under which the model is almost symmetric. The parameters can only be set if the parameter *'train_self_similar_poses'* was activated during the call of create_surface_model.
  Note that for performance reasons, when changing the self-similarity search with any of the parameters below, certain internal data structures of the surface model are re-created, which can take a few seconds.

  'self_similar_poses': Set the self-similar poses of the model. Those are poses under which the model is very similar to itself and which can be confused during search.
  find_surface_model will find such poses automatically if the parameter *'use_self_similar_poses'* is activated. The poses can be obtained with get_surface_model_param. If the automatically determined poses are not sufficient to resolve self-similarities, the self-similar poses can be adapted with this parameter. It is usually not recommended to modify this parameter.
  GenParamValue must contain a list of poses. The identity pose will automatically be added to the list of poses, if it is not already contained in it.

Attention

Note that in some cases, if this operator encounters an error condition while modifying the surface model, such as an out-of-memory error, the model might be left in an inconsistent, partly changed state. In such cases, it is recommended to clear the surface model and to no longer use it.

This does not apply to error codes due to invalid parameters, which are checked before performing any model modification.

Also note that setting some of the options requires re-generation of internal data structures and can take as long as the original create_surface_model.

Parameters

▷ **SurfaceModelID** (input_control) . . . . . surface_model ⇝ *handle*
Handle of the surface model.

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Name of the parameter.
Default: 'camera_parameter'
List of values: GenParamName ∈ {'camera_parameter', 'camera_pose', 'clear_cameras', 'symmetry_axis_direction', 'symmetry_axis_origin', 'symmetry_poses', 'pose_restriction_reference_pose', 'pose_restriction_max_angle_diff', 'pose_restriction_allowed_axis_direction', 'pose_restriction_allowed_axis_origin', 'pose_restriction_filter_final_poses_only', 'self_similar_poses'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / string / integer
Value of the parameter.
Suggested values: GenParamValue ∈ {'true', 'false', [], [0,0,0,0,0,0,0], [0,0,1]}

Result

set_surface_model_param returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- SurfaceModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_surface_model`, `read_surface_model`, `get_surface_model_param`

Possible Successors

`find_surface_model`, `refine_surface_model_pose`, `write_surface_model`, `find_surface_model_image`, `refine_surface_model_pose_image`

See also

`create_surface_model`, `get_surface_model_param`

Module

3D Metrology

write_surface_model (: : SurfaceModelID, FileName :)

Write a surface model to a file.

The operator `write_surface_model` writes a surface model to the file `FileName`. The file can be read again with `read_surface_model`. The default HALCON file extension for the surface model (SFM) file is 'sfm'.

Parameters

▷ **SurfaceModelID** (input_control) . . . . . surface_model ⇝ *handle*
Handle of the surface model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: `.sfm`

Result

`write_surface_model` returns `2` (`H_MSG_TRUE`) if all parameters are correct and the HALCON process has write permission to the file. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_surface_model`, `create_surface_model`, `get_surface_model_param`

Possible Successors

`clear_surface_model`

See also

`create_surface_model`, `read_surface_model`

Module

3D Metrology

Chapter 4

3D Object Model

4.1 Creation

clear_object_model_3d (: : ObjectModel3D :)

Free the memory of a 3D object model.

The operator clear_object_model_3d frees the memory of a 3D object model that was previously created. After calling clear_object_model_3d, the model can no longer be used. The handle ObjectModel3D becomes invalid.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

Result

If the handle of the model is valid, the operator clear_object_model_3d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ObjectModel3D

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d

Module

3D Metrology

copy_object_model_3d (: : ObjectModel3D,
Attributes : CopiedObjectModel3D)

Copy a 3D object model.

A 3D object model consists of a set of attributes. The operator copy_object_model_3d creates a new 3D object model and copies the selected attributes of the input 3D object model to this new output 3D object

model. The input 3D object model is defined by a handle ObjectModel3D. The operator returns the handle CopiedObjectModel3D of the new 3D object model. The operator can be used to save memory space by removing not needed attributes. Access to the attributes of the 3D object model is possible, e.g., with the operator get_object_model_3d_params.

The parameter Attributes determines which attributes should be copied. In addition, attributes can be excluded from copying by using the prefix ~. In order to remove attributes from a 3D object model, the operator remove_object_model_3d_attrib can be used instead.

Note that because a 3D object model itself consists of a set of attributes, even the point coordinates are an attribute of the model. This means, that at least this one attribute must be selected for copy_object_model_3d else the object model to be copied would be empty. So if only a 3D object model representing a point cloud shall be copied without further attributes, Attributes must be set to *'point_coord'*. If an attribute to be copied is not available or no attribute is selected, an exception is raised.

The following values for the parameter Attributes are possible:

'point_coord': This value specifies that the attribute with the 3D point coordinates is copied.

'point_normal': This value specifies that the attribute with the 3D point normals and the attribute with the 3D point coordinates are copied.

'triangles': This value specifies that the attribute with the face triangles and attribute with the 3D point coordinates are copied.

'polygons': This value specifies that the attribute with the face polygons and the attribute with the 3D point coordinates are copied.

'lines': This value specifies that the attribute with the lines and the attribute with the 3D point coordinates are copied.

'xyz_mapping': This value specifies that the attribute with the mapping to image coordinates and the attribute with the 3D point coordinates are copied.

'extended_attribute': This value specifies that all extended attributes are copied. If it is necessary to copy further attributes that are related to the extended attributes, these attributes are copied, too. These further attributes could be, e.g., 3D point coordinates, face triangles, face polygons, or lines.

'primitives_all': This value specifies that the attribute with the parameters of the primitive (including an empty primitive) is copied (e.g., obtained from the operator fit_primitives_object_model_3d).

'primitive_plane': This value specifies that the attribute with the primitive plane is copied (e.g., obtained from the operator fit_primitives_object_model_3d).

'primitive_sphere': This value specifies that the attribute with the primitive sphere is copied (e.g., obtained from the operator fit_primitives_object_model_3d).

'primitive_cylinder': This value specifies that the attribute with the primitive cylinder is copied (e.g., obtained from the operator fit_primitives_object_model_3d).

'primitive_box': This value specifies that the attribute with the primitive cylinder is copied.

'shape_based_matching_3d_data': This value specifies that the attribute with the prepared shape model for shape-based 3D matching is copied.

'distance_computation_data': This value specifies that the attribute with the distance computation data structure is copied. The distance computation data can be created with prepare_object_model_3d, and can be used with distance_object_model_3d. If this attribute is selected, then the corresponding target data attribute of the distance computation is copied as well. For example, if the distance computation was prepared for triangles, the triangles and the vertices are copied.

'surface_based_matching_data': This value specifies that the data for surface based matching are copied. The attributes with the 3D point coordinates and the attribute with the point normals are copied. If the attribute with point normals is not available, the attribute with the mapping from the 3D point coordinates to the image coordinates is copied. If the attribute with the mapping from the 3D point coordinates to the image coordinates is not available, the attribute with the face triangles is copied. If the attribute with face triangles is not available, too, the attribute with the face polygons is copied. If none of these attributes is available, an exception is raised.

'segmentation_data': This value specifies that the data for a 3D segmentation is copied. The attributes with the 3D point coordinates and the attribute with the face triangles are copied. If the attribute with the face triangles is not available, the attribute with the mapping from the 3D point coordinates to the image coordinates is copied. If none of these attributes is available, an exception is raised.

'score': This value specifies that the attribute with the scores and the attribute with the 3D point coordinates are copied. Scores may be obtained from the operator reconstruct_surface_stereo.

'red': This value specifies that the attribute containing the red color and the attribute with the 3D point coordinates are copied.

'green': This value specifies that the attribute containing the green color and the attribute with the 3D point coordinates are copied.

'blue': This value specifies that the attribute containing the blue color and the attribute with the 3D point coordinates are copied.

'original_point_indices': This value specifies that the attribute with the original point indices and the attribute with the 3D point coordinates are copied. Original point indices may be obtained from the operator triangulate_object_model_3d.

'all': This value specifies that all available attributes are copied. That is, the attributes are the point coordinates, the point normals, the face triangles, the face polygons, the mapping to image coordinates, the shape model for matching, the parameter of a primitive, and the extended attributes.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the input 3D object model.

▷ **Attributes** (input_control) . . . . . string(-array) ⇝ *string* / real / integer
Attributes to be copied.
Default: 'all'
List of values: Attributes ∈ {'point_coord', 'point_normal', 'triangles', 'polygons', 'xyz_mapping', 'extended_attribute', 'shape_based_matching_3d_data', 'primitives_all', 'primitive_plane', 'primitive_sphere', 'primitive_cylinder', 'primitive_box', 'surface_based_matching_data', 'segmentation_data', 'distance_computation_data', 'score', 'red', 'green', 'blue', 'all', 'original_point_indices'}

▷ **CopiedObjectModel3D** (output_control) . . . . . object_model_3d ⇝ *handle*
Handle of the copied 3D object model.

Result

copy_object_model_3d returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d

Possible Successors

get_object_model_3d_params

See also

remove_object_model_3d_attrib, set_object_model_3d_attrib

Module

3D Metrology

deserialize_object_model_3d (
: : SerializedItemHandle : ObjectModel3D)

Deserialize a serialized 3D object model.

deserialize_object_model_3d deserializes a 3D object model that was serialized by serialize_object_model_3d (see fwrite_serialized_item for an introduction of the basic

principle of serialization). The serialized 3D object model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created 3D object model with the handle ObjectModel3D.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **ObjectModel3D** (output_control) ..... object_model_3d ⇝ *handle*
Handle of the 3D object model.

Result

If the parameters are valid, the operator deserialize_object_model_3d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

write_object_model_3d, fread_serialized_item, receive_serialized_item, serialize_object_model_3d

Possible Successors

affine_trans_object_model_3d, object_model_3d_to_xyz, prepare_object_model_3d

Alternatives

xyz_to_object_model_3d

See also

write_object_model_3d, clear_object_model_3d

Module

3D Metrology

gen_box_object_model_3d (: : Pose, LengthX, LengthY, LengthZ : ObjectModel3D)

Create a 3D object model that represents a box.

gen_box_object_model_3d creates a box-shaped 3D primitive, i.e., a 3D object model that represents a box. The box is specified by a Pose and the side lengths LengthX, LengthY, and LengthZ along the respective axis of the pose. The handle of the resulting 3D object model is returned by parameter ObjectModel3D.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

▷ **Pose** (input_control) ..... pose(-array) ⇝ *real* / integer
The pose that describes the position and orientation of the box. The pose has its origin in the center of the box.

▷ **LengthX** (input_control) ..... number(-array) ⇝ *real*
The length of the box along the x-axis.

▷ **LengthY** (input_control) ..... number(-array) ⇝ *real*
The length of the box along the y-axis.
Number of elements: LengthY == LengthX

▷ **LengthZ** (input_control) ..... number(-array) ⇝ *real*
The length of the box along the z-axis.
Number of elements: LengthZ == LengthX

▷ **ObjectModel3D** (output_control) ..... object_model_3d(-array) ⇝ *handle*
Handle of the resulting 3D object model.

Result

`gen_box_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`smallest_bounding_box_object_model_3d`

Possible Successors

`get_object_model_3d_params`, `sample_object_model_3d`, `clear_object_model_3d`

See also

`gen_cylinder_object_model_3d`, `gen_sphere_object_model_3d`, `gen_sphere_object_model_3d_center`, `gen_plane_object_model_3d`

Module

3D Metrology

gen_cylinder_object_model_3d (: : Pose, Radius, MinExtent, MaxExtent : ObjectModel3D)

Create a 3D object model that represents a cylinder.

`gen_cylinder_object_model_3d` creates a cylinder-shaped 3D primitive, i.e., a 3D object model that represents a cylinder. A cylinder is described by its center and the direction of its axis in `Pose` and by its radius in `Radius`. The pose has the origin on the rotation axis of the cylinder and is oriented such that the z-axis is aligned with the main direction of the cylinder. Additionally, the extensions of the cylinder are given by `MinExtent` and `MaxExtent`. `MinExtent` and `MaxExtent` represent the z-coordinates of the lowest and highest points of the cylinder on the rotation axis. The handle of the 3D object model is returned by the parameter `ObjectModel3D`.

Parameters

▷ **Pose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
The pose that describes the position and orientation of the cylinder.

▷ **Radius** (input_control) . . . . . number(-array) ⇝ *real*
The radius of the cylinder.

▷ **MinExtent** (input_control) . . . . . number(-array) ⇝ *real*
Lowest z-coordinate of the cylinder in the direction of the rotation axis.

▷ **MaxExtent** (input_control) . . . . . number(-array) ⇝ *real*
Highest z-coordinate of the cylinder in the direction of the rotation axis.
Restriction: MinExtent < MaxExtent

▷ **ObjectModel3D** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the resulting 3D object model.

Result

`gen_cylinder_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

get_object_model_3d_params, sample_object_model_3d, clear_object_model_3d

See also

gen_sphere_object_model_3d, gen_sphere_object_model_3d_center, gen_plane_object_model_3d, gen_box_object_model_3d

Module

3D Metrology

gen_empty_object_model_3d (: : : EmptyObjectModel3D)

Create an empty 3D object model.

gen_empty_object_model_3d creates an empty 3D object model. The handle of the 3D object model is returned by the parameter EmptyObjectModel3D. Attributes can be added using the operators set_object_model_3d_attrib or set_object_model_3d_attrib_mod.

Parameters

▷ **EmptyObjectModel3D** (output_control) ..... object_model_3d ⇝ *handle*
Handle of the new 3D object model.

Result

gen_empty_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_object_model_3d_attrib, set_object_model_3d_attrib_mod

See also

gen_box_object_model_3d, gen_cylinder_object_model_3d, gen_sphere_object_model_3d, gen_sphere_object_model_3d_center, gen_plane_object_model_3d

Module

3D Metrology

gen_object_model_3d_from_points (: : X, Y, Z : ObjectModel3D)

Create a 3D object model that represents a point cloud from a set of 3D points.

gen_object_model_3d_from_points creates a 3D object model that represents a point cloud. The points are described by x-, y-, and z-coordinates in the parameters X, Y, and Z.

Parameters

▷ **X** (input_control) ..... point3d.x(-array) ⇝ *real*
The x-coordinates of the points in the 3D point cloud.

▷ **Y** (input_control) ..... point3d.y(-array) ⇝ *real*
The y-coordinates of the points in the 3D point cloud.

▷ **Z** (input_control) . . . . . point3d.z(-array) ⇝ *real*
The z-coordinates of the points in the 3D point cloud.

▷ **ObjectModel3D** (output_control) . . . . . object_model_3d ⇝ *handle*
Handle of the resulting 3D object model.

Result

`gen_object_model_3d_from_points` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`get_object_model_3d_params`

Possible Successors

`connection_object_model_3d`, `convex_hull_object_model_3d`

Alternatives

`xyz_to_object_model_3d`

See also

`gen_box_object_model_3d`, `gen_sphere_object_model_3d`, `gen_cylinder_object_model_3d`

Module

3D Metrology

gen_plane_object_model_3d (: : Pose, XExtent, YExtent : ObjectModel3D)

Create a 3D object model that represents a plane.

`gen_plane_object_model_3d` creates a planar 3D primitive, i.e., a 3D object model that represents a plane. The plane is described by its center and rotation. The normal vector of the plane is aligned to the z-axis of the rotated coordinate system. The center and the rotation is set with the parameter `Pose`. Additionally, the plane can be limited by a polygon, that is defined by points with the coordinates `XExtent` and `YExtent`. The handle of the 3D object model is returned by the parameter `ObjectModel3D`.

Parameters

▷ **Pose** (input_control) . . . . . pose ⇝ *real* / integer
The center and the rotation of the plane.
Number of elements: Pose == 7

▷ **XExtent** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
x coordinates specifying the extent of the plane.

▷ **YExtent** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
y coordinates specifying the extent of the plane.
Number of elements: XExtent == YExtent

▷ **ObjectModel3D** (output_control) . . . . . object_model_3d ⇝ *handle*
Handle of the resulting 3D object model.

Result

`gen_plane_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

get_object_model_3d_params, sample_object_model_3d, clear_object_model_3d

See also

gen_cylinder_object_model_3d, gen_sphere_object_model_3d, gen_sphere_object_model_3d_center, gen_box_object_model_3d

Module

3D Metrology

gen_sphere_object_model_3d (: : Pose, Radius : ObjectModel3D)

Create a 3D object model that represents a sphere.

gen_sphere_object_model_3d creates a sphere-shaped 3D primitive, i.e., a 3D object model that represents a sphere. A sphere is defined by its center given in Pose and its radius given in Radius. The handle of the 3D object model is returned by the parameter ObjectModel3D.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

▷ **Pose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
The pose that describes the position of the sphere.

▷ **Radius** (input_control) . . . . . number(-array) ⇝ *real*
The radius of the sphere.

▷ **ObjectModel3D** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the resulting 3D object model.

Result

gen_sphere_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

smallest_sphere_object_model_3d

Possible Successors

get_object_model_3d_params, sample_object_model_3d, clear_object_model_3d

Alternatives

gen_sphere_object_model_3d_center

See also

gen_cylinder_object_model_3d, gen_plane_object_model_3d, gen_box_object_model_3d

Module

3D Metrology

gen_sphere_object_model_3d_center (: : X, Y, Z, Radius : ObjectModel3D)

Create a 3D object model that represents a sphere from x,y,z coordinates.

gen_sphere_object_model_3d_center creates a sphere-shaped 3D primitive, i.e., a 3D object model that represents a sphere. A sphere is defined by its center given in X, Y, and Z, and its radius given in Radius. The handle of the 3D object model is returned by the parameter ObjectModel3D.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

- ▷ **X** (input_control) ..... point3d.x(-array) ⇝ *real* / integer
  The x-coordinate of the center point of the sphere.
- ▷ **Y** (input_control) ..... point3d.y(-array) ⇝ *real* / integer
  The y-coordinate of the center point of the sphere.
- ▷ **Z** (input_control) ..... point3d.z(-array) ⇝ *real* / integer
  The z-coordinate of the center point of the sphere.
- ▷ **Radius** (input_control) ..... number(-array) ⇝ *real*
  The radius of the sphere.
- ▷ **ObjectModel3D** (output_control) ..... object_model_3d(-array) ⇝ *handle*
  Handle of the resulting 3D object model.

Result

gen_sphere_object_model_3d_center returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

smallest_sphere_object_model_3d

Possible Successors

get_object_model_3d_params, sample_object_model_3d, clear_object_model_3d

Alternatives

gen_sphere_object_model_3d

See also

gen_cylinder_object_model_3d, gen_plane_object_model_3d, gen_box_object_model_3d

Module

3D Metrology

read_object_model_3d (: : FileName, Scale, GenParamName, GenParamValue : ObjectModel3D, Status)

Read a 3D object model from a file.

The operator read_object_model_3d reads a 3D object model from the file FileName and returns a 3D object model handle in ObjectModel3D.

The operator supports the following file formats:

'om3': HALCON format for 3D object model. Files with this format can be written by `write_object_model_3d`. The default file extension for this format is 'om3'.

'dxf': AUTOCAD format. HALCON supports only the ASCII version of the format. See below for details about reading this file format. The default file extension for this format is 'dxf'.

'off': Object File Format. This is a simple ASCII-based format that can hold 3D points and polygons. The binary OFF format is not supported. The default file extension for this format is 'off'.

'ply': Polygon File Format (also Stanford Triangle Format). This is a simple format that can hold 3D points, point normals, polygons, color information and point-based extended attributes. HALCON supports both the ASCII and the binary version of the format. If the file to be read contains unsupported information, the additional data is ignored and only the supported data is read. If the name of a `property` entry of a 'ply' file coincides with the name of a standard attribute (see `set_object_model_3d_attrib`), the property will preferably be read into the standard attribute. The default file extension for this format is 'ply'.

'obj': OBJ file format, also 'Wavefront OBJ-Format'. This is an ASCII-based format that can hold 3D points, polygons, normals, texture coordinates, materials and other information. HALCON supports points ('v'-lines), point normals ('vn'-lines) and polygonal faces ('f'-lines). Existing point normals are only returned if there are exactly as many point normals as there are points. Other entities are ignored. The default file extension for this format is 'obj'.

'stl',

'stl_binary',

'stl_ascii': STL file format, also 'Stereolithography format', 'SurfaceTesselationLanguage ', 'StandardTriangulationLanguage', and 'StandardTesselationLanguage'. This format stores triangles and triangle normals. However, as triangle normals are not supported by HALCON 3D object models, only triangles are read while the triangle normals are ignored. Normals are recomputed from the triangles if required. HALCON reads both the ASCII and the binary version of this format. If *'stl'* is set, HALCON will auto-detect the type. Setting the type to *'stl_binary'* or *'stl_ascii'* will enforce the corresponding format. The default file extension for this format is 'stl'.

'step': STEP file format, also STP or 'Standard for the Exchange of Product Model Data'. This is a complex format that stores a large variety of geometrical definitions which allows an accurate storage of 3D models. Due to the limited support for the geometrical structures defined by STEP in HALCON 3D object models, triangulation is performed on these geometries, resulting in models comprised of triangle meshes. The default file extensions for this format are 'step' and 'stp'.

'generic_ascii': This format can be used to read different ASCII files containing 3D data in tabular form, e.g. 'ptx', 'pts', 'xyz' or 'pcd'. Currently, only point based attributes are supported, no triangles or polygons. The information for each 3D point is expected to be written in a single line, one point at a time. The file format must be further specified by setting the generic parameter *'ascii_format'*.

When reading a DXF file, the output parameter `Status` contains information about the number of 3D faces that were read and, if necessary, warnings that parts of the DXF file could not be interpreted.

The parameter `Scale` defines the scale of the file. For example, if the parameter is set to *'mm'*, all units in the file are assumed to have the unit *'mm'* and are transformed into the usual HALCON-internal unit *'m'* by multiplication with *0.001*. A value of *'100 mm'* thus becomes *'0.1 m'*. Alternatively, a scaling factor can be passed to `Scale`, which is multiplied with all coordinate values found in the file. The relation of units to scaling factors is given in the following table:

Unit	Scaling factor
m	1
dm	0.1
cm	0.01
mm	0.001
um, microns	10^{-6}
nm	10^{-9}
km	1000
in	0.0254
ft	0.3048
yd	0.9144

Note that the parameter Scale is ignored for files of type *'om3'* and *'step'*. om3-files are always read without any scale changes. For step-files, the unit is directly defined in the files, read along with the stored data and used to scale to the HALCON-internal unit *'m'*. For changing the scale manually after reading a 3D object model, use affine_trans_object_model_3d.

A set of additional optional parameters can be set. The names and values of the parameters are passed in GenParamName and GenParamValue, respectively. Some of the optional parameters can only be set for a certain file type. The following values for GenParamName are possible:

'file_type': Forces a file type. If this parameter is not set, the operator read_object_model_3d tries to auto-detect the file type using the file ending and the file header. If the parameter is set, the given file is interpreted as this file format.

List of values: *'om3'*, *'dxf'*, *'off'*, *'ply'*, *'obj'*, *'stl'*, *'step'*, *'generic_ascii'*.

'convert_to_triangles': Convert all faces to triangles. If this parameter is set to *'true'*, all faces read from the file are converted to triangles.

Valid for formats: 'dxf', 'ply', 'off', 'obj'.

List of values: *'true'*, *'false'*.

Default: *'false'*.

'invert_normals': Invert normals and face orientations. If this parameter is set to *'true'*, the orientation of all normals and faces is inverted.

Valid for formats: 'dxf', 'ply', 'off', 'obj', 'stl', 'step', 'generic_ascii'.

List of values: *'true'*, *'false'*.

Default: *'false'*.

'max_approx_error', *'min_num_points'*: DXF-specific parameters (see below).

Valid for formats: 'dxf'.

'max_surface_deviation': STEP-specific parameter.

Specifies the maximum allowed deviation (in *'m'*) from the model surface during the triangulation. A smaller value will generate a more accurate model but will also increase the reading time and the number of points and triangles in the resulting model. Set the parameter to *'auto'* in order to estimate it automatically depending on the size of the model.

Valid for formats: 'step'.

Suggested values: *'auto'*, *0.0001*, *0.00001*.

Default: *'auto'*.

Restriction: *'max_surface_deviation' > 0*

'split_level': STEP-specific parameter.

STEP files can contain definitions of independent model components. With this parameter, each component can be imported as a HALCON 3D object model. If the parameter is set to *0*, the file is imported as a single model. With *1* the model components are roughly separated from each other, while *2* separates the model components at a more detailed level.

Valid for formats: 'step'.

List of values: *0*, *1*, *2*.

Default: *0*.

'ascii_format': generic_ascii-specific parameter.

Specifies the format of the ASCII file to be read. As value, a dict containing information about the file content must be provided. The dict defines the columns to be read and meta-data like the first line number containing point information. Examples are given at the bottom of the operator reference or in the HDevelop example read_object_model_3d_generic_ascii.hdev. The following parameters can be set as dict keys:

'columns': **(mandatory)** Defines the column attributes in the read file, given as a tuple of strings. All point-related standard and extended attributes as listed in the reference of set_object_model_3d_attrib are supported. At least, *'point_coord_x'*, *'point_coord_y'* and *'point_coord_z'* must be set. When setting normals, all three components *'point_normal_x'*, *'point_normal_y'* and *'point_normal_z'* must be set. Ignoring columns is possible by setting *''* at the according tuple position.

Suggested values: *['point_coord_x', 'point_coord_y', 'point_coord_z']*, *['point_normal_x', 'point_normal_y', 'point_normal_z']*, *'red'*, *'green'*, *'blue'*, *'&my_custom_attrib'*, *''*.

'separator': **(mandatory)** Defines the separator between the columns. Currently, whitespace (blanks or tabs) and semicolon are supported.
List of values: *' '*, *';'*.

'first_point_line': **(optional)** Describes the number of the first line to be read from the file and can e.g. be used to skip header information. The top line in the file corresponds to *'first_point_line' 1*.
Default: *1*.
Restriction: *'first_point_line'* > *0*

'last_point_line': **(optional)** Describes the number of the last line to be read from the file and can e.g. be used to skip unsupported information. The top line in the file corresponds to *'last_point_line' 1*. When *'last_point_line'* is set to *-1*, all lines are read.
Default: *-1*.
Restriction: *'last_point_line'* >= -1

'comment': **(optional)** Describes the start of comments in the read file. Information behind the comment start are ignored when reading the file.
Suggested values: *'#'*, *'*'*, *'/'*, *'comment'*.

Valid for formats: 'generic_ascii'.

'xyz_map_width': Creates for the read 3D object model a mapping that assigns an image coordinate to each read 3D point, as in `xyz_to_object_model_3d`. It is assumed that the read file contains the 3D points row-wise. The passed value is used as width of the image. The height of the image is computed automatically. If this parameter is set, the read 3D object model can be projected by `object_model_3d_to_xyz` using the method *'from_xyz_map'*. Only one of the two parameters *'xyz_map_width'* and *'xyz_map_height'* can be set.

Valid for formats: 'ply', *'off'*, *'obj'*, *'generic_ascii'*.

Default: *-1*.

Restriction: *'xyz_map_width'* > 0

'xyz_map_height': As *'xyz_map_width'*, but assuming that the 3D points are aligned column-wise. The width of the image is computed automatically. Only one of the two parameters *'xyz_map_width'* and *'xyz_map_height'* can be set.

Valid for formats: 'ply', *'off'*, *'obj'*, *'generic_ascii'*.

Default: *-1*.

Restriction: *'xyz_map_height'* > 0

Note that in many cases, it is recommended to use the 2D mapping data, if available, for speed and robustness reasons. This is beneficial especially when using `sample_object_model_3d`, `surface_normals_object_model_3d`, or when preparing a 3D object model for surface-based matching, e.g., smoothing, removing outliers, and reducing the domain.

The operator `read_object_model_3d` supports the following DXF entities:

- POLYLINE
  - Polyface meshes (Polyline flag 64)
  - 3D Polylines (Polyline flag 8,9)
  - 2D Polylines (Polyline flag 0)
- LWPOLYLINE
  - 2D Polylines
- 3DFACE
- LINE
- CIRCLE
- ARC
- SOLID
- BLOCK
- INSERT

The two-dimensional linear DXF entities LINE, CIRCLE and ARC are not interpreted as faces. Only if these elements are extruded, the resulting faces are inserted in the 3D object model. All elements that represent no faces but lines are added as 3D lines to the 3D object model.

The curved surface of extruded DXF entities of the type CIRCLE and ARC is approximated by planar faces. The accuracy of this approximation can be controlled with the two generic parameters *'min_num_points'* and *'max_approx_error'*. The parameter *'min_num_points'* defines the minimum number of sampling points that are used for the approximation of the DXF element CIRCLE or ARC. Note that the parameter *'min_num_points'* always refers to the full circle, even for ARCs, i.e., if *'min_num_points'* is set to *50* and a DXF entity of the type ARC is read that represents a semi-circle, this semi-circle is approximated by at least 25 sampling points. The parameter *'max_approx_error'* defines the maximum deviation of the XLD contour from the ideal circle. The determination of this deviation is carried out in the units used in the DXF file. For the determination of the accuracy of the approximation both criteria are evaluated. Then, the criterion that leads to the more accurate approximation is used.

Internally, the following default values are used for the generic parameters:

- *'min_num_points' = 20*
- *'max_approx_error' = 0.25*

To achieve a more accurate approximation, either the value for *'min_num_points'* must be increased or the value for *'max_approx_error'* must be decreased.

One possible way to create a suitable DXF file is to create a 3D model of the object with the CAD program AutoCAD. Ensure that the surface of the object is modeled, not only its edges. Lines that, e.g., define object edges, will not be used by HALCON, because they do not define the surface of the object. Once the modeling is completed, you can store the model in DWG format. To convert the DWG file into a DXF file that is suitable for HALCON's 3D matching, carry out the following steps:

- Export the 3D CAD model to a 3DS file using the 3dsout command of AutoCAD. This will triangulate the object's surface, i.e., the model will only consist of planes. (Users of AutoCAD 2007 or newer versions can download this command utility from Autodesk's web site.)
- Open a new empty sheet in AutoCAD.
- Import the 3DS file into this empty sheet with the 3dsin command of AutoCAD.
- Save the object into a DXF R12 file.

Users of other CAD programs should ensure that the surface of the 3D model is triangulated before it is exported into the DXF file. If the CAD program is not able to carry out the triangulation, it is often possible to save the 3D model in the proprietary format of the CAD program and to convert it into a suitable DXF file by using a CAD file format converter that is able to perform the triangulation.

Parameters

▷ **`FileName`** (input_control) . . . . . filename.read ⇝ *string*
Filename of the file to be read.
Default: 'mvtec_bunny_normals'
Suggested values: `FileName` ∈ {'mvtec_bunny', 'glass_mug', 'bmc_mini', 'pipe_joint', 'clamp_sloped', 'tile_spacer', 'engine_part_bearing'}
File extension: `.off`, `.ply`, `.dxf`, `.om3`, `.obj`, `.stl`, `.step`, `.stp`

▷ **`Scale`** (input_control) . . . . . number ⇝ *string* / real / integer
Scale of the data in the file.
Default: 'm'
Suggested values: `Scale` ∈ {'m', 'cm', 'mm', 'microns', 'um', 'nm', 'km', 'in', 'ft', 'yd', 1.0, 0.01, 0.001, 1.0e-6, 0.0254, 0.3048, 0.9144}

▷ **`GenParamName`** (input_control) . . . . . string(-array) ⇝ *string*
Names of the generic parameters.
Default: []
List of values: `GenParamName` ∈ {'ascii_format', 'convert_to_triangles', 'invert_normals', 'file_type', 'min_num_points', 'max_approx_error', 'max_surface_deviation', 'split_level', 'xyz_map_width', 'xyz_map_height'}

▷ **GenParamValue** (input_control) ..... string(-array) ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false', 1, 0, 'auto', 'om3', 'off', 'ply', 'dxf', 'obj', 'stl', 'stl_binary', 'stl_ascii', 'step', 'generic_ascii'}

▷ **ObjectModel3D** (output_control) ..... object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **Status** (output_control) ..... string(-array) ⇝ *string*
Status information.

Example

```
* Example how to use file_type generic_ascii and generic parameter ascii_format to read
FileFormat := dict{}
FileFormat.separator := ' '
FileFormat.columns := ['point_coord_x', 'point_coord_y', 'point_coord_z', 'point_normal_
FileFormat.first_point_line := 14
FileFormat.last_point_line := 2273
FileFormat.comment := 'comment'
read_object_model_3d ('glass_mug.ply', 'm', ['file_type', 'ascii_format'], ['generic_asc
```

Result

The operator read_object_model_3d returns the value 2 (H_MSG_TRUE) if the given parameters are correct, the file can be read, and the file is valid. If the file format is unknown or cannot be determined, the error 9512 is raised. If the file is invalid, the error 9510 is raised. If necessary, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

write_object_model_3d

Possible Successors

affine_trans_object_model_3d, object_model_3d_to_xyz, prepare_object_model_3d

Alternatives

xyz_to_object_model_3d

See also

write_object_model_3d, clear_object_model_3d

Module

3D Metrology

remove_object_model_3d_attrib (: : ObjectModel3D,
Attributes : ObjectModel3DOut)

Remove attributes of a 3D object model.

remove_object_model_3d_attrib copies the 3d object model ObjectModel3D and removes within this copy the standard and/or extended attributes given in Attributes. The new 3d object model is returned in ObjectModel3DOut. Doing so does not modify the 3d object model ObjectModel3D but a new model is created. This is in contrast to the operator remove_object_model_3d_attrib_mod, which modifies the input model but functions identically otherwise.

If the Attributes do not exist in ObjectModel3D, no exception is raised.

Standard attributes

The following values for the parameter Attributes are possible:

'point_normal': This value specifies that the attribute with the 3D point normals and the attribute with the 3D point coordinates are removed.

'triangles': This value specifies that the attribute with the face triangles is removed.

'polygons': This value specifies that the attribute with the face polygon is removed.

'lines': This value specifies that the attribute with the lines is removed.

'xyz_mapping': This value specifies that the attribute with the mapping to image coordinates is removed.

'extended_attribute': This value specifies that all user-defined extended attributes are removed.

'primitives_all': This value specifies that the attribute with the parameters of the primitive (including an empty primitive) is removed (e.g., obtained from the operator fit_primitives_object_model_3d).

'primitive_plane': This value specifies that the attribute with the primitive plane is removed (e.g., obtained from the operator fit_primitives_object_model_3d).

'primitive_sphere': This value specifies that the attribute with the primitive sphere is removed (e.g., obtained from the operator fit_primitives_object_model_3d).

'primitive_cylinder': This value specifies that the attribute with the primitive cylinder is removed (e.g., obtained from the operator fit_primitives_object_model_3d).

'primitive_box': This value specifies that the attribute with the primitive cylinder is removed.

'shape_based_matching_3d_data': This value specifies that the attribute with the prepared shape model for shape-based 3D matching is removed.

'distance_computation_data': This value specifies that the attribute with the distance computation data structure is removed. The distance computation data can be created with prepare_object_model_3d, and can be used with distance_object_model_3d.

'all': This value specifies that all available attributes are removed except for the point coordinates. That is, the attributes are the point normals, the face triangles, the face polygons, the mapping to image coordinates, the shape model for matching, the parameter of a primitive, and the extended attributes.

Extended attributes

Extended attributes are attributes, that can be derived from standard attributes by special operators (e.g., distance_object_model_3d), or user-defined attributes (set with set_object_model_3d_attrib or set_object_model_3d_attrib_mod). The extended attributes can be removed by setting their names in Attributes.

The following predefined extended attributes can be removed:

'original_point_indices': This value specifies that the attribute with the original point indices is removed. Original point indices may be obtained from the operator triangulate_object_model_3d.

'score': This value specifies that the attribute with the scores is removed. Scores may be obtained from the operator reconstruct_surface_stereo.

'red': This value specifies that the attribute containing the red color is removed.

'green': This value specifies that the attribute containing the green color is removed.

'blue': This value specifies that the attribute containing the blue color is removed.

'edge_dir_x': This value specifies that the vector for the X axis is removed.

'edge_dir_y': This value specifies that the vector for the Y axis is removed.

'edge_dir_z': This value specifies that the vector for the Z axis is removed.

'edge_amplitude': This value specifies that the vector for the amplitude is removed.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the input 3D object model.

▷ **Attributes** (input_control) . . . . . string(-array) ⇝ *string*
Name of the attributes to be removed.
Default: 'extended_attribute'
List of values: Attributes ∈ {'point_normal', 'triangles', 'lines', 'polygons', 'xyz_mapping', 'shape_based_matching_3d_data', 'distance_computation_data', 'primitives_all', 'primitive_plane', 'primitive_sphere', 'primitive_cylinder', 'primitive_box', 'extended_attribute', 'score', 'red', 'green', 'blue', 'original_point_indices', 'all'}

▷ **ObjectModel3DOut** (output_control) . . . . . object_model_3d ⇝ *handle*
Handle of the resulting 3D object model.

Result

If the parameters are valid, the operator remove_object_model_3d_attrib returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_object_model_3d_attrib

Possible Successors

get_object_model_3d_params

Alternatives

remove_object_model_3d_attrib_mod

See also

copy_object_model_3d, set_object_model_3d_attrib

Module

3D Metrology

remove_object_model_3d_attrib_mod (: : ObjectModel3D,
Attributes :)

Remove attributes of a 3D object model.

remove_object_model_3d_attrib_mod removes the standard and/or extended attributes given in Attributes of a 3D object model ObjectModel3D. Doing so changes the 3D object model. This is in contrast to the operator remove_object_model_3d_attrib, which creates a new model but functions identically otherwise.

If the Attributes do not exist in ObjectModel3D, no exception is raised.

For a detailed description of Attributes see operator remove_object_model_3d_attrib.

Attention

remove_object_model_3d_attrib_mod removes Attributes unchecked from the 3D object model. Special attention must be paid to retain a consistent 3D object model, as most of the operators expect consistent 3D object models. Furthermore, the mapping of the 3D points to image coordinates should not be removed because it speeds up the computation of many operators.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the input 3D object model.

▷ **Attributes** (input_control) ..... string(-array) ⇝ *string*
Name of the attributes to be removed.
Default: 'extended_attribute'
List of values: `Attributes` ∈ {'point_normal', 'triangles', 'lines', 'polygons', 'xyz_mapping', 'shape_based_matching_3d_data', 'distance_computation_data', 'primitives_all', 'primitive_plane', 'primitive_sphere', 'primitive_cylinder', 'primitive_box', 'extended_attribute', 'score', 'red', 'green', 'blue', 'original_point_indices', 'all'}

Result

If the parameters are valid, the operator `remove_object_model_3d_attrib_mod` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`set_object_model_3d_attrib_mod`

Possible Successors

`get_object_model_3d_params`

Alternatives

`remove_object_model_3d_attrib`

See also

`copy_object_model_3d`, `set_object_model_3d_attrib_mod`

Module

3D Metrology

serialize_object_model_3d (: : ObjectModel3D : SerializedItemHandle)

Serialize a 3D object model.

`serialize_object_model_3d` serializes the data of a 3D object model (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The same data that is written in a file using the file format 'om3' of `write_object_model_3d` is converted to a serialized item. The 3D object model is defined by the handle `ObjectModel3D`. The serialized 3D object model is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_object_model_3d`.

Parameters

▷ **ObjectModel3D** (input_control) ..... object_model_3d ⇝ *handle*
Handle of the 3D object model.
▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_object_model_3d` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d

Possible Successors

read_object_model_3d, fwrite_serialized_item, send_serialized_item, deserialize_object_model_3d

See also

read_object_model_3d

Module

3D Metrology

set_object_model_3d_attrib (: : ObjectModel3D, AttribName, AttachExtAttribTo, AttribValues : ObjectModel3DOut)

Set attributes of a 3D object model.

set_object_model_3d_attrib sets the standard attributes or the extended attributes given in AttribName of a 3D object model ObjectModel3D to the values in AttribValues and returns a 3D object model with the new attribute values in ObjectModel3DOut. set_object_model_3d_attrib is identical to set_object_model_3d_attrib_mod with the exception that it creates a new 3D object model and leaves the original 3D object model unchanged. It is possible to attach the values of extended attributes to already existing standard attributes of the 3D object model by setting the parameter AttachExtAttribTo. For standard attributes, AttachExtAttribTo is ignored.

If the attributes in AttribName do not exist, they are created if possible. If already existing attributes are set, the length of AttribValues must match the existing attribute values. In this case the existing attribute values are replaced. If extended attributes are attached to already existing standard attributes with AttachExtAttribTo, the length of AttribValues must match the existing attribute values.

Standard attributes

The following standard attributes can be set:

'point_coord_x': The x-coordinates of the 3D points are set with AttribValues. If the attribute does not exist, the x-, y- and z-coordinates must be set with *'point_coord_x'*, *'point_coord_y'*, and *'point_coord_z'* at once. The number of x-, y-, and z-coordinates must be identical.

'point_coord_y': The y-coordinates of the 3D points are set with AttribValues. If the attribute does not exist, the x-, y- and z-coordinates must be set with *'point_coord_x'*, *'point_coord_y'*, and *'point_coord_z'* at once. The number of x-, y-, and z-coordinates must be identical.

'point_coord_z': The z-coordinates of the 3D points are set with AttribValues. If the attribute does not exist, the x-, y- and z-coordinates must be set with *'point_coord_x'*, *'point_coord_y'*, and *'point_coord_z'* at once. The number of x-, y-, and z-coordinates must be identical.

'point_normal_x': The x-components of the 3D point normals of the 3D points are set with AttribValues. If the attribute does not exist, the x-, y- and z-components of 3D point normals must be set with *'point_normal_x'*, *'point_normal_y'*, and *'point_normal_z'* at once. The number of x-, y-, and z-components must be identical to the number of 3D points. Note that the given 3D point normals will not be normalized to a length of 1.

'point_normal_y': The y-components of the 3D point normals of the 3D points are set with AttribValues. If the attribute does not exist, the x-, y- and z-components of 3D point normals must be set with *'point_normal_x'*, *'point_normal_y'*, and *'point_normal_z'* at once. The number of x-, y-, and z-components must be identical to the number of 3D points. Note that the given 3D point normals will not be normalized to a length of 1.

'point_normal_z': The z-components of the 3D point normals of the 3D points are set with AttribValues. If the attribute does not exist, the x-, y- and z-components of 3D point normals must be set with *'point_normal_x'*, *'point_normal_y'*, and *'point_normal_z'* at once. The number of x-, y-, and z-components must be identical to the number of 3D points. Note that the given 3D point normals will not be normalized to a length of 1.

'triangles': The indices of the 3D points that represent triangles are set with AttribValues in the following order: The first three values of AttribValues (input values 0,1,2) represent the first triangle and contain the indices of the corresponding 3D points of the triangle corners. The second three values (input values 3,4,5) represent the second triangle etc. The direction of the triangles results from the order of the point indices.

'polygons': The indices of the 3D points that represent polygons are set with AttribValues in the following order: The first value of AttribValues contains the number n of points of the first polygon. The following values (input values 1,2,..,n) contains the indices of the points of the first polygon. The next value (input value n+1) contains the number m of the points of the second polygon. The following m values (input values n+2,n+3,..,n+1+m) contain the indices of the points of the second polygon etc.

'lines': The indices of the 3D points that represent polylines are set with AttribValues in the following order: The first value of AttribValues contains the number n of points of the first polyline. The following values (input values 1,2,..,n) represent the indices of the points of the first polyline. The next value (input value n+1) contains the number m of points of the second polyline. The following m values (input values n+2,n+3,..,n+1+m) represent the indices of the points of the second polyline etc. All indices correspond to already existing 3D points.

'xyz_mapping': The mapping of 3D points to image coordinates is set with AttribValues in the following order: The first two values of AttribValues (input value 0 and 1) contain the width and height of the respective image. The following n values (input values 2,3,..,n+1, with n being the number of 3D points) represent the row coordinates of the n points given in image coordinates. The next n input values (input values n+2,n+3,..,n*2+1) represent the column coordinates of the n points in image coordinates. Hence, the total number of input values is n*2+2.

Extended attributes

Extended attributes are attributes, that can be derived from standard attributes by special operators (e.g., distance_object_model_3d), or user-defined attributes. Predefined extended attributes can only be set separately, for these attributes AttachExtAttribTo will be ignored. The names of user-defined extended attributes are arbitrary, but must start with the prefix *'&'*, e.g., *'&my_attrib'*. Extended attributes can have an arbitrary number of floating point values.

The following predefined extended attributes can be set:

'original_point_indices': The original points indices of the 3D points are set with AttribValues. The number of the original points indices must be identical to the number of 3D points.

'score': The score of a 3D reconstruction of the 3D points are set with AttribValues. Since the score is evaluated separately for each 3D point, the number of the score-components must be identical to the number of 3D points.

'red': The red channel intensities of the 3D points are set with AttribValues. The number of color values must be identical to the number of 3D points.

'green': The green channel intensities of the 3D points are set with AttribValues. The number of color values must be identical to the number of 3D points.

'blue': The blue channel intensities of the 3D points are set with AttribValues. The number of color values must be identical to the number of 3D points.

'edge_dir_x': The x-component of a vector that is perpendicular to the edge direction and the viewing direction.

'edge_dir_y': The y-component of a vector that is perpendicular to the edge direction and the viewing direction.

'edge_dir_z': The z-component of a vector that is perpendicular to the edge direction and the viewing direction.

'edge_amplitude': Contains the amplitude of edge points.

Extended attributes can be attached to already existing standard attributes of the 3D object model by setting the parameter AttachExtAttribTo. The following values of AttachExtAttribTo are possible:

'object' **or** *[]*: If this value is set, the extended attribute specified in AttribName is associated to the 3D object model as a whole. The number of values specified in AttribValues is not restricted.

'points': If this value is set, the extended attribute specified in AttribName is associated to the 3D points of the object model. The number of values specified in AttribValues must be the same as the number of already existing 3D points.

'triangles': If this value is set, the extended attribute specified in `AttribName` is associated to the triangles of the object model. The number of values specified in `AttribValues` must be the same as the number of already existing triangles.

'polygons': If this value is set, the extended attribute specified in `AttribName` is associated to the polygons of the object model. The number of values specified in `AttribValues` must be the same as the number of already existing polygons.

'lines': If this value is set, the extended attribute specified in `AttribName` is associated to the lines of the object model. The number of values specified in `AttribValues` must be the same as the number of already existing lines.

Attention

If multiple attributes are given in `AttribName`, `AttribValues` is divided into sub-tuples of equal length. Each sub-tuple is then assigned to one attribute. E.g., if `AttribName` and `AttribValues` are set to

`AttribName` := *['&attrib1','&attrib2','&attrib3']*,

`AttribValues` := *[0.0,1.0,2.0,3.0,4.0,5.0]*,

the following values are assigned to the individual attributes:

'&attrib1' = [0.0,1.0], *'&attrib2' = [2.0,3.0]*, *'&attrib3' = [4.0,5.0]*.

Consequently, it is not possible to set multiple attributes of different lengths in one call.

`set_object_model_3d_attrib` stores the input `AttribValues` unmodified in the 3D object model. Therefore, special attention must be paid to the consistency of the input data, as most of the operators expect consistent 3D object models.

Parameters

▷ **`ObjectModel3D`** (input_control) ..... object_model_3d ⇝ *handle*
Handle of the input 3D object model.

▷ **`AttribName`** (input_control) ..... string(-array) ⇝ *string*
Name of the attributes.
List of values: `AttribName` ∈ {'point_coord_x', 'point_coord_y', 'point_coord_z', 'point_normal_x', 'point_normal_y', 'point_normal_z', 'triangles', 'polygons', 'lines', 'xyz_mapping', 'red', 'green', 'blue', 'score', 'original_point_indices'}

▷ **`AttachExtAttribTo`** (input_control) ..... string ⇝ *string*
Defines where extended attributes are attached to.
Default: []
List of values: `AttachExtAttribTo` ∈ {[], 'object', 'points', 'polygons', 'triangles', 'lines'}

▷ **`AttribValues`** (input_control) ..... real(-array) ⇝ *real* / integer
Attribute values.

▷ **`ObjectModel3DOut`** (output_control) ..... object_model_3d ⇝ *handle*
Handle of the resulting 3D object model.

Result

If the parameters are valid, the operator `set_object_model_3d_attrib` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_empty_object_model_3d`

Possible Successors

`get_object_model_3d_params`

Alternatives

`set_object_model_3d_attrib_mod`

See also

copy_object_model_3d, remove_object_model_3d_attrib

Module

3D Metrology

set_object_model_3d_attrib_mod (: : ObjectModel3D, AttribName, AttachExtAttribTo, AttribValues :)

Set attributes of a 3D object model.

set_object_model_3d_attrib_mod sets the standard attributes or the extended attributes given in AttribName of a 3D object model ObjectModel3D to the values in AttribValues. set_object_model_3d_attrib_mod is identical to set_object_model_3d_attrib, with the exception that it does not create a new 3D object model but modifies the given one. It is possible to attach the values of extended attributes to already existing standard attributes of the 3D object model by setting the parameter AttachExtAttribTo. For standard attributes, AttachExtAttribTo is ignored.

If the attributes in AttribName do not exist, they are created if possible. If already existing attributes are set, the length of AttribValues must match the existing attribute values. In this case the existing attribute values are replaced. If extended attributes are attached to already existing standard attributes with AttachExtAttribTo, the length of AttribValues must match the existing attribute values.

For a detailed description see operator set_object_model_3d_attrib.

Attention

If multiple attributes are given in AttribName, AttribValues is divided into sub-tuples of equal length. Each sub-tuple is then assigned to one attribute. E.g., if AttribName and AttribValues are set to

AttribName := *['&attrib1','&attrib2','&attrib3']*,

AttribValues := *[0.0,1.0,2.0,3.0,4.0,5.0]*,

the following values are assigned to the individual attributes:

'&attrib1' = [0.0,1.0], *'&attrib2' = [2.0,3.0]*, *'&attrib3' = [4.0,5.0]*.

Consequently, it is not possible to set multiple attributes of different lengths in one call.

set_object_model_3d_attrib_mod modifies the content of an already existing 3D object model. The operator stores the input AttribValues unmodified in the 3D object model. Therefore, special attention must be paid to the consistency of the input data, as most of the operators expect consistent 3D object models.

Parameters

- ▷ **ObjectModel3D** (input_control) ..... object_model_3d ⇝ *handle*
  Handle of the 3D object model.
- ▷ **AttribName** (input_control) ..... string(-array) ⇝ *string*
  Name of the attributes.
  List of values: AttribName ∈ {'point_coord_x', 'point_coord_y', 'point_coord_z', 'point_normal_x', 'point_normal_y', 'point_normal_z', 'triangles', 'polygons', 'lines', 'xyz_mapping', 'red', 'green', 'blue', 'score', 'original_point_indices'}
- ▷ **AttachExtAttribTo** (input_control) ..... string ⇝ *string*
  Defines where extended attributes are attached to.
  Default: []
  List of values: AttachExtAttribTo ∈ {[], 'object', 'points', 'polygons', 'triangles', 'lines'}
- ▷ **AttribValues** (input_control) ..... real(-array) ⇝ *real* / integer
  Attribute values.

Result

If the parameters are valid, the operator set_object_model_3d_attrib_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_empty_object_model_3d

Possible Successors

get_object_model_3d_params

Alternatives

set_object_model_3d_attrib

See also

copy_object_model_3d, remove_object_model_3d_attrib_mod

Module

3D Metrology

union_object_model_3d (: : ObjectModels3D, Method : UnionObjectModel3D)

Combine several 3D object models to a new 3D object model.

union_object_model_3d combines the data of all input models in ObjectModels3D to a new 3D object model that is returned in UnionObjectModel3D.

Overlapping areas in the 3D object models might cause the potential 2D mapping, polygons, or triangles in the output to be less useful, since they might overlap, too.

The only supported Method is so far *'points_surface'*, which combines all points, surfaces and lines into the output UnionObjectModel3D. Extended Attributes are copied if no holes appear, i.e., if they are present in all input object models where the standard attribute they are attached to exists.

Attention

union_object_model_3d ignores 3D object models of type 3D primitive and 3D shape model.

Parameters

▷ **ObjectModels3D** (input_control) ..... object_model_3d(-array) ⇝ *handle*
Handle of input 3D object models.

▷ **Method** (input_control) ..... string ⇝ *string*
Method used for the union.
Default: 'points_surface'
List of values: Method ∈ {'points_surface'}

▷ **UnionObjectModel3D** (output_control) ..... object_model_3d ⇝ *handle*
Handle of the resulting 3D object model.

Example

```
gen_object_model_3d_from_points ([0,0,0,0],[1,1,0,0], [0,1,1,0],\
                                 ObjectModel3D1)
gen_object_model_3d_from_points ([1,1,1,1],[1,1,0,0], [0,1,1,0],\
                                 ObjectModel3D2)
get_object_model_3d_params (ObjectModel3D1, 'diameter', DiameterOld)
union_object_model_3d ([ObjectModel3D1,ObjectModel3D2], 'points_surface',\
                       UnionObjectModel3D)
get_object_model_3d_params (UnionObjectModel3D, 'diameter', DiameterNew)
```

Result

union_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If there is no attribute common in all input objects, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

get_object_model_3d_params

Possible Successors

connection_object_model_3d, convex_hull_object_model_3d

See also

gen_box_object_model_3d, gen_sphere_object_model_3d, gen_cylinder_object_model_3d

Module

3D Metrology

write_object_model_3d (: : ObjectModel3D, FileType, FileName, GenParamName, GenParamValue :)

Writes a 3D object model to a file.

The operator write_object_model_3d writes the 3D object model ObjectModel3D to the file FileName. The object model can be read again with read_object_model_3d, or can be imported into an appropriate CAD program. Please note, that primitives may only be stored in the HALCON format *'om3'*. Should it be necessary to store the primitives in another format, the operator sample_object_model_3d has to be called beforehand. However, this results in a transformation of the primitives into 3D points and therefore only corresponds to an approximation of the primitives.

All coordinates are written in meters. If the file is read later using read_object_model_3d, the parameter Scale must be set to *'m'* to avoid scaling the data.

The parameter FileType determines the type of the file. The following types are supported by this operator:

'om3': HALCON format for object model 3D. Files with this format can be read by read_object_model_3d. The default file extension for this format is 'om3'.

'dxf': AUTOCAD format. See read_object_model_3d for details about reading this file format. The default file extension for this format is 'dxf'.

'off': Object File Format. This is an ASCII-based format that can hold 3D points and polygons. The default file extension for this format is 'off'.

'ply',

'ply_binary': Polygon File Format (also Stanford Triangle Format). This is a simple format that can hold 3D points, point normals, polygons, color information and point-based extended attributes. HALCON supports the writing of both the ASCII and the binary version of this format. The default file extension for this format is 'ply'.

'obj': OBJ file format, also Wavefront OBJ-Format. This is an ASCII-based format that can hold 3D points, polygons, normals, and triangles, which are stored as polygons. The default file extension for this format is 'obj'.

'stl',

'stl_binary',

'stl_ascii': STL file format, also 'Stereolithography format', 'SurfaceTesselationLanguage ', 'StandardTriangulationLanguage', and 'StandardTesselationLanguage'. This format stores triangles and triangle normals. However, as triangle normals are not supported by HALCON 3D object models and point normals (which are, for example, calculated by surface_normals_object_model_3d) are not supported by the STL format, no normals are written to file. If the 3D object model contains polygons, they are converted to triangles before before writing them to disc. If the file type is set to *'stl'* or *'stl_binary'*, the binary version of STL is written while *'stl_ascii'* selects the ASCII version. The default file extension for this format is 'stl'.

A set of additional optional parameters can be set. The names and values of the parameters are passed in GenParamName and GenParamValue, respectively. Some of the optional parameters can only be set for a certain file type. The following values for GenParamName are possible:

'invert_normals': Invert normals and face orientation before saving the 3D object model. If this value is set to *'true'*, for the formats *'off'*, *'ply'*, *'obj'*, and *'stl'* the orientation of faces (triangles and polygons) is inverted . For formats that support point normals (*'ply'*,*'obj'*), all normals are inverted before writing them to disc. Note that for the types *'om3'* and *'dxf'* the parameter has no effect.
Valid for formats: 'off', 'ply', 'obj', 'stl'. **List of values:** *'true'*, *'false'*.
Default: *'false'*.

Parameters

▷ **ObjectModel3D** (input_control) ..... object_model_3d ⇝ *handle*
Handle of the 3D object model.

▷ **FileType** (input_control) ..... string ⇝ *string*
Type of the file that is written.
Default: 'om3'
List of values: FileType ∈ {'off', 'ply', 'ply_binary', 'dxf', 'om3', 'obj', 'stl', 'stl_binary', 'stl_ascii'}

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Name of the file that is written.
File extension: .off, .ply, .dxf, .om3, .obj, .stl

▷ **GenParamName** (input_control) ..... string(-array) ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'invert_normals'}

▷ **GenParamValue** (input_control) ..... string(-array) ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

Result

The operator write_object_model_3d returns the value 2 (H_MSG_TRUE) if the given parameters are correct and the file can be written. If necessary, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d

Possible Successors

read_object_model_3d

See also

read_object_model_3d

Module

3D Metrology

4.2 Features

area_object_model_3d (: : ObjectModel3D : Area)

Calculate the area of all faces of a 3D object model.

area_object_model_3d calculates the area of all faces in a 3D object model. The 3D object model requires faces or triangles. The resulting area is returned in Area.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **Area** (output_control) . . . . . number(-array) ⇝ *real*
Calculated area.
Number of elements: Area == ObjectModel3D

Example

```
gen_box_object_model_3d ([0,0,0,0,0,0,0],3,2,1, ObjectModel3D)
convex_hull_object_model_3d (ObjectModel3D, ObjectModel3DConvexHull)
area_object_model_3d (ObjectModel3DConvexHull, Area)
```

Result

`area_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`connection_object_model_3d`, `select_points_object_model_3d`, `prepare_object_model_3d`, `convex_hull_object_model_3d`

Possible Successors

`select_object_model_3d`

See also

`volume_object_model_3d_relative_to_plane`, `max_diameter_object_model_3d`, `moments_object_model_3d`

Module

3D Metrology

distance_object_model_3d (: : ObjectModel3DFrom, ObjectModel3DTo, Pose, MaxDistance, GenParamName, GenParamValue :)

Compute the distances of the points of one 3D object model to another 3D object model.

The operator `distance_object_model_3d` computes the distances of the points in the 3D object model `ObjectModel3DFrom` to the points, triangles, polygons, or primitive in the 3D object model `ObjectModel3DTo`. The distances are stored as an extended attribute named *'&distance'* in the 3D object model `ObjectModel3DFrom`. This attribute can subsequently be queried with `get_object_model_3d_params` or be processed with `select_points_object_model_3d` or other operators that use extended attributes.

The target data (points, triangles, polygons, or primitive) is selected based on the attributes contained in `ObjectModel3DTo`. It is selected based on the presence of the data in the following precedence: Primitive, triangles, polygons, and points. As alternative to this automatic target data selection, the target data type can also be set with the generic parameter *'distance_to'* (see below). Generic, non-triangular polygons are internally triangulated by the operator before the distance to the resulting triangles is calculated. Thus, calling the operator with triangulated objects is faster than calling it with objects having different polygon faces.

`MaxDistance` can be used to limit the range of the distance values to be computed. If `MaxDistance` is set to *0*, all distances are computed. If `MaxDistance` is set to another value, points whose distance would exceed `MaxDistance` are not processed and set to `MaxDistance`. Thus, setting `MaxDistance` to a value different than *0* can significantly speed up the execution of this operator.

If `Pose` is a non-empty tuple, it must contain a pose which is applied to the points in `ObjectModel3DFrom` before computing the distances. The pose can be inverted using the generic parameter *'invert_pose'* (see below).

Depending on the target data type (points, triangles, or primitive), several methods for computing the distances are available. Some of these methods compute a data structure on the elements of ObjectModel3DTo to speed up the distance computation. Those data structures can be precomputed using the operator prepare_object_model_3d. This allows multiple calls to distance_object_model_3d to re-use the data structure, thus saving the time to re-compute it for each call. For objects with non-triangular polygon faces, the operator prepare_object_model_3d can additionally perform the triangulation and save it to the object to further speed-up the distance_object_model_3d operator. This triangulation is only performed when the generic parameter *'distance_to'* is set to *'triangles'*. Note that this triangulation, contrary to that of the operator triangulate_object_model_3d, does not clear out the polygons attribute.

When computing the distance to points or to triangles, the operator can optionally return the index of the closest point or triangle for each point in ObjectModel3DFrom by setting the generic parameter *'store_closest_index'* to *'true'* (see below). The index is stored as extended attribute named *'&closest_index'* in the 3D object model ObjectModel3DFrom. Note that the closest index can not be computed when using the *'voxel'* method. If a point's distance to its closest element exceeds the maximum distance set in MaxDistance, the closest index is set to -1.

Optionally, signed distances to points, triangles or to a primitive can be calculated. Therefore, the generic parameter *'signed_distances'* has to be set to *'true'*. Note that signed distances can not be computed when using the *'voxel'* method in combination with point to point distances.

In the following, the different target types and methods are explained, and their advantages and disadvantages are described. Note that the operator automatically selects a default method depending on the target data type. This method can be overridden using the generic parameter *'method'*.

Distance to points: The following methods are available to compute the distances from points to points:

- **Linear search:** For each point in ObjectModel3DFrom, the distances to all points in ObjectModel3DTo are computed, and the smallest distance is used. This method requires no precomputed data structure, and is the fastest for a small number of points in ObjectModel3DTo.
- **KD-Tree:** The points in ObjectModel3DTo are organized in a KD-Tree, which speeds up the search for the closest point. The construction of the tree is very efficient. The search time is approximately logarithmic to the number of points in ObjectModel3DTo. However, the search time is not constant, and can vary significantly depending on the position of the query points in ObjectModel3DFrom.
- **Voxel:** The points in ObjectModel3DTo are organized in a voxel structure. This voxel structure allows searching in almost constant time, i.e., independent from the position of the query points in ObjectModel3DFrom and the number of points in ObjectModel3DTo.
  Note that the preparation of this data structure takes several seconds or minutes. However, it is possible to perform a precomputation using prepare_object_model_3d on ObjectModel3DTo with Purpose set to *'distance_computation'*.

Distance to triangles: For computing the distances to triangles, the following methods are supported:

- **Linear search:** For each point in ObjectModel3DFrom, the distances to all triangles in ObjectModel3DTo are computed, and the smallest distance is used. This method requires no precomputed data structure, and is the fastest for a small number of triangles in ObjectModel3DTo.
- **KD-Tree:** The triangles in ObjectModel3DTo are organized in a KD-Tree, which speeds up the search for the closest triangle. The construction of the tree is efficient. The search time is approximately logarithmic to the number of triangles in ObjectModel3DTo. However, the search time is not constant, and can vary significantly depending on the position of the query points in ObjectModel3DFrom.
- **Voxel:** The triangles in ObjectModel3DTo are organized in a voxel structure. This voxel structure allows searching in almost constant time, i.e., independent from the position of the query points in ObjectModel3DFrom and the number of triangles in ObjectModel3DTo.
  Note that the preparation of this data structure takes several seconds or minutes. However, it is possible to perform a precomputation using prepare_object_model_3d on ObjectModel3DTo with Purpose set to *'distance_computation'*. For creating the voxel data structure, the triangles are sampled. The corresponding sampling distance can be set with the generic parameters *'sampling_dist_rel'* and *'sampling_dist_abs'*.
  By default, a relative sampling distance of *0.03* is used. See below for a more detailed description of the sampling distance. Note that this data structure is only approximate. It is possible that some of the distances are off by around 10% of the sampling distance. In these cases, the returned distances will always be larger than the actual distances.

Distance to primitive: Since ObjectModel3DTo can contain only one primitive, the distances from the query points to this primitive are computed linearly. The creation or usage of a data structure is not possible.

Note that computing the distance to primitive planes fitted with segment_object_model_3d or fit_primitives_object_model_3d can be slow, since those planes contain a complex convex hull of the points that were used to fit the plane. If only the distance to the plane is required, and the boundary should be ignored, it is recommended to obtain the plane pose using get_object_model_3d_params with parameter *'primitive_parameter_pose'* and create a new plane using gen_plane_object_model_3d.

The following table lists the different target data types, methods, and their properties. The search time is the approximate time per point in ObjectModel3DFrom. N is the number of target elements in ObjectModel3DTo.

Target Data	Method	Creation Time	Approximate Search Time	Properties
points	linear	0	$O(N)$	· No precomputation · Fastest for small N · Default for $N < 100$
points	kd-tree	$O(N \log(N))$	$O(\log(N))$	· Fast structure creation · Non-constant search time · Default for $N \geq 100$
points	voxel	$O(N \log(N))$	$O(\log(\log(N)))$	· Slow structure creation · Very fast search · Default for precomputation with prepare_object_model_3d
triangles	linear	0	$O(N)$	· No precomputation · Fastest for small N · Default
triangles	kd-tree	$O(N \log(N))$	$O(\log(N))$	· Fast structure creation · Non-constant search time
triangles	voxel	$O(N \log(N))$	$O(\log(\log(N)))$	· Slow structure creation · Requires sampling distance · Very fast search · Small errors possible · Default for precomputation with prepare_object_model_3d
primitive	linear	0	$O(1)$	

Additionally to the parameters described above, the following parameters can be set to influence the distance computation. If desired, these parameters and their corresponding values can be specified by using GenParamName and GenParamValue, respectively. All of the following parameters are optional.

'distance_to' This parameter can be used to explicitly set the target data to which the distances are computed.

'auto' (Default) Automatically set the target data. The following list of attributes is queried, and the first appearing attribute from the list is used as target data: Primitive, Triangle, Point.

'primitive' Compute the distance to the primitive contained in `ObjectModel3DTo`.

'triangles' Compute the distance to the triangles contained in `ObjectModel3DTo`.

'points' Compute the distance to the points contained in `ObjectModel3DTo`.

'method' This parameter can be used to explicitly set the method to be used for the distance computation. Note that not all methods are available for all target data types. For the list of possible pairs of target data type and method, see above.

'auto' (Default) Use the default method for the used target data type.

'linear' Use a linear search for computing the distances.

'kd-tree' Use a KD-Tree for computing the distances.

'voxel' Use a voxel structure for computing the distances.

'invert_pose' This parameter can be used to invert the pose given in `Pose`.

'false' (Default) The pose is not inverted.

'true' The pose is inverted.

'output_attribute' This parameter can be used to set the name of the attribute in which the distances are stored. By default, the distances are stored in an extended attribute named *'&distance'* in `ObjectModel3DFrom`. However, if the same 3D object model is used for different calls of this operator, the result of the previous call would be overwritten. This can be avoided by changing the name of the extended attribute. Valid extended attribute names start with a *'&'*.

'sampling_dist_rel', *'sampling_dist_abs'* These parameters are used when computing the distances to triangles using the voxel method. For this, the triangles need to be sampled. The sampling distance can be set either in absolute terms, using *'sampling_dist_abs'*, or relative to the diameter of the axis aligned bounding box, using *'sampling_dist_rel'*. By default, *'sampling_dist_rel'* is set to 0.03. Only one of the two parameters can be set. The diameter of the axis aligned bounding box can be queried using `get_object_model_3d_params`. Note that the creation of the voxel data structure is very time consuming, and is usually performed offline using `prepare_object_model_3d` (see above).

'store_closest_index' This parameter can be used to return the index of the closest point or triangle in the extended attribute '&closest_index'.

'false' (Default) The index is not returned.

'true' The index is returned.

'signed_distances' This parameter can be used to calculate signed distances of the points in the 3D object model `ObjectModel3DFrom` to the points, triangles or primitive in the 3D object model `ObjectModel3DTo`.

'false' (Default) Unsigned distances are returned.

'true' Signed distances are returned.

Dependent on the available target data (points, triangles or primitive) the following particularities have to be considered:

Distance to points: The computation of signed distances is only supported for the methods *'kd-tree'* and *'linear'*. However, signed distances can only be calculated if point normals are available for the points in the 3D object model or attached via the operator `set_object_model_3d_attrib_mod`.

Distance to triangles: Signed distances can be calculated for all methods listed above. The operator returns a negative distance, if the dot product with the normal vector of the triangle is less than zero. Otherwise, the distance is positive.

Distance to primitive: When calculating signed distances to a cylindrical, spherical or box-shaped primitive, the points of the 3D object model `ObjectModel3DFrom` inside the primitive obtain a negative distance, whereas all others have a positive distance. When calculating signed distances to planes, all points beneath the plane obtain a negative distance, whereas all others have a positive one.

Parameters

▷ **ObjectModel3DFrom** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the source 3D object model.

▷ **ObjectModel3DTo** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the target 3D object model.

▷ **Pose** (input_control) . . . . . pose ⇝ *real* / integer
Pose of the source 3D object model in the target 3D object model.
Default: []

▷ **MaxDistance** (input_control) . . . . . number ⇝ *real* / integer
Maximum distance of interest.
Default: 0

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic input parameters.
Default: []
List of values: GenParamName ∈ {'distance_to', 'method', 'invert_pose', 'output_attribute', 'sampling_dist_rel', 'sampling_dist_abs', 'signed_distances', 'store_closest_index'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Values of the generic input parameters.
Default: []
List of values: GenParamValue ∈ {'auto', 'triangles', 'points', 'polygons', 'primitive', 'kd-tree', 'voxel', 'linear', 'true', 'false'}

Result

distance_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

prepare_object_model_3d, read_object_model_3d, find_surface_model, xyz_to_object_model_3d

Possible Successors

get_object_model_3d_params, render_object_model_3d, disp_object_model_3d, clear_object_model_3d

See also

prepare_object_model_3d

Module

3D Metrology

get_object_model_3d_params (: : ObjectModel3D, GenParamName : GenParamValue)

Return attributes of 3D object models.

A 3D object model consists of a set of attributes and meta data. The operator get_object_model_3d_params allows to access attributes and meta data of the given 3D object models. The name of the requested attribute or meta data is passed in the generic parameter GenParamName, the corresponding value is returned in GenParamValue. If a requested attribute or meta data is not available, an exception is raised. get_object_model_3d_params supports to access several 3D object models and several attributes at once. Note that the attributes or meta data can have different lengths. Some of the standard attributes have a defined length as noted in the attribute description below. The length of other attributes depends on the actual 3D object model, and can be queried by setting the parameter GenParamName to, e.g.,

'num_points', *'num_triangles'*, *'num_polygons'*, or *'num_lines'*. Thus, to get the length of the standard attribute *'point_coord_x'*, set GenParamName to *'num_points'*.

Standard attributes

The following standard attributes and meta data can be accessed:

'point_coord_x': The x-coordinates of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator xyz_to_object_model_3d or read_object_model_3d.

'point_coord_y': The y-coordinates of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator xyz_to_object_model_3d or read_object_model_3d.

'point_coord_z': The z-coordinates of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator xyz_to_object_model_3d or read_object_model_3d.

'point_normal_x': The x-components of 3D point normals of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator smooth_object_model_3d.

'point_normal_y': The y-components of 3D point normals of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator smooth_object_model_3d.

'point_normal_z': The z-components of 3D point normals of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator smooth_object_model_3d.

'mapping_row': The row-components of the 2D mapping of the set of 3D points. (length can be queried by *'num_points'*, height of the original image can be queried by *'mapping_size'*). This attribute is obtained typically from the operator xyz_to_object_model_3d.

'mapping_col': The column-components of the 2D mapping of the set of 3D points. (length can be queried by *'num_points'*, width of the original image can be queried by *'mapping_size'*). This attribute is obtained typically from the operator xyz_to_object_model_3d.

'mapping_size': The size of the original image. A tuple with the two entries width and height is returned.

'triangles': The indices of the 3D points that represent triangles in the following order: The first three values (return values 0,1,2) represent the first triangle. The next three values (return values 3,4,5) represent the second triangle etc. All indices correspond to the coordinates of the 3D points. Access to the coordinates of the 3D points is possible, e.g., with the generic parameter GenParamName set to the values *'point_coord_x'*, *'point_coord_y'*, and *'point_coord_z'*, respectively. The length of this attribute corresponds to three times the number of triangles, which can be queried using *'num_triangles'*. This attribute is obtained typically from the operator triangulate_object_model_3d or read_object_model_3d.

'polygons': The indices of the 3D points that represent polygons in the following order: The first return value contains the number n of the points of the first polygon. The following values (return values 1,2,..,n) represent the indices of the points of the first polygon. The next value (return value n+1) contains the number m of the points of the second polygon. The following m return values (return values n+2,n+3,..,n+1+m) represent the indices of the points of the second polygon etc. All indices correspond to the coordinates of the 3D points. Access to the coordinates of the 3D points is possible, e.g., with the generic parameter GenParamName set to the values *'point_coord_x'*, *'point_coord_y'*, and *'point_coord_z'*, respectively. The number of polygons per 3D object model can be queried using *'num_polygons'*. This attribute is obtained typically from the operator read_object_model_3d.

'lines': The indices of the 3D points that represent polylines in the following order: The first return value contains the number n of points of the first polyline. The following values (return values 1,2,..,n) represent the indices of the points of the first polyline. The next value (return value n+1) contains the number m of points of the second polyline. The following m values (return values n+2,n+3,..,n+1+m) represent the indices of the points of the second polyline etc. All indices correspond to the coordinates of the 3D points. Access to the coordinates of the 3D points is possible, e.g., with the generic parameter GenParamName set to the values *'point_coord_x'*, *'point_coord_y'*, and *'point_coord_z'*, respectively. The number of lines per 3D object model can be queried using *'num_lines'*. This attribute is obtained typically from the operator intersect_plane_object_model_3d.

'diameter_axis_aligned_bounding_box': The diameter of the set of 3D points, defined as the length of the diagonal of the smallest enclosing axis-parallel cuboid (see parameter *'bounding_box1'*). This attribute has length 1.

'center': 3D coordinates of the center of the set of 3D points. These coordinates are the center of the smallest enclosing axis-parallel cuboid (see parameter *'bounding_box1'*). This attribute has length 3. If there are no 3D coordinates in the 3D object model the following rules are valid:

If the 3D object model is a primitive of type cylinder (see gen_cylinder_object_model_3d) and there are extensions, the center point between the extensions are returned. If there are no extensions the translation parameters of the pose are returned.

If the 3D object model is a primitive of type plane (see gen_plane_object_model_3d) and there are extensions, the center of gravity of the plane is computed from the extensions. If there are no extensions the translation parameters of the pose are returned.

If the 3D object model is a primitive of type sphere or box (see gen_sphere_object_model_3d or gen_box_object_model_3d), the center point of the object model is returned.

'primitive_type': The primitive type (e.g., obtained from the operator fit_primitives_object_model_3d). The return value of a sphere is *'sphere'*. The return value of a cylinder is *'cylinder'*. The return value of a plane is *'plane'*. The return value of a box is *'box'*. This attribute has length 1.

'primitive_parameter': The parameters of the primitive (e.g., obtained from the operator fit_primitives_object_model_3d). The length of this attribute depends on *'primitive_type'* and is between 4 and 10 for each 3D object model.

If the 3D object model is a primitive of type cylinder (see gen_cylinder_object_model_3d), the return values are the (x-, y-, z-)coordinates of the center [x_center, y_center, z_center], the normed (x-, y-, z-)directions of the main axis of the cylinder [x_axis, y_axis, z_axis], and the radius [radius] of the cylinder. The order is [x_center, y_center, z_center, x_axis, y_axis, z_axis, radius].

If the 3D object model is a primitive of type sphere (see gen_sphere_object_model_3d), the return values are the (x-, y-, z-)coordinates of the center [x_center, y_center, z_center] and the radius [radius] of the sphere. The order is [x_center, y_center, z_center, radius].

If the 3D object model is a primitive of type plane (see gen_plane_object_model_3d), the 4 parameters of the hessian normal form are returned, i.e., the unit normal (x-, y-, z-) vector [x, y, z] and the orthogonal distance (d) of the plane from the origin of the coordinate system. The order is [x, y, z, d]. The sign of the distance (d) determines the side of the plane on which the origin is located.

If the 3D object model is a primitive of type box (gen_box_object_model_3d), the return values are the 3D pose (translation, rotation, type of the rotation) and the half edge lengths (length1, length2, length3) of the box. length1 is the length of the box along the x axis of the pose. length2 is the length of the box along the y axis of the pose. length3 is the length of the box along the z axis of the pose. The order is [trans_x, trans_y, trans_z, rot_x, rot_y, rot_z, rot_type, length1, length2, length3]. For details about 3D poses and the corresponding transformation matrices see the operator create_pose.

'primitive_parameter_pose': The parameters of the primitive with format of a 3D pose (e.g., obtained from the operator fit_primitives_object_model_3d). For all types of primitives the return values are the 3D pose (translation, rotation, type of the rotation). For details about 3D poses and the corresponding transformation matrices see the operator create_pose. The length of this attribute depends on *'primitive_type'* and is between 7 and 10 for each 3D object model.

If the 3D object model is a primitive of type cylinder (see gen_cylinder_object_model_3d), additionally, the radius [radius] of the cylinder is returned. The order is [trans_x, trans_y, trans_z, rot_x, rot_y, rot_z, rot_type, radius].

If the 3D object model is a primitive of type sphere (see gen_sphere_object_model_3d), additionally, the radius [radius] of the sphere is returned. The order is [trans_x, trans_y, trans_z, rot_x, rot_y, rot_z, rot_type, radius].

If the 3D object model is a primitive of type plane (see gen_plane_object_model_3d), the order is [trans_x, trans_y, trans_z, rot_x, rot_y, rot_z, rot_type].

If the 3D object model is a primitive of type box (see gen_box_object_model_3d), additionally the half edge lengths (length1, length2, length3) of the box are returned. length1 is the length of the box along the x axis of the pose. length2 is the length of the box along the y axis of the pose. length3 is the length of the box along the z axis of the pose. The order is [trans_x, trans_y, trans_z, rot_x, rot_y, rot_z, rot_type, length1, length2, length3].

'primitive_pose': The parameters of the primitive with format of a 3D pose (e.g., obtained from the operator fit_primitives_object_model_3d). For all types of primitives the return values are the 3D pose

(translation, rotation, type of the rotation). For details about 3D poses and the corresponding transformation matrices see the operator create_pose. The length of this attribute is 7 for each 3D object model. The order is [trans_x, trans_y, trans_z, rot_x, rot_y, rot_z, rot_type].

'primitive_parameter_extension': The extents of the primitive of type cylinder and plane (e.g., obtained from the operator fit_primitives_object_model_3d). The length of this attribute depends on *'primitive_type'* and can be queried using *'num_primitive_parameter_extension'*.

If the 3D object model is a primitive of type cylinder (see gen_cylinder_object_model_3d), the return values are the extents (MinExtent, MaxExtent) of the cylinder. They are returned in the order [MinExtent, MaxExtent]. MinExtent represents the length of the cylinder in negative direction of the rotation axis. MaxExtent represents the length of the cylinder in positive direction of the rotation axis.

If the 3D object model is a primitive of type plane (created using fit_primitives_object_model_3d), the return value is a tuple of co-planar points regarding the fitted plane. The order is [x coordinate of point 1, x coordinate of point 2, x coordinate of point 3, ..., y coordinate of point 1, y coordinate of point 2, y coordinate of point 3, ...]. The coordinate values describe the support points of a convex hull. This is computed based on the projections of those points on the fitted plane which contribute to the fitting. If the plane was created using gen_plane_object_model_3d, all points that were used to create the plane (XExtent, YExtent) are returned.

'primitive_rms': The quadratic residual error of the primitive (e.g., obtained from the operator fit_primitives_object_model_3d). This attribute has length 1.

'reference_point': 3D coordinates of the reference point of the prepared 3D shape model for shape-based 3D matching. The reference point is the center of the smallest enclosing axis-parallel cuboid (see parameter *'bounding_box1'*). This attribute has length 3.

'bounding_box1': Smallest enclosing axis-parallel cuboid (min_x, min_y, min_z, max_x, max_y, max_z). This attribute has length 6.

'num_points': The number of points. This attribute has length 1.

'num_triangles': The number of triangles. This attribute has length 1.

'num_polygons': The number of polygons. This attribute has length 1.

'num_lines': The number of polylines. This attribute has length 1.

'num_primitive_parameter_extension': The number of extended data of primitives. This attribute has length 1.

'has_points': The existence of 3D points. This attribute has length 1.

'has_point_normals': The existence of 3D point normals. This attribute has length 1.

'has_triangles': The existence of triangles. This attribute has length 1.

'has_polygons': The existence of polygons. This attribute has length 1.

'has_lines': The existence of lines. This attribute has length 1.

'has_xyz_mapping': The existence of a mapping of the 3D points to image coordinates. This attribute has length 1.

'has_shape_based_matching_3d_data': The existence of a shape model for shape-based 3D matching. This attribute has length 1.

'has_distance_computation_data': The existence of a precomputed data structure for 3D distance computation. This attribute has length 1. The data structure can be created with prepare_object_model_3d using the purpose *'distance_computation'*. It is used by the operator distance_object_model_3d.

'has_surface_based_matching_data': The existence of data for the surface-based matching. This attribute has length 1.

'has_segmentation_data': The existence of data for a 3D segmentation. This attribute has length 1.

'has_primitive_data': The existence of a primitive. This attribute has length 1.

'has_primitive_rms': The existence of a quadratic residual error of a primitive. This attribute has length 1.

'neighbor_distance':

'neighbor_distance N': For every point the distance of the N-th nearest point. N must be a positive integer and is by default 25. For every point, all other points are sorted according to their distance and the distance of the N-th point is returned.

'num_neighbors X': For every point the number of neighbors within a distance of at most X.

'num_neighbors_fast X': For every point the approximate number of neighbors within a distance of at most X. The distances are approximated using voxels, leading to a faster processing compared to *'num_neighbors'*.

Extended attributes

Extended attributes are attributes, that can be derived from standard attributes by special operators (e.g., `distance_object_model_3d`), or user-defined attributes. User-defined attributes can be created by the operator `set_object_model_3d_attrib`. The following extended attributes and meta data can be accessed:

'extended_attribute_names': The names of all extended attributes. For each extended attribute name a value is returned.

'extended_attribute_types': The type of all extended attributes. For each extended attribute type a value is returned, thereby the values are sorted as the output for the extended attribute names.

'has_extended_attribute': The existence of at least one extended attribute. For each 3D object model a value is returned.

'num_extended_attribute': The number of extended attributes. For each 3D object model a value is returned.

'&attribute_name': The values stored under a user-defined extended attribute. Note that this name must start with *'&'*, e.g., *'&my_attrib'*. The data of the requested extended attributes are returned in `GenParamValue`. The order in which the data is returned is the same as the order of the attribute names specified in `GenParamName`.

'original_point_indices': Indices of the 3D points in a different 3D object model (length can be queried by *'num_points'*). This attribute is obtained typically from the operator `triangulate_object_model_3d`.

'score': The score of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator `reconstruct_surface_stereo`.

'red': The red channel of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator `reconstruct_surface_stereo`.

'green': The green channel of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator `reconstruct_surface_stereo`.

'blue': The blue channel of the set of the 3D points (length can be queried by *'num_points'*). This attribute is obtained typically from the operator `reconstruct_surface_stereo`.

'edge_dir_x': The x-component of a vector that is perpendicular to the edge direction and the viewing direction. This attribute is obtained typically from the operator `edges_object_model_3d`

'edge_dir_y': The y-component of a vector that is perpendicular to the edge direction and the viewing direction. This attribute is obtained typically from the operator `edges_object_model_3d`

'edge_dir_z': The z-component of a vector that is perpendicular to the edge direction and the viewing direction. This attribute is obtained typically from the operator `edges_object_model_3d`

'edge_amplitude': Contains the amplitude of edge points. This attribute is obtained typically from the operator `edges_object_model_3d`

Parameters

▷ **`ObjectModel3D`** (input_control) ..... object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **`GenParamName`** (input_control) ..... attribute.name-array ⇝ *string*
Names of the generic attributes that are queried for the 3D object model.
Default: 'num_points'
List of values: `GenParamName` ∈ {'point_coord_x', 'point_coord_y', 'point_coord_z', 'point_normal_x', 'point_normal_y', 'point_normal_z', 'mapping_row', 'mapping_col', 'mapping_size', 'triangles', 'polygons', 'lines', 'diameter_axis_aligned_bounding_box', 'center', 'primitive_type', 'primitive_rms', 'primitive_parameter', 'primitive_parameter_pose', 'primitive_pose', 'primitive_parameter_extension', 'reference_point', 'bounding_box1', 'num_points', 'num_triangles', 'num_polygons', 'num_lines', 'num_primitive_parameter_extension', 'has_points', 'has_point_normals', 'has_triangles', 'has_polygons', 'has_lines', 'has_xyz_mapping', 'has_shape_based_matching_3d_data', 'has_surface_based_matching_data', 'has_segmentation_data', 'has_primitive_data', 'has_primitive_rms', 'extended_attribute_names', 'extended_attribute_types', 'has_extended_attribute', 'num_extended_attribute', 'has_distance_computation_data', 'red', 'green', 'blue', 'score', 'neighbor_distance', 'num_neighbors', 'num_neighbors_fast', 'original_point_indices', 'edge_amplitude', 'edge_dir_x', 'edge_dir_y', 'edge_dir_z'}

▷ **GenParamValue** (output_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Values of the generic parameters.

Result

The operator `get_object_model_3d_params` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`read_object_model_3d`, `xyz_to_object_model_3d`, `prepare_object_model_3d`, `sample_object_model_3d`, `triangulate_object_model_3d`, `intersect_plane_object_model_3d`, `set_object_model_3d_attrib`, `fit_primitives_object_model_3d`, `gen_plane_object_model_3d`, `gen_sphere_object_model_3d`, `gen_cylinder_object_model_3d`, `gen_box_object_model_3d`, `gen_sphere_object_model_3d_center`

Possible Successors

`select_object_model_3d`, `write_object_model_3d`, `clear_object_model_3d`

Module

3D Metrology

max_diameter_object_model_3d (: : ObjectModel3D : Diameter)

Calculate the maximal diameter of a 3D object model.

`max_diameter_object_model_3d` calculates the maximal diameter of the 3D object model by calculating the convex hull of the object and searching for the pair of points on the convex hull with the largest distance.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **Diameter** (output_control) . . . . . number(-array) ⇝ *real*
Calculated diameter.
Number of elements: Diameter == ObjectModel3D

Example

```
gen_object_model_3d_from_points (rand(200), rand(200),\
                                 rand(200), ObjectModel3D)
max_diameter_object_model_3d (ObjectModel3D, Diameter)
```

Result

`max_diameter_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_object_model_3d, connection_object_model_3d

Possible Successors

select_object_model_3d

See also

volume_object_model_3d_relative_to_plane, area_object_model_3d, moments_object_model_3d

Module

3D Metrology

moments_object_model_3d (: : ObjectModel3D, MomentsToCalculate : Moments)

Calculates the mean or the central moment of second order for a 3D object model.

moments_object_model_3d calculates the mean or the central moment of second order for a 3D object model. To calculate the mean of the points of the 3D object model, select *'mean_points'* in MomentsToCalculate. If instead the central moment of second order should be calculated, select *'central_moment_2_points'*. The results are the variances of the x, y, z, x-y, x-z, and y-z axes. To compute the three principal axes of the 3D object model select *'principal_axes'* in MomentsToCalculate. The result is a pose with the mean of the points as center. The coordinate system that corresponds to the pose has the x-axis along the first principal axis, the y-axis along the second principal axis and the z-axis along the third principal axis.

Parameters

▷ **ObjectModel3D** (input_control) ..... object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **MomentsToCalculate** (input_control) ..... number(-array) ⇝ *string*
Moment to calculate.
Default: 'mean_points'
List of values: MomentsToCalculate ∈ {'mean_points', 'central_moment_2_points', 'principal_axes'}

▷ **Moments** (output_control) ..... number(-array) ⇝ *real*
Calculated moment.
Number of elements: Moments == ObjectModel3D

Example

```
gen_object_model_3d_from_points (rand(200), rand(200),\
                                 rand(200), ObjectModel3D)
moments_object_model_3d (ObjectModel3D, ['mean_points',\
                         'central_moment_2_points','principal_axes'], \
                         Moments)
```

Result

moments_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_object_model_3d, connection_object_model_3d

Possible Successors

project_object_model_3d, object_model_3d_to_xyz, select_object_model_3d

See also

volume_object_model_3d_relative_to_plane

Module

3D Metrology

select_object_model_3d (: : ObjectModel3D, Feature, Operation, MinValue, MaxValue : ObjectModel3DSelected)

Select 3D object models from an array of 3D object models according to global features.

select_object_model_3d selects 3D object models from an array of 3D object models for which the values of specified global features lie within a specified range. The list of possible features that may be specified in Feature are:

'mean_points_x': The mean x-coordinate of the points in the 3D object model.

'mean_points_y': The mean y-coordinate of the points in the 3D object model.

'mean_points_z': The mean z-coordinate of the points in the 3D object model.

'diameter_axis_aligned_bounding_box': The diameter of the set of 3D points, defined as the length of the diagonal of the smallest enclosing axis-parallel cuboid.

'diameter_bounding_box': The diameter of the set of 3D points, defined as the length of the diagonal of the smallest enclosing oriented cuboid. This feature has a high calculation complexity.

'diameter_object': The diameter of the set of 3D points, defined as the length of the distance between two points.

'volume': The volume of the triangulation of the 3D object model over the x-y plane in the coordinate origin. This corresponds to the default parametrization of volume_object_model_3d_relative_to_plane with the plane [0,0,0,0,0,0,0]. The plane con not be changed here.

'volume_axis_aligned_bounding_box': The volume of the smallest enclosing axis-parallel cuboid.

'area': The area of the triangulation of the 3D object model.

'central_moment_2_x': The x-value of the second central moment of the 3D object model.

'central_moment_2_y': The y-value of the second central moment of the 3D object model.

'central_moment_2_z': The z-value of the second central moment of the 3D object model.

'central_moment_2_xy': The xy-value of the second central moment of the 3D object model.

'central_moment_2_xz': The xz-value of the second central moment of the 3D object model.

'central_moment_2_yz': The yz-value of the second central moment of the 3D object model.

'num_points': The number of points.

'num_triangles': The number of triangles.

'num_faces': The number of faces.

'num_lines': The number of polylines.

'has_points': The existence of 3D points.

'has_point_normals': The existence of 3D point normals.

'has_triangles': The existence of triangles.

'has_faces': The existence of faces or polygons.

'has_lines': The existence of lines.

'has_xyz_mapping': The existence of a mapping of the 3D points to image coordinates.

'has_shape_based_matching_3d_data': The existence of a shape model for shape-based 3D matching.

'has_surface_based_matching_data': The existence of data for the surface-based 3D matching.

'has_segmentation_data': The existence of data for a 3D segmentation.

'has_primitive_data': The existence of a 3D primitive.

For all features listed in Feature a minimal and maximal threshold must be specified in MinValue and MaxValue. This range is then used to select all given 3D object models that fulfill the given conditions. These are copied to ObjectModel3DSelected. For logical parameters (e.g., *'has_points'*, *'has_point_normals'*, ...), MinValue and MaxValue can both be set to *'true'* to select all 3D object models that have the respective attribute or to *'false'* to select all that do not have it. MinValue and MaxValue can be set to *'min'* and *'max'* accordingly to ignore the respective threshold.

The parameter Operation defines the logical operation that is used to combine different features in Feature. It can be either a logical *'or'* or *'and'*.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handles of the available 3D object models to select.

▷ **Feature** (input_control) . . . . . string(-array) ⇝ *string*
List of features a test is performed on.
Default: 'has_triangles'
List of values: Feature ∈ {'mean_points_x', 'mean_points_y', 'mean_points_z', 'volume', 'volume_axis_aligned_bounding_box', 'central_moment_2_x', 'central_moment_2_y', 'central_moment_2_z', 'central_moment_2_xy', 'central_moment_2_xz', 'central_moment_2_yz', 'diameter_axis_aligned_bounding_box', 'diameter_bounding_box', 'diameter_object', 'area', 'has_points', 'has_triangles', 'has_faces', 'has_lines', 'has_xyz_mapping', 'has_point_normals', 'has_shape_based_matching_3d_data', 'has_surface_based_matching_data', 'has_segmentation_data', 'has_primitive_data', 'num_points', 'num_triangles', 'num_faces', 'num_lines'}

▷ **Operation** (input_control) . . . . . string ⇝ *string*
Logical operation to combine the features given in Feature.
Default: 'and'
List of values: Operation ∈ {'and', 'or'}

▷ **MinValue** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Minimum value for the given feature.
Default: 1
Suggested values: MinValue ∈ {0, 1, 100, 0.1, 'true', 'false', 'min'}

▷ **MaxValue** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Maximum value for the given feature.
Default: 1
Suggested values: MaxValue ∈ {0, 1, 10, 100, 0.1, 'true', 'false', 'max'}

▷ **ObjectModel3DSelected** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
A subset of ObjectModel3D fulfilling the given conditions.

Example

```
gen_object_model_3d_from_points (rand(20)-1.0, rand(20)-1.0,\
                                 rand(20)-1.0, ObjectModel3D1)
gen_object_model_3d_from_points (rand(20), rand(20),\
                                 rand(20), ObjectModel3D2)
select_object_model_3d ([ObjectModel3D1, ObjectModel3D2],\
                        'mean_points_x', 'and', 0, 1, ObjectModel3DSelected)
```

Result

select_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_object_model_3d, select_points_object_model_3d,

connection_object_model_3d, get_object_model_3d_params, volume_object_model_3d_relative_to_plane, area_object_model_3d, max_diameter_object_model_3d, moments_object_model_3d

Possible Successors

project_object_model_3d, object_model_3d_to_xyz

See also

volume_object_model_3d_relative_to_plane, area_object_model_3d, max_diameter_object_model_3d, moments_object_model_3d, get_object_model_3d_params

Module

3D Metrology

smallest_bounding_box_object_model_3d (: : ObjectModel3D, Type : Pose, Length1, Length2, Length3)

Calculate the smallest bounding box around the points of a 3D object model.

smallest_bounding_box_object_model_3d calculates the smallest bounding box around the points of a 3D object model. The resulting bounding box is described using its coordinate system (Pose), which is oriented such that the longest side of the box is aligned with the x-axis, the second longest side is aligned with the y-axis and the smallest side is aligned with the z-axis. The lengths of the sides are returned in Length1, Length2, and Length3, in descending order. The box can be either axis-aligned or oriented, which can be chosen by the Type. The algorithm for *'oriented'* is computationally significantly more costly than the algorithm for *'axis_aligned'*, and returns only an approximation of the oriented bounding box. Note that the algorithm for the oriented bounding box is randomized and can return a different box for each call.

In order to retrieve the corners of the *'axis_aligned'* box, the operator get_object_model_3d_params can be used with the parameter *'bounding_box1'*.

Parameters

- ▷ **ObjectModel3D** (input_control) ..... object_model_3d(-array) ⇝ *handle*
  Handle of the 3D object model.
- ▷ **Type** (input_control) ..... string ⇝ *string*
  The method that is used to estimate the smallest box.
  Default: 'oriented'
  List of values: Type ∈ {'oriented', 'axis_aligned'}
- ▷ **Pose** (output_control) ..... pose(-array) ⇝ *real* / integer
  The pose that describes the position and orientation of the box that is generated. The pose has its origin in the center of the box and is oriented such that the x-axis is aligned with the longest side of the box.
- ▷ **Length1** (output_control) ..... number(-array) ⇝ *real*
  The length of the longest side of the box.
  Number of elements: Length1 == ObjectModel3D
- ▷ **Length2** (output_control) ..... number(-array) ⇝ *real*
  The length of the second longest side of the box.
  Number of elements: Length2 == ObjectModel3D
- ▷ **Length3** (output_control) ..... number(-array) ⇝ *real*
  The length of the third longest side of the box.
  Number of elements: Length3 == ObjectModel3D

Example

```
gen_object_model_3d_from_points (rand(20), rand(20), rand(20), \
                                 ObjectModel3D)
smallest_bounding_box_object_model_3d (ObjectModel3D, 'oriented', \
                                       Pose, Length1, Length2, Length3)
gen_box_object_model_3d (Pose, Length1, Length2, Length3, ObjectModel3D1)
dev_get_window (WindowHandle)
visualize_object_model_3d (WindowHandle, [ObjectModel3D,ObjectModel3D1], \
                           [], [], ['alpha_1'], [0.5], [], [], [], PoseOut)
```

Result

`smallest_bounding_box_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`connection_object_model_3d`, `simplify_object_model_3d`

Possible Successors

`gen_box_object_model_3d`

See also

`smallest_sphere_object_model_3d`

Module

3D Metrology

`smallest_sphere_object_model_3d` `( : : ObjectModel3D : CenterPoint, Radius )`

Calculate the smallest sphere around the points of a 3D object model.

`smallest_sphere_object_model_3d` calculates the smallest sphere around the points of the 3D object model given by `ObjectModel3D`. The resulting center will be stored as x-, y-, and z-coordinates in `CenterPoint` as 3 values representing X, Y, and Z. The Radius is given in `Radius`.

Parameters

▷ **`ObjectModel3D`** (input_control) ..... object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **`CenterPoint`** (output_control) ..... number-array ⇝ *real*
x-, y-, and z-coordinates describing the center point of the sphere.
Number of elements: CenterPoint == 3 * ObjectModel3D

▷ **`Radius`** (output_control) ..... number(-array) ⇝ *real*
The estimated radius of the sphere.
Number of elements: Radius == ObjectModel3D

Example

```
gen_object_model_3d_from_points (rand(20), rand(20), rand(20),\
                                 ObjectModel3D)
smallest_sphere_object_model_3d(ObjectModel3D, CenterPoint, Radius)
gen_sphere_object_model_3d_center (CenterPoint[0], CenterPoint[1], \
                                   CenterPoint[2], Radius, ObjectModel3D1)
dev_get_window (WindowHandle)
visualize_object_model_3d (WindowHandle, [ObjectModel3D,ObjectModel3D1], \
                           [], [], ['alpha_1'], [0.5], [], [], [], PoseOut)
```

Result

`smallest_sphere_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

connection_object_model_3d

Possible Successors

gen_sphere_object_model_3d

See also

smallest_bounding_box_object_model_3d

Module

3D Metrology

volume_object_model_3d_relative_to_plane (: : ObjectModel3D, Plane, Mode, UseFaceOrientation : Volume)

Calculate the volume of a 3D object model.

volume_object_model_3d_relative_to_plane calculates the volume under the faces of a 3D object model relative to a plane. The plane is defined by the x-y plane of the pose given in Plane.

For ObjectModel3D, a triangulation or a list of polygons must be available. With default settings, if the mesh is watertight and ordered, the operator calculates the actual volume of the 3D object model. To also cover cases where the mesh is not closed or the faces are not ordered consistently, the calculation of the volume can be influenced with the parameters Mode and UseFaceOrientation.

How the volume is calculated:

First, the operator calculates the volume of the prisms that are constructed by projecting each face onto the plane.

The individual volumes of the prisms can be positive or negative depending on the orientation of the face (away or towards the plane) or the location of the face (above or below the plane). This can be controlled with the parameter UseFaceOrientation.

After that, the volumes of the prisms are added up depending on the parameter Mode.

The volume returned in Volume is the absolute value of the calculated sum.

How to set the parameters:

Mode can be set to the following options:

'signed' (default) The volumes above and below the plane are added.

'unsigned' The volume below the plane is subtracted from the volume above the plane.

'positive' Only faces above the plane are taken into account.

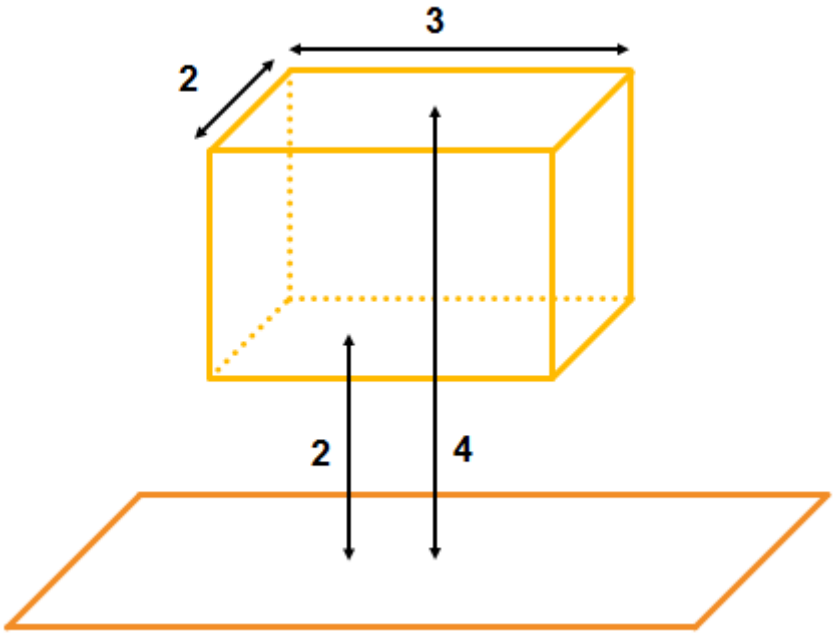
'negative' Only faces below the plane are taken into account.

UseFaceOrientation can be set to the following options:

'true' (default) Use the orientation of the faces relative to the plane. A face points away from the plane if the corner points are ordered clockwise when viewed from the plane. The volume under a face is considered **positive** if the orientation of the face is **away from the plane**. In contrast, it is considered **negative** if the orientation of the face is **towards the plane**.

'false' The volume under a face is considered **positive** if the face is located **above** the plane. In contrast, it is considered **negative** if the face is located **below** the plane.

For example, with the default combination (Mode: *'signed'*, UseFaceOrientation: *'true'*), you can approximate the real volume of a closed object. In this case, the Plane is still required, but does not change the resulting volume.



Example: (A) `Mode`: *'signed'*, `UseFaceOrientation`: *'true'*: $V = (2 * 3 * 4) + (-2 * 3 * 2) = 24 - 12 = 12$ (B) `Mode`: *'signed'*, `UseFaceOrientation`: *'false'*: $V = (2 * 3 * 4) + (2 * 3 * 2) = 24 + 12 = 36$ (C) `Mode`: *'negative'*: $V = 0$

Attention

The calculation of the volume might be numerically unstable in case of a large distance between the plane and the object (approx. distance > 10000 times the object diameter).

Parameters

- ▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
  Handle of the 3D object model.
- ▷ **`Plane`** (input_control) . . . . . pose(-array) ⇝ *real* / integer
  Pose of the plane.
  Default: [0,0,0,0,0,0,0]
- ▷ **`Mode`** (input_control) . . . . . string(-array) ⇝ *string*
  Method to combine volumes laying above and below the reference plane.
  Default: 'signed'
  List of values: `Mode` ∈ {'positive', 'negative', 'unsigned', 'signed'}
- ▷ **`UseFaceOrientation`** (input_control) . . . . . string(-array) ⇝ *string*
  Decides whether the orientation of a face should affect the resulting sign of the underlying volume.
  Default: 'true'
  List of values: `UseFaceOrientation` ∈ {'true', 'false'}
- ▷ **`Volume`** (output_control) . . . . . number(-array) ⇝ *real*
  Absolute value of the calculated volume.
  Number of elements: Volume == ObjectModel3D

Example

```
gen_box_object_model_3d ([0,0,0,0,0,0,0],3,2,1, ObjectModel3D)
convex_hull_object_model_3d (ObjectModel3D, ObjectModel3DConvexHull)
volume_object_model_3d_relative_to_plane (ObjectModel3DConvexHull,\
                                          [0,0,0,0,0,0,0], 'signed',\
                                          'true', Volume)
```

Result

`volume_object_model_3d_relative_to_plane` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, select_points_object_model_3d

Possible Successors

project_object_model_3d, object_model_3d_to_xyz, select_object_model_3d

See also

area_object_model_3d

Module

3D Metrology

4.3 Segmentation

fit_primitives_object_model_3d (: : ObjectModel3D, GenParamName, GenParamValue : ObjectModel3DOut)

Fit 3D primitives into a set of 3D points.

The operator fit_primitives_object_model_3d fits a 3D primitive, i.e., a simple 3D shape, into a set of 3D points given by a 3D object model with the handle ObjectModel3D. The shapes that are available as 3D primitives comprise a cylinder, a sphere, and a plane. As the operator does not perform a segmentation of the set of 3D points that is contained in the input 3D object model, you have to make sure that the contained 3D points already correspond to a 3D primitive. A segmentation can be performed, e.g., with the operator segment_object_model_3d.

fit_primitives_object_model_3d returns the handle ObjectModel3DOut for the output 3D object model, which contains information that concern, e.g., the type and parameters of the fitted 3D primitive. This information can be queried from the 3D object model with get_object_model_3d_params. Note that the extent of primitives of the type plane and cylinder can be queried with get_object_model_3d_params, as well.

The parameters of a cylinder are the (x-, y-, z-)coordinates of the center, the normed (x-, y-, z-)directions of the main axis of the cylinder, and the radius of the cylinder. The center does not necessarily lie in the center of gravity of the cylinder (see the explanation of the parameters MinExtent and MaxExtent of the operator gen_cylinder_object_model_3d). The sign of the main axis is determined such that the main axis points towards the half space in which the origin is located. For a sphere the parameters are the (x-, y-, z-)coordinates of the center and the radius of the sphere. A plane is given by the 4 parameters of the hessian normal form, i.e., the unit normal (x-, y-, z-) vector and the orthogonal distance of the plane from the origin of the coordinate system. The sign of the hessian normal form is determined such that the normal vector points towards the side of the plane on which the origin is located and the distance is not positive.

If no primitive can be fitted to the set of 3D points, the returned object model will not contain a primitive. However, depending on the parameter values for *'output_point_coord'* and *'output_xyz_mapping'* (see below), the returned object model is either empty, or contains the 3D points, or contains the 3D points and the mapping from the 3D points to image coordinates of the input object model ObjectModel3D.

To control the fitting, you can adjust some generic parameters within GenParamName and GenParamValue. But note that for a lot of applications the default values are sufficient and no adjustment is necessary. The following values for GenParamName and GenParamValue are possible:

'primitive_type': The parameter specifies which type of 3D primitive should be fitted into the set of 3D points. You can specify a specific primitive type by setting *'primitive_type'* to *'cylinder'*, *'sphere'*, or *'plane'*. Then, only the selected type of 3D primitive is fitted into the set of 3D points. You can also specify a set of specific 3D primitives that should be fitted by setting *'primitive_type'* to a tuple consisting of different primitive types. If all types of 3D primitives should be fitted, you can set *'primitive_type'* to *'all'*. Note that if more than one

primitive type is selected, only the best fitting 3D primitive, i.e., the 3D primitive with the smallest quadratic residual error, is returned.

List of values: *'cylinder'*, *'sphere'*, *'plane'*, *'all'*

Default: *'cylinder'*

'fitting_algorithm': The parameter specifies the used algorithm for the fitting of the 3D primitive. When fitting a plane, the results are identical for the different algorithms. If *'fitting_algorithm'* is set to *'least_squares'*, the approach minimizes the quadratic distance between the 3D points and the resulting primitive. If *'fitting_algorithm'* is set to *'least_squares_huber'*, the approach is similar to *'least_squares'*, but the points are weighted to decrease the impact of outliers based on the approach of Huber (see below). If *'fitting_algorithm'* is set to *'least_squares_tukey'*, the approach is also similar to *'least_squares'*, but the points are weighted and outliers are ignored based on the approach of Tukey (see below).

For *'least_squares_huber'* and *'least_squares_tukey'* a robust error statistics is used to estimate the standard deviation of the distances from the object points without outliers from the fitting primitive. The Tukey algorithm removes outliers, whereas the Huber algorithm only damps them, or more precisely, weights them linearly. In practice, the approach of Tukey is recommended.

List of values: *'least_squares'*, *'least_squares_huber'*, *'least_squares_tukey'*

Default: *'least_squares'*

'min_radius': The parameter specifies the minimum radius of a cylinder or a sphere. If a cylinder or a sphere with a smaller radius is fitted, the resulting 3D object model is empty. The parameter is ignored when fitting a plane. The unit is meter.

Suggested values: *0.01*, *0.02*, *0.1*

Default: *0.01*

'max_radius': The parameter specifies the maximum radius of a cylinder or a sphere. If a cylinder or a sphere with a larger radius is fitted, the resulting 3D object model is empty. The parameter is ignored when fitting a plane. The unit is meter.

Suggested values: *0.02*, *0.04*, *0.2*

Default: *0.2*

'output_point_coord': The parameter determines if the 3D points used for the fitting are copied to the output 3D object model. If *'copy_point_coord'* is set to *'true'*, the 3D points are copied. If *'copy_point_coord'* is set to *'false'*, no 3D points are copied.

List of values: *'true'*, *'false'*

Default: *'true'*

'output_xyz_mapping': The parameter determines if a mapping from the 3D points to image coordinates is copied to the output 3D object model. This information is needed, e.g., when using the operator `object_model_3d_to_xyz` after the fitting (e.g., for a visualization). If *'output_xyz_mapping'* is set to *'true'*, the image coordinate mapping is copied. Note that the parameter is only valid, if the image coordinate mapping is available in the input 3D object model. Make sure that, if you derive the input 3D object model by copying it with the operator `copy_object_model_3d` from a 3D object model that contains such a mapping, the mapping is copied, too. Furthermore, the parameter is only valid, if the 3D points are copied to the output 3D object model, which is set with the parameter *'output_point_coord'*.

List of values: *'true'*, *'false'*

Default: *'false'*

The minimum number of 3D points that are necessary to fit a plane is three. The minimum number of 3D points that is necessary to fit a sphere is four. The minimum number of 3D points that is necessary to fit a cylinder is five.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the input 3D object model.

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Number of elements: GenParamName == GenParamValue
List of values: `GenParamName` ∈ {'primitive_type', 'fitting_algorithm', 'min_radius', 'max_radius', 'output_point_coord', 'output_xyz_mapping'}

▷ **GenParamValue** (input_control) . . . . . attribute.name-array ⇝ *string* / real / integer
Values of the generic parameters.
Number of elements: GenParamValue == GenParamName
Suggested values: GenParamValue ∈ {'cylinder', 'sphere', 'plane', 'all', 'least_squares', 'least_squares_huber', 'least_squares_tukey', 0.01, 0.05, 0.1, 0.2, 'true', 'false'}

▷ **Object3DOut** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the output 3D object model.

Result

fit_primitives_object_model_3d returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

xyz_to_object_model_3d, read_object_model_3d

Possible Successors

get_object_model_3d_params, object_model_3d_to_xyz, write_object_model_3d, clear_object_model_3d

Alternatives

segment_object_model_3d

Module

3D Metrology

reduce_object_model_3d_by_view (Region : : ObjectModel3D, CamParam, Pose : ObjectModel3DReduced)

Remove points from a 3D object model by projecting it to a virtual view and removing all points outside of a given region.

reduce_object_model_3d_by_view projects the points of ObjectModel3D into the image plane given by Pose and CamParam and reduces the 3D object model to the points lying inside the region given in Region. In particular, the points are first transformed with the pose and then projected using the camera parameters. Only those points that are located inside the specified region are copied to the new 3D object model.

Faces of a mesh are only contained in the output 3D object model if all corner points are within the region.

As alternative to camera parameters and a pose, an XYZ-mapping contained in ObjectModel3D can be used for the reduction. For this, CamParam must be set to *'xyz_mapping'* or an empty tuple and an empty tuple must be passed to Pose. In this case, the original image coordinates of the 3D points are used to check if a point is inside Region.

Attention

Cameras with hypercentric lenses are not supported.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Region in the image plane.

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **CamParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.
Suggested values: CamParam ∈ {'xyz_mapping', []}

▷ **Pose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
3D pose of the world coordinate system in camera coordinates.
Number of elements: Pose == 7

▷ **ObjectModel3DReduced** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the reduced 3D object model.

Example

```
gen_object_model_3d_from_points (200*(rand(100)-0.5), \
                                 200*(rand(100)-0.5), \
                                 200*(rand(100)-0.5), ObjectModel3D)
gen_circle (Circle, 240, 320, 60)
CamParam := ['area_scan_telecentric_division',1,0,1,1,320,240,640,480]
Pose := [0,0,1,0,0,0,0]
reduce_object_model_3d_by_view (Circle, ObjectModel3D, CamParam, \
                                Pose, ObjectModel3DReduced)
dev_get_window (WindowHandle)
visualize_object_model_3d (WindowHandle, [ObjectModel3D, \
                           ObjectModel3DReduced], CamParam, Pose, \
                           ['color_0', 'point_size_1'], ['blue',6], \
                           [], [], [], PoseOut)
```

Result

`reduce_object_model_3d_by_view` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_object_model_3d`, `xyz_to_object_model_3d`

Possible Successors

`project_object_model_3d`, `object_model_3d_to_xyz`

See also

`select_points_object_model_3d`

Module

3D Metrology

segment_object_model_3d (: : ObjectModel3D, GenParamName, GenParamValue : ObjectModel3DOut)

Segment a set of 3D points into sub-sets with similar characteristics.

The operator `segment_object_model_3d` segments a set of 3D points given by a 3D object model with the handle `ObjectModel3D` into several sub-sets of neighbored 3D points with similar characteristics like the same normal orientation or curvature. By default, the operator then tries to fit a 3D primitive, i.e., a simple 3D shape like a plane, a sphere, or a cylinder, into each of these sub-sets. As result, the operator returns a tuple of handles for the 3D object models that represent the individual sub-sets of 3D points (`ObjectModel3DOut`). Within these 3D object models information is stored that concern, e.g., the success of the fitting and the type and parameters of the fitted 3D primitive. This information can be queried from the individual 3D object model with `get_object_model_3d_params`.

Before calling `segment_object_model_3d`, the input 3D object model should be prepared for the segmentation using the operator `prepare_object_model_3d` with the parameter `Purpose` set to *'segmentation'*. If the input 3D object model is not prepared this way, the operator `prepare_object_model_3d` is called internally within `segment_object_model_3d` to extend the 3D object model with attributes that were not explicitly but only implicitly contained in the 3D object model.

To control the segmentation and the fitting, you can adjust some generic parameters within GenParamName and GenParamValue. But note that for a lot of applications the default values are sufficient and no adjustment is necessary. The following values for GenParamName and GenParamValue are possible:

'max_orientation_diff': The parameter specifies the maximum angle between the point normals of two neighbored 3D points (in radians) that is allowed so that the two points belong to the same sub-set of 3D points. For a cylinder or sphere, the parameter value depends on the dimension of the object and on the distance of the neighbored 3D points. I.e., if the cylinder or sphere has a very small radius or if the 3D points are not very dense, the value must be chosen higher. For a plane the value is independent from the dimension of the object and can be set to a small value.
Suggested values: *0.10*, *0.15*, *0.20*
Default: *0.15*

'max_curvature_diff': The parameter specifies the maximum difference between the curvatures of the surface at the positions of two neighbored 3D points that is allowed so that the two points belong to the same sub-set of 3D points. The value depends on the noise of the 3D points. I.e., if the noise level of the 3D points is very high, the value must be set to a higher value. Generally, the number of resulting 3D object models decreases for a higher value, because more 3D points are merged to a sub-set of 3D points.
Suggested values: *0.03*, *0.04*, *0.05*
Default: *0.05*

'min_area': The parameter specifies the minimum number of 3D points needed for a sub-set of connected 3D points to be returned by the segmentation. Thus, for a sub-set with fewer points the points are deleted and no output handle is created.
Suggested values: *1*, *10*, *100*
Default: *100*

'fitting': The parameter specifies whether after the segmentation 3D primitives are fitted into the sub-sets of 3D points. If *'fitting'* is set to *'true'*, which is the default, the fitting is calculated and the 3D object models with the resulting handles contain the parameters of the corresponding 3D primitives. The output parameters of a cylinder, a sphere, or a plane are described with the operator fit_primitives_object_model_3d. If *'fitting'* is set to *'false'*, only a segmentation is performed and the output 3D object models contain the segmented sub-sets of 3D points. A later fitting can be performed with the operator fit_primitives_object_model_3d.
List of values: *'false'*, *'true'*
Default: *'true'*

'output_xyz_mapping': The parameter determines if a mapping from the segmented 3D points to image coordinates is copied to the output 3D object model. This information is needed, e.g., when using the operator object_model_3d_to_xyz after the segmentation (e.g., for a visualization). If *'output_xyz_mapping'* is set to *'true'*, the image coordinate mapping is copied. Note that the parameter is only valid, if the image coordinate mapping is available in the input 3D object model. Make sure that, if you derive the input 3D object model by copying it with the operator copy_object_model_3d from a 3D object model that contains such a mapping, the mapping is copied, too. Furthermore, the parameter is only valid, if the 3D points are copied to the output 3D object model, which is set with the parameter *'output_point_coord'*. If *'output_xyz_mapping'* is set to *'false'*, the image coordinate mapping is not copied.
List of values: *'true'*, *'false'*
Default: *'false'*

'primitive_type', *'fitting_algorithm'*, *'min_radius'*, *'max_radius'*, *'output_point_coord'*: These parameters are used, if *'fitting'* is set to *'true'*, which is the default. The meaning and the use of these parameters is described with the operator fit_primitives_object_model_3d.

'surface_check': The parameter determines whether the surface of a triangulated input object model is checked regarding its conformity to the expected requirements. If the input 3D object model contains triangles that are topologically invalid an error message is raised. If the triangulation was created (triangulate_object_model_3d) or edited (e.g., by simplify_object_model_3d) by a HALCON operator, a surface check should not be necessary. The check can be disabled in order to enhance the runtime by setting *'surface_check'* to *'false'*.
List of values: *'true'*, *'false'*
Default: *'true'*

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the input 3D object model.

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters.
Number of elements: GenParamName == GenParamValue
List of values: GenParamName ∈ {'max_orientation_diff', 'max_curvature_diff', 'min_area', 'primitive_type', 'fitting_algorithm', 'min_radius', 'max_radius', 'output_point_coord', 'output_xyz_mapping', 'surface_check'}

▷ **GenParamValue** (input_control) . . . . . attribute.name-array ⇝ *string* / real / integer
Values of the generic parameters.
Number of elements: GenParamValue == GenParamName
Suggested values: GenParamValue ∈ {0.15, 0.05, 100, 'true', 'false', 'cylinder', 'sphere', 'plane', 'all', 'least_squares', 'least_squares_huber', 'least_squares_tukey'}

▷ **ObjectModel3DOut** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the output 3D object model.

Result

segment_object_model_3d returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

xyz_to_object_model_3d, read_object_model_3d, prepare_object_model_3d

Possible Successors

get_object_model_3d_params, object_model_3d_to_xyz, write_object_model_3d, clear_object_model_3d

See also

fit_primitives_object_model_3d

Module

3D Metrology

select_points_object_model_3d (: : ObjectModel3D, Attrib, MinValue, MaxValue : ObjectModel3DThresholded)

Apply a threshold to an attribute of 3D object models.

select_points_object_model_3d selects points of the 3D object model ObjectModel3D according to the attributes and thresholds passed in Attrib, MinValue, and MaxValue respectively. The selected points are returned in the 3D object model ObjectModel3DThresholded. All attributes that are connected with the points (e.g., polygons or triangles) are adapted in such a way that there is no reference to the removed points left.

Attrib can either contain a tuple of numbers that has the same length as ObjectModel3D has points, or a list of attribute names on which the thresholds are applied.

If Attrib contains a tuple of numbers, exactly one number must be passed in both MinValue and MaxValue. All points for which the corresponding entry in Attrib is between the two thresholds are added to the output 3D object model ObjectModel3DThresholded.

Otherwise, Attrib can contain a list of attribute names that refer to properties of the 3D object model ObjectModel3D. All points, for which the value stored in the attribute Attrib is inside the interval specified in MinValue and MaxValue are stored in the output 3D object model. MinValue and MaxValue must contain exactly as many values as Attrib. If Attrib contains multiple values, only those points are stored in the output 3D object model that fulfill all the criteria.

Depending on the properties of `ObjectModel3D`, the following values are possible for `Attrib`:
The following attributes are available:

'point_coord_x': The x-coordinates of the set of 3D points.

'point_coord_y': The y-coordinates of the set of 3D points.

'point_coord_z': The z-coordinates of the set of 3D points.

'point_normal_x': The x-components of the 3D point normals of the set of 3D points.

'point_normal_y': The y-components of the 3D point normals of the set of 3D points.

'point_normal_z': The z-components of the 3D point normals of the set of 3D points.

'mapping_row': The row-components of the 2D mapping of the set of 3D points.

'mapping_col': The column-components of the 2D mapping of the set of 3D points.

'neighbor_distance':

'neighbor_distance N': The distance of the N-th nearest point. N must be a positive integer and is by default 25. For every point, all other points are sorted according to their distance and the distance of the N-th point is used.

'num_neighbors X': The number of neighbors within a distance of at most X. It can be used to remove sparsely populated parts of the 3D object model, such as outliers or points that are created by smoothing between 3D surfaces.

'num_neighbors_fast X': The approximate number of neighbors within a distance of at most X. The distances are approximated using voxels, leading to a faster processing compared to *'num_neighbors'*.

Extended attribute: Enter the name of an extended attribute of the type *'vertices'* and the selection will be applied based on the values of the extended attribute.

Parameters

▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object models.

▷ **`Attrib`** (input_control) . . . . . string(-array) ⇝ *string*
Attributes the threshold is applied to.
Default: 'point_coord_z'
List of values: `Attrib` ∈ {'point_coord_x', 'point_coord_y', 'point_coord_z', 'point_normal_x', 'point_normal_y', 'point_normal_z', 'mapping_row', 'mapping_col', 'neighbor_distance', 'num_neighbors', 'num_neighbors_fast'}

▷ **`MinValue`** (input_control) . . . . . number(-array) ⇝ *real* / integer
Minimum value for the attributes specified by `Attrib`.
Default: 0.5

▷ **`MaxValue`** (input_control) . . . . . number(-array) ⇝ *real* / integer
Maximum value for the attributes specified by `Attrib`.
Default: 1.0

▷ **`ObjectModel3DThresholded`** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the reduced 3D object models.

Example

```
gen_object_model_3d_from_points (rand(100), rand(100),\
                                 rand(100), ObjectModel3D)
select_points_object_model_3d (ObjectModel3D, 'point_coord_z',\
                               0.5, 1, ObjectModel3DThresholded)
get_object_model_3d_params (ObjectModel3DThresholded, 'num_points',\
                            NumPoints)
```

Result

`select_points_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised. If the required points are missing in the object model, i.e., an empty object model is passed, the error 9515 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d

Possible Successors

connection_object_model_3d, project_object_model_3d, object_model_3d_to_xyz

See also

connection_object_model_3d, reduce_object_model_3d_by_view

Module

3D Metrology

4.4 Transformations

affine_trans_object_model_3d (: : ObjectModel3D, HomMat3D : ObjectModel3DAffineTrans)

Apply an arbitrary affine 3D transformation to 3D object models.

affine_trans_object_model_3d applies arbitrary affine 3D transformations, i.e., scaling, rotation, and translation, to 3D object models and returns the handles of the transformed 3D object models. The affine transformations are described by the homogeneous transformation matrices given in HomMat3D.

The transformation matrices can be created using the operators hom_mat3d_identity, hom_mat3d_scale, hom_mat3d_rotate, hom_mat3d_translate, etc., or it can be the result of pose_to_hom_mat3d (see affine_trans_point_3d).

In general, the operator affine_trans_object_model_3d is not necessary in the context of shape based 3D matching. Instead, if a rotation of the 3D object model into a reference orientation should be performed, appropriate values for the parameters RefRotX, RefRotY, RefRotZ, and OrderOfRotation should be passed to the operator create_shape_model_3d.

affine_trans_object_model_3d transforms one or more 3D object models with the same transformation matrix if only one transformation matrix is passed in HomMat3D (N:1). If a single 3D object model is passed in ObjectModel3D, it is transformed with all passed transformation matrices (1:N). If the number of transformation matrices corresponds to the number of 3D object models, every 3D object model is transformed individually with the respective transformation matrix (N:N). In those cases, N can be zero, i.e., no matrix or no 3D object model can be passed to the operator. In this case, an empty tuple is returned in ObjectModel3DAffineTrans. This can be used to, for example, transform the results of other operators without checking first if at least one matrix was returned.

Attention

affine_trans_object_model_3d transforms the attributes of type 3D points, 3D point normals, and the prepared shape model for shape-based 3D matching. Primitives and precomputed data structures for 3D distance computation are not copied. All other attributes are copied without modification. To transform 3D primitives, the operator rigid_trans_object_model_3d must be used.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handles of the 3D object models.

▷ **HomMat3D** (input_control) . . . . . hom_mat3d(-array) ⇝ *real*
Transformation matrices.

▷ **ObjectModel3DAffineTrans** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handles of the transformed 3D object models.

Result

affine_trans_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d

Possible Successors

project_object_model_3d, object_model_3d_to_xyz

See also

affine_trans_point_3d, rigid_trans_object_model_3d, projective_trans_object_model_3d

Module

3D Metrology

connection_object_model_3d (: : ObjectModel3D, Feature, Value : ObjectModel3DConnected)

Determine the connected components of the 3D object model.

connection_object_model_3d determines the connected components of the input 3D object model given in ObjectModel3D. The decision if two parts of the 3D object model are connected can be based on different attributes and respective distance functions. The attribute and distance function can be selected in Feature:

'distance_3d': The euclidean distance between the point coordinates of the set of the 3D points are tested. For any distance below Value the points are considered as connected.

'angle': The angles between the normals of the points in the 3D object model are compared. Similar normals are considered as connected if their angular distance is below Value. Value is specified in radians and should be between 0 and pi.

Prerequisite: The 3D object model must contain normals, which can be computed with surface_normals_object_model_3d.

'distance_mapping': The mapping measures the distance between the pixel coordinates of points in the 3D object model that are stored in the 2D mapping. Use a value larger than 1.5 for Value to get a connection in an 8-neighborhood in the image.

Prerequisite: The 3D object model must contain a 2D mapping, which is available if the 3D object model has been created with xyz_to_object_model_3d.

'mesh': Returns parts of the 3D object model that are connected with triangles or polygons. Value is ignored.

Prerequisite: The 3D object model must provide a triangulation, which can be obtained with triangulate_object_model_3d. Alternatively, if the 3D object model already contains a 2D mapping, prepare_object_model_3d can be used with Purpose set to *'segmentation'* to quickly triangulate the 3D object model.

'lines': Returns parts of the object model that are connected by lines. Value is ignored.

Prerequisite: The 3D object model must contain polylines, which can be computed with intersect_plane_object_model_3d.

Alternatively, the required attributes can be set manually with set_object_model_3d_attrib or set_object_model_3d_attrib_mod. Note that the 3D object model might already contain the required attribute, especially if the 3D object model has been read with read_object_model_3d or if it has been deserialized with deserialize_object_model_3d. To check whether the required attribute is available, use get_object_model_3d_params.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **Feature** (input_control) . . . . . string(-array) ⇝ *string*
Attribute used to calculate the connected components.
Default: 'distance_3d'
List of values: Feature ∈ {'distance_3d', 'angle', 'distance_mapping', 'mesh', 'lines'}

▷ **Value** (input_control) . . . . . number(-array) ⇝ *real* / integer
Maximum value for the distance between two connected components.
Default: 1.0
Suggested values: Value ∈ {1.0, 1.1, 1.5, 10.0, 100.0}

▷ **ObjectModel3DConnected** (output_control) . . . . . object_model_3d-array ⇝ *handle*
Handle of the 3D object models that represent the connected components.

Example

```
gen_object_model_3d_from_points (rand(100), rand(100),\
                                 rand(100), ObjectModel3D)
connection_object_model_3d (ObjectModel3D, 'distance_3d', 0.2,\
                            ObjectModel3DConnected)
dev_get_window (WindowHandle)
visualize_object_model_3d (WindowHandle, [ObjectModel3DConnected], [], [],\
                           ['colored'], [12], [], [], [], PoseOut)
```

Result

connection_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, select_points_object_model_3d

Possible Successors

project_object_model_3d, object_model_3d_to_xyz, select_object_model_3d

See also

select_object_model_3d, select_points_object_model_3d

Module

3D Metrology

convex_hull_object_model_3d (: : ObjectModel3D : ObjectModel3DConvexHull)

Calculate the convex hull of a 3D object model.

convex_hull_object_model_3d calculates the convex hull of the 3D object model given in ObjectModel3D. The operator returns the convex hull as a 3D object model with the handle ObjectModel3DConvexHull.

If one of the dimensions of the input points has no deviation at all, the result will consist of lines and not triangles.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **ObjectModel3DConvexHull** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model that describes the convex hull.
Number of elements: ObjectModel3DConvexHull == ObjectModel3D

Example

```
gen_object_model_3d_from_points (rand(20)-0.5, rand(20)-0.5,\
                                 rand(20)-0.5, ObjectModel3D)
```

```
convex_hull_object_model_3d (ObjectModel3D, ObjectModel3DConvexHull)
dev_get_window (WindowHandle)
visualize_object_model_3d (WindowHandle, [ObjectModel3DConvexHull], \
                           [], [], [], [], [], [], [], PoseOut)
```

Result

`convex_hull_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_object_model_3d`, `connection_object_model_3d`, `select_points_object_model_3d`

Possible Successors

`project_object_model_3d`

Module

3D Metrology

`edges_object_model_3d` (: : `ObjectModel3D`, `MinAmplitude`, `GenParamName`, `GenParamValue` : `ObjectModel3DEdges`)

Find edges in a 3D object model.

`edges_object_model_3d` finds 3D edges in the 3D object model `ObjectModel3D` and returns them in the 3D object model `ObjectModel3DEdges`.

The operator supports edge extraction only from 3D object models that contain a XYZ mapping, such as models that were created with `xyz_to_object_model_3d` or that were obtained with a sensor that delivers the mapping. `MinAmplitude` defines the minimum amplitude of a discontinuity in order to be classified as an edge. It is given in the same unit as used in `ObjectModel3D`.

The extracted edges are a subset of the points of the input object model. In addition to the coordinates of the edges, the point normal vectors in `ObjectModel3DEdges` contain the viewing direction of each 3D edge point from the viewpoint towards the edge point. Also, the attributes *'edge_dir_x'*, *'edge_dir_y'* and *'edge_dir_z'* contain a vector that is perpendicular to the edge direction and to the viewing direction. The attributes are set such that the 3D object model can be used for edge-supported surface-based matching in `find_surface_model`.

Generic parameters can optionally be used to influence the edge extraction. If desired, these parameters and their corresponding values can be specified with `GenParamName` and `GenParamValue`. The following values for `GenParamName` are possible:

'max_gap': This parameter specifies the maximum gap size in pixels in the XYZ-images that are closed. Gaps larger than this value will contain edges at their boundary, while gaps smaller than this value will not. This suppresses edges around smaller patches that were not reconstructed by the sensor as well as edges at the more distant part of a discontinuity. For sensors with very large resolutions, the value should be increased to avoid spurious edges.

Default: *30*.

'estimate_viewpose': This parameter can be used to turn off the automatic viewpose estimation and set a manual viewpoint.

Default: *'true'*.

'viewpoint': This parameter only has an effect when *'estimate_viewpose'* is set to *'false'*. It specifies the viewpoint from which the 3D data is seen. It is used to determine the viewing directions and edge directions. It defaults to the origin *'0 0 0'* of the 3D data. If the projection center is at a different location, for example, if the 3D

object model was transformed with rigid_trans_object_model_3d or if the 3D sensor performed a similar transformation, the original viewpoint must be set. For this, GenParamValue must contain a string consisting of the three coordinates (x, y and z) of the viewpoint, separated by spaces. The viewpoint is defined in the same coordinate frame as ObjectModel3D. Note that for use of this parameter, the values in the X-, Y-, and Z- images obtained from object_model_3d_to_xyz must have increasing values from left to right, top to bottom, and for object parts further away from the camera, respectively.
Default: *'0 0 0'*.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the 3D object model whose edges should be computed.

▷ **MinAmplitude** (input_control) . . . . . number ⇝ *real* / integer
Edge threshold.

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'max_gap', 'estimate_viewpose', 'viewpoint'}

▷ **GenParamValue** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Values of the generic parameters.
Default: []
Suggested values: GenParamValue ∈ {'0 0 0', 10, 30, 100, 'true', 'false'}

▷ **ObjectModel3DEdges** (output_control) . . . . . object_model_3d ⇝ *handle*
3D object model containing the edges.

Result

edges_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d

Possible Successors

find_surface_model, find_surface_model_image, refine_surface_model_pose, refine_surface_model_pose_image

Module

3D Metrology

fuse_object_model_3d (: : ObjectModel3D, BoundingBox, Resolution, SurfaceTolerance, MinThickness, Smoothing, NormalDirection, GenParamName, GenParamValue : ObjectModel3DFusion)

Fuse 3D object models into a surface.

fuse_object_model_3d fuses multiple point clouds representing an object surface into a watertight surface ObjectModel3DFusion. The operator can be used to simplify the postprocessing step of point clouds that are already registered in the same coordinate system. In particular, unification, suppression of outliers, trade-off between smoothing and preservation of edges, equidistant sub-sampling, hole filling, and meshing of the output surface can often be handled nicely and in high quality. On the other hand, these advantages come at the price of a high runtime.

If you want to fuse 3D point clouds acquired by stereo reconstruction, you should use reconstruct_surface_stereo instead of fuse_object_model_3d.

Workflow

1. Acquire point clouds and transform them into a common coordinate system, for example using register_object_model_3d_pair and register_object_model_3d_global.
2. If not already available, compute triangles or point normals for the point clouds using triangulate_object_model_3d or surface_normals_object_model_3d. A triangulation is more suitable if you have surfaces with many outliers or holes that should be closed. Otherwise, for clean surfaces, you can work with normals.
3. Inspect the normals of the input models using visualize_object_model_3d with GenParamName *'disp_normals'* or dev_inspect_ctrl. The point or triangle normals have to be oriented consistently towards the inside or outside of the object. Set NormalDirection accordingly to *'inwards'* or *'outwards'*.
4. Specify the volume of interest in BoundingBox. To obtain a first guess for BoundingBox, use get_object_model_3d_params with GenParamName set to *'bounding_box1'*.
5. Specify an initial set of parameters: a rough Resolution (e.g., 1/100 of the diameter of the BoundingBox), SurfaceTolerance at least a bit larger (e.g., 5*Resolution), MinThickness as the minimum thickness of the object (if the input point clouds represent the object only from one side, set it very high, so that the object is cut off at the BoundingBox), Smoothing set to *1.0*.
6. Apply fuse_object_model_3d and readjust the parameters to improve the results with respect to quality and runtime, see below. Use a Resolution just fine enough to make out the details of your object while tuning the other parameters, in order to avoid long runtimes. Also consider using the additional parameters in GenParamName.

Parameter Description

See the HDevelop example fuse_object_model_3d_workflow for an explanation how to fine-tune the parameters for your application.

The input point clouds ObjectModel3D have to lie in a common coordinate system and add up to the initial surface. Furthermore, they must contain triangles or point normals. If both attributes are present, normals are used as a default due to speed advantages. If triangles should be used, use copy_object_model_3d to obtain only point and triangle information. Surfaces with many outliers or holes to be closed should be used with a triangulation, clean surfaces with normals. The point or triangle normals have to be oriented consistently towards the inside or outside of the object.

NormalDirection is used to specify whether the point or triangle normals point *'inwards'* or *'outwards'*. If only one value is specified, it is applied to all input models. Otherwise, the number of values has to equal the number of input models.

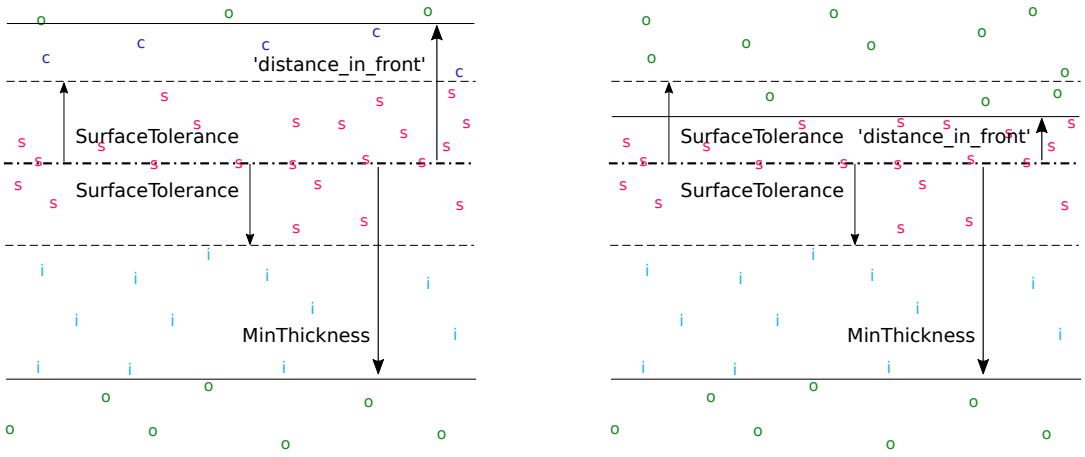
BoundingBox specifies the volume of interest to be taken into account for input and output. Note that points outside the bounding box are discarded. Triangles of the input point cloud with a point outside the BoundingBox are discarded, not clipped. The BoundingBox is specified as a tuple [x1,y1,z1,x2,y2,z2] assigning two opposite corner points P1=[x1,y1,z1] and P2=[x2,y2,z2] of the rectangular cuboid (with edges parallel to the coordinate axes). For a valid bounding box, P1 must be the point on the front lower left corner and P2 on the back upper right corner of the bounding box, i.e., x1<x2, y1<y2 and z1<z2. Note that the operator will try to produce a closed surface. If the input point clouds represent the object from only one point of view, one wants the bounding box usually to cut off the unknown part, wherefore MinThickness should be set e.g., to a value larger than or equal to the length of the diagonal of the bounding box (which can be obtained by using get_object_model_3d_params with the parameter *'diameter_axis_aligned_bounding_box'*). An object cut off by a surface of the bounding box has no points at this specific surface, thus has a hole. Note also that you may have to rotate the input point clouds in order make the bounding box cut off the unknown part in the right place, since the edges of the bounding box are always parallel to the coordinate axes. This can be achieved e.g., using affine_trans_object_model_3d or rigid_trans_object_model_3d.

Resolution specifies the distance of neighboring grid points in each coordinate direction in the discretization of the BoundingBox. Resolution is set in the same unit as used in ObjectModel3D. Too small values will unnecessarily increase the runtime, so it is recommended to begin with a coarse resolution. Too large values will lead to a reconstruction with high loss of details. Smoothing may need to be adapted when Resolution is changed. Resolution should always be a bit smaller than SurfaceTolerance in order to avoid discretization artifacts.

SurfaceTolerance specifies how much noise in the input point cloud should be combined to the surface from its inside and outside. Sole exemption when SurfaceTolerance is larger than *'distance_in_front'*, in that case *'distance_in_front'* determines the surface thickness to the front of the object. SurfaceTolerance is set in the same unit as used in ObjectModel3D. Points in the interior of the object as specified by NormalDirection

(and also GenParamName='angle_threshold') are considered surely inside the object if their distance to the initial surface exceeds SurfaceTolerance but is smaller than MinThickness. SurfaceTolerance always has to be smaller than MinThickness. SurfaceTolerance should always be a bit larger than Resolution in order to avoid discretization artifacts.

MinThickness specifies the thickness of the object in normal direction of the initial surfaces. MinThickness is set in the same unit as used in ObjectModel3D. Points which are specified by NormalDirection (and also GenParamName='angle_threshold') to be in the interior of the object are only considered as being inside if their distance to the initial surface does not exceed MinThickness. Note that this can lead to a hollow part of the object. MinThickness always has to be larger than SurfaceTolerance. For point clouds representing the object from different sides, MinThickness is best set as the thickness of the objects narrowest part. Note that the operator will try to produce a closed surface. If the input point clouds represent the object only from one side, this parameter should be set very large, so that the object is cut off at the bounding box. The backside of the objects is not observed and thus its reconstruction will probably be incorrect. If you observe several distinct objects from only one side, you may want to reduce the parameter MinThickness to restrict the depth of reconstructed objects and thus keep them from being smudged into one surface. Too small values can result in holes or double walls in the fused point cloud. Too large values can result in a distorted point cloud or blow up the surface towards the outside of the object (if the surface is blown up beyond the bounding box, no points will be returned).



(1) (2)

Schematic view of the parameters SurfaceTolerance, MinThickness and the value *'distance_in_front'* with the aid of an example surface (_.): o are points taken as outside, s are points of the surface, i are points surely inside the object, c are points also considered for the evaluation of the surface. (1): *'distance_in_front'* smaller than SurfaceTolerance (2): *'distance_in_front'* larger than SurfaceTolerance.

Smoothing determines how important a small total variation of the distance function is compared to data fidelity. Thus, Smoothing regulates the 'jumpiness' of the resulting surface. Note that the actual value of Smoothing for a given data to result in an appropriate and visually pleasing surface has to be found by trial and error. Too small values lead to integrating many outliers into the surface even if the surface then exhibits many jumps. Too large values lead to lost fidelity towards the input point clouds (how the algorithm views distances to the input point clouds depends heavily on SurfaceTolerance and MinThickness). Smoothing may need to be adapted when Resolution is changed.

By setting GenParamName to the following values, the additional parameters can be set with GenParamValue:

'distance_in_front' Points in the exterior of the object as specified by NormalDirection (and also GenParamName='angle_threshold') are only considered as part of the object if their distance to the initial surface does not exceed *'distance_in_front'*. This is the outside analogous to MinThickness of the interior, except that *'distance_in_front'* does not have to be larger than SurfaceTolerance. In case *'distance_in_front'* is smaller than SurfaceTolerance it determines the surface thickness to the front. This parameter is useful if holes in the surface should be closed along a jump in the surface (for example along the viewing direction of the sensor). In this case, *'distance_in_front'* can be set to a small value in order to avoid a wrong initialization of the distance field. *'distance_in_front'* is set in the same unit as used in ObjectModel3D. *'distance_in_front'* should always be a bit larger than Resolution in order to avoid discretization artifacts. Per default, *'distance_in_front'* is set to a value larger than the bounding box diameter, therewith all points outside of the object in the bounding box are considered.

Suggested values: *0.001*, *0.1*, *1*, *10*.

Default: Larger than the bounding box diameter.

Restriction: *'distance_in_front'* > 0

'angle_threshold' specifies the angle of a cone around a surface normal. *'angle_threshold'* is set in [rad]. When determining the distance information for data fidelity, only points are considered lying in such a cone starting at their closest surface point. For example, if distances to triangles are considered, *'angle_threshold'* can be set to *0.0*, so that only the volume directly above the triangle is considered (thus a right prism). If point normals are used and thus distances to normals are considered, *'angle_threshold'* has to be set to a higher value. When outliers disrupt the result, decreasing *'angle_threshold'* may help. If holes in the surface should be closed along a jump in the surface (for example along the viewing direction of the sensor), enlarging *'angle_threshold'* may help.

Suggested values: *'rad(0.0)'*, *'rad(10.0)'*, *'rad(30.0)'*.

Default: *'rad(10.0)'*.

Restriction: *'angle_threshold'* >= 0

'point_meshing' determines whether the output points should be triangulated with the algorithm 'marching tetrahedra', which can be activated by setting *'point_meshing'* to *'isosurface'*. Note that there are more points in `ObjectModel3DFusion` if meshing of the isosurface is enabled even if the used `Resolution` is the same.

List of values: *'none'*, *'isosurface'*.

Default: *'isosurface'*.

Fusion algorithm

The algorithm will produce a watertight, closed surface (which is maybe cut off at the `BoundingBox`). The goal is to obtain a preferably smooth surface while keeping form fidelity. To this end, the bounding box is sampled and each sample point is assigned an initial distance to a so-called isosurface (consisting of points with distance 0). The final distance values (and thus the isosurface) are obtained by minimizing an error function based on fidelity to the initial point clouds on the one hand and total variation ('jumpiness') of the distance function on the other hand. This leads to a fusion of the input point clouds (see paper in References below).

The calculation of the isosurface can be influenced with the parameters of the operator. The distance between sample points in the bounding box (in each coordinate direction) can be set with the parameter `Resolution`.

Fidelity to the initial point clouds is grasped as the signed distances of sample points, lying on the grid, in the bounding box to their nearest neighbors (points or triangles) on the input point clouds. Whether a sample point in the bounding box is considered to lie outside or inside the object (the sign of the distance) is determined by the normal of its nearest neighbor on the initial surface and the set `NormalDirection`. To determine if a sample point is surely inside or outside the object with respect to an input point cloud, the distance to its nearest neighbor on the initial surface is determined. A point on the inside is considered surely inside if the distance exceeds `SurfaceTolerance` but not `MinThickness`, while a point on the outside counts as exteriorly if the distance exceeds *'distance_in_front'*.

Fidelity to the initial point clouds is only considered for those sample points lying within `MinThickness` inside or within `GenParamName` *'distance_in_front'* outside the initial surface.

Furthermore, fidelity is not maintained for a given sample point lying outside a cone around `GenParamName` *'angle_threshold'*. Thus it is not maintained if the line from the sample point to its nearest neighbor on the initial surface differs from the surface normal of the nearest neighbor by an angle more than `GenParamName` *'angle_threshold'*. Note that the distances to nearest neighboring triangles will often yield more satisfying results while distances to nearest points can be calculated much faster.

The subsequent optimization of the distance values is the same as the one used in `reconstruct_surface_stereo` with `Method`=*'surface_fusion'*.

The parameter `Smoothing` regulates the 'jumpiness' of the distance function by weighing the two terms in the error function: Fidelity to the initial point clouds on the one hand, total variation of the distance function on the other hand. Note that the actual value of `Smoothing` for a given data set to be visually pleasing has to be found by trial and error.

Each 3D point of the object model returned in `ObjectModel3DFusion` is extracted from the isosurface where the distance function equals zero. Its normal vector is calculated from the gradient of the distance function. The so-obtained point cloud can also be meshed using the algorithm 'marching tetrahedra' by setting the `GenParamName` *'point_meshing'* to the `GenParamValue` *'isosurface'*.

Troubleshooting

Please follow the workflow above. If the results are not satisfactory, please consult the following hints and ideas:

Quality of the input point clouds The input point clouds should represent the entire object surface. If point normals are used, the points should be dense on the entire surface, not only along edges of the object. In particular, for CAD-data typically triangulation has to be used.

Used attribute Using triangles instead of point normals will typically yield results of higher quality. If both attributes are present, point normals are used per default. If triangles should be used, use `copy_object_model_3d` to obtain only point and triangle information.

Outliers If outliers of the input models disturb the output surface even for high values of `Smoothing`, try to decrease `GenParamName` *'angle_threshold'*. If wanted, outliers of the input models can also be removed, for example using `connection_object_model_3d`. With reduced influence also modifying `GenParamName` *'distance_in_front'* may help to reduce certain outliers.

Closing of holes If holes in the surface are not closed even for high values of `Smoothing` (for example a jump in the surface along the viewing direction of the sensor), try to decrease `GenParamName` *'distance_in_front'*. Enlarging `GenParamName` *'angle_threshold'* may help the algorithm to close the gap. Note that `triangulate_object_model_3d` can close gaps when triangulating sensor data which contains a 2D mapping.

Empty output If the output contains no point, try to decrease `Smoothing`. If there is no output even for very low values of `Smoothing`, you may want to check if `MinThickness` is set too large and if the set `NormalDirection` is correct.

Runtime

In order to improve the runtime, consider the following hints:

Extent of the bounding box The bounding box should be tight around the volume of interest. Else, the runtime will increase drastically but without any benefit.

Resolution Enlarging the parameter `Resolution` will speed up the execution considerably.

Used attribute Using point normals instead of triangles will speed up the execution. If both, normals and triangles, are present in the input models, normals are used per default.

Density of input point clouds The input point clouds can be thinned out using `sample_object_model_3d` (if normals are used) or `simplify_object_model_3d` with `GenParamName` *'avoid_triangle_flips'* set to *'true'* (if triangles are used).

Distances to surface Make sure that `MinThickness` and `GenParamName` *'distance_in_front'* are not set unnecessarily large, since this can slow down the preparation and distance computation.

Parameters

▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handles of the 3D object models.

▷ **`BoundingBox`** (input_control) . . . . . number-array ⇝ *real* / integer
The two opposite bound box corners.

▷ **`Resolution`** (input_control) . . . . . number ⇝ *real* / integer
Used resolution within the bounding box.
Default: 1.0
Suggested values: `Resolution` ∈ {1.0, 1.1, 1.5, 10.0, 100.0}

▷ **`SurfaceTolerance`** (input_control) . . . . . number ⇝ *real* / integer
Distance of expected noise to surface.
Default: 1.0
Suggested values: `SurfaceTolerance` ∈ {1.0, 1.1, 1.5, 10.0, 100.0}

▷ **`MinThickness`** (input_control) . . . . . number ⇝ *real* / integer
Minimum thickness of the object in direction of the surface normal.
Default: 1.0
Suggested values: `MinThickness` ∈ {1.0, 1.1, 1.5, 10.0, 100.0}

▷ **`Smoothing`** (input_control) . . . . . number ⇝ *real* / integer
Weight factor for data fidelity.
Default: 1.0
Suggested values: `Smoothing` ∈ {1.0, 1.1, 1.5, 10.0, 100.0}

▷ **NormalDirection** (input_control) ..... string(-array) ⇝ *string*
Direction of normals of the input models.
Default: 'inwards'
List of values: NormalDirection ∈ {'inwards', 'outwards'}

▷ **GenParamName** (input_control) ..... attribute.name-array ⇝ *string*
Name of the generic parameter.
Default: []
List of values: GenParamName ∈ {'point_meshing', 'angle_threshold', 'distance_in_front'}

▷ **GenParamValue** (input_control) ..... attribute.value-array ⇝ *string* / real / integer
Value of the generic parameter.
Default: []
Suggested values: GenParamValue ∈ {'isosurface', 'none', 0.0, 0.1, 0.175, 0.524}

▷ **ObjectModel3DFusion** (output_control) ..... object_model_3d ⇝ *handle*
Handle of the fused 3D object model.

Result

fuse_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_object_model_3d, register_object_model_3d_pair, register_object_model_3d_global, surface_normals_object_model_3d, triangulate_object_model_3d, simplify_object_model_3d, get_object_model_3d_params

Possible Successors

write_object_model_3d, create_surface_model

See also

reconstruct_surface_stereo

References

C. Zach, T. Pock, and H. Bischof: "A globally optimal algorithm for robust TV-L1 range image integration." Proceedings of IEEE International Conference on Computer Vision (ICCV 2007).

Module

3D Metrology

intersect_plane_object_model_3d (: : ObjectModel3D, Plane : ObjectModel3DIntersection)

Intersect a 3D object model with a plane.

intersect_plane_object_model_3d intersects a 3D object model with a plane that is defined by the x-y plane of the pose that is specified with the parameter Plane. The z-axis of the pose corresponds to the normal of the plane.

The result is a set of 3D points connected by lines that is returned as 3D object model in ObjectModel3DIntersection. Every triangle that intersects with the plane creates two intersection points and a line between the two points. The resulting set of lines is coplanar.

The lines can be displayed with disp_object_model_3d and queried with get_object_model_3d_params using the parameter *'lines'*.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* (*7* for `Plane`) or *N* (*N*7* for `Plane`). Parameters with tuple length *1* (*7* for `Plane`) will be repeated internally such that the number of computed output models is always *N*.

Parameters

▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **`Plane`** (input_control) . . . . . pose(-array) ⇝ *real* / integer
Pose of the plane.
Default: [0,0,0,0,0,0,0]

▷ **`ObjectModel3DIntersection`** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model that describes the intersection as a set of lines.

Example

```
gen_object_model_3d_from_points (rand(20)-0.5, rand(20)-0.5,\
                                 rand(20)-0.5, ObjectModel3D)
convex_hull_object_model_3d (ObjectModel3D, ObjectModel3DConvexHull)
intersect_plane_object_model_3d (ObjectModel3DConvexHull, [0,0,0,0,0,0,0], \
                                 ObjectModel3DIntersection)
dev_get_window (WindowHandle)
visualize_object_model_3d (WindowHandle, [ObjectModel3DIntersection, \
                           ObjectModel3DConvexHull], [], [], \
                           ['alpha_1'], [0.5], [], [], [], PoseOut)
```

Result

`intersect_plane_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`read_object_model_3d`, `select_points_object_model_3d`

Possible Successors

`connection_object_model_3d`

See also

`reduce_object_model_3d_by_view`

Module

3D Metrology

`object_model_3d_to_xyz` (: X, Y, Z : ObjectModel3D, Type, CamParam, Pose :)

Transform 3D points from a 3D object model to images.

The operator `object_model_3d_to_xyz` transforms the 3D points of the 3D object model `ObjectModel3D` into the three images `X`, `Y`, and `Z`.

Three transformation modes are possible. The parameter `Type` is used to select one of them. Note that multiple 3D object models can be passed in `ObjectModel3D` only for the mode *'cartesian_faces'*. All other modes expect a single 3D object model.

'cartesian': First, each point is transformed into the camera coordinate system using the given `Pose`. Then, these coordinates are projected into the image coordinate system based on the internal camera parameters `CamParam`.

The internal camera parameters `CamParam` describe the projection characteristics of the camera (see Calibration). The `Pose` is in the form ${}^{ccs}\mathbf{P}_{mcs}$, where *ccs* denotes camera coordinate system and *mcs* the model coordinate system (which is a 3D world coordinate system), see Transformations / Poses and "`Solution Guide III-C - 3D Vision`". Hence, it describes the position and orientation of the model coordinate system relative to the camera coordinate system.

The X-, Y-, and Z-coordinates of the transformed point are written into the corresponding image at the position of the projection. If multiple points are projected to the same image coordinates, the point with the smallest Z-value is written (hidden-point removal). The dimensions of the returned images are defined by the camera parameters.

The returned images show the object as it would look like when seeing it with the specified camera under the specified pose.

'cartesian_faces': In order to use this transformation, the input 3D object models need to contain faces (triangles or polygons), otherwise, the 3D object model without faces is disregarded. Note that if the 3D object models have polygon faces, those are converted internally to triangles. This conversion can be done beforehand to speed up this operator. For this, `read_object_model_3d` can be called with the `GenParamName` *'convert_to_triangles'* set to *'true'*, to convert all faces to triangles. Alternatively, `triangulate_object_model_3d` can be called prior to this operator.

First, each face of the 3D object models `ObjectModel3D` is transformed into the camera coordinate system using the given `Pose`. Then, these coordinates are projected into the image coordinate system based on the internal camera parameters `CamParam`, while keeping the 3D information (X-, Y-, and Z-coordinates) for each of those pixels. For a more detailed explanation of `CamParam` and `Pose` please refer to the section *'cartesian'*. If multiple faces are projected to the same image coordinates, the value with the smallest Z-value is written (hidden-point removal). The dimensions of the returned images are defined by the camera parameters.

The returned images show the objects as they would look like when seeing them with the specified camera under the specified pose.

In case that OpenGL 2.1, GLSL 1.2, and the OpenGL extensions GL_EXT_framebuffer_object and GL_EXT_framebuffer_blit are available, speed increases.

This `Type` can be used to create 3D object models containing 2D mapping data, by creating a 3D object model from the returned images using `xyz_to_object_model_3d`. Note that in many cases, it is recommended to use the 2D mapping data, if available, for speed and robustness reasons. This is beneficial for example when using `sample_object_model_3d`, `surface_normals_object_model_3d`, or when preparing a 3D object model for surface-based matching, e.g., smoothing, removing outliers, and reducing the domain.

'cartesian_faces_no_opengl': This transformation mode works in the same way as the method *'cartesian_faces'* but does not use OpenGL. In general, *'cartesian_faces'* automatically determines if OpenGL is available. Thus, it is usually not required to use *'cartesian_faces_no_opengl'* explicitly. It can make sense, however, to use it in cases where the automatic mode selection does not work due to, for example, driver issues with OpenGL.

'from_xyz_map': This transformation mode works only if the 3D object model was created with the operator `xyz_to_object_model_3d`. It writes each 3D point to the image coordinate where it originally came from, using the mapping attribute that is stored within the 3D object model.

The parameters `CamParam` and `Pose` are ignored. The dimensions of the returned images are equal to the dimensions of the original images that were used with `xyz_to_object_model_3d` to create the 3D object model and can be queried from `get_object_model_3d_params` with *'mapping_size'*.

This transformation mode is faster than *'cartesian'*. It is suitable, e.g., to visualize the results of a segmentation done with `segment_object_model_3d`.

Attention

Cameras with hypercentric lenses are not supported. For displaying large faces with a non-zero distortion in `CamParam`, note that the distortion is only applied to the points of the model. In the projection, these points are subsequently connected by straight lines. For a good approximation of the distorted lines, please use a triangulation with sufficiently small triangles.

Parameters

▷ **X** (output_object) . . . . . singlechannelimage ⇝ *object :* real
Image with the X-Coordinates of the 3D points.

▷ **Y** (output_object) . . . . . singlechannelimage ⇝ *object :* real
Image with the Y-Coordinates of the 3D points.

▷ **Z** (output_object) . . . . . singlechannelimage ⇝ *object :* real
Image with the Z-Coordinates of the 3D points.

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **Type** (input_control) . . . . . string ⇝ *string*
Type of the conversion.
Default: 'cartesian'
List of values: `Type` ∈ {'cartesian', 'cartesian_faces', 'from_xyz_map', 'cartesian_faces_no_opengl'}

▷ **CamParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Camera parameters.

▷ **Pose** (input_control) . . . . . pose ⇝ *real* / integer
Pose of the 3D object model.
Number of elements: Pose == 0 || Pose == 7 || Pose == 12

Result

The operator `object_model_3d_to_xyz` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_object_model_3d`, `xyz_to_object_model_3d`, `triangulate_object_model_3d`

Alternatives

`project_object_model_3d`

See also

`xyz_to_object_model_3d`, `get_object_model_3d_params`

Module

3D Metrology

prepare_object_model_3d (: : ObjectModel3D, Purpose, OverwriteData, GenParamName, GenParamValue :)

Prepare a 3D object model for a certain operation.

The operator `prepare_object_model_3d` prepares the 3D object model `ObjectModel3D` for a following operation given in `Purpose`. It computes values required for the operation and stores them in `ObjectModel3D`, thus speeding up the following operation. It is not necessary to call `prepare_object_model_3d`. However, if the 3D object model is to be used multiple times for the same operation, it can be faster to do so.

The following values are possible for `Purpose`:

'shape_based_matching_3d': The 3D object model is prepared to be used in `create_shape_model_3d`. For this, there are no generic parameters to set.

'segmentation': The 3D object model is prepared to be used in `segment_object_model_3d`. For the preparation the 3D object model must have an attribute with the face triangles and an attribute with the 3D point coordinates.

If the 3D object model has no attribute with the face triangles, a simple triangulation is performed (even if `OverwriteData` is set to *'false'*). For this, the 3D object model must have an attribute with the 3D point

coordinates and an attribute with the mapping from the point coordinates to image coordinates. Only points originating from neighboring pixels are triangulated. Additionally, holes in the image region can be filled with a Delaunay triangulation (see *'max_area_holes'* below). Only holes which are completely surrounded by the image region are closed.

'distance_computation': The 3D object model is prepared to be used in `distance_object_model_3d`.

'gen_xyz_mapping': The XYZ-mapping information of a 3D object model containing an ordered point cloud is computed, i.e. image coordinates are assigned for each 3D point. For this, either the generic parameter *'xyz_map_width'* or *'xyz_map_height'* must be set, to indicate whether the point cloud is ordered row-wise or column-wise and define the image dimensions (see *'xyz_map_width'* and *'xyz_map_height'* below).

Note that in many cases, it is recommended to use the 2D mapping data, if available, for speed and robustness reasons. This is beneficial especially when using `sample_object_model_3d`, `surface_normals_object_model_3d`, or when preparing a 3D object model for surface-based matching, e.g., smoothing, removing outliers, and reducing the domain.

The parameter `OverwriteData` defines if the existing data of an already prepared 3D object model shall be removed. If `OverwriteData` is set to *'true'*, the prepared data, defined with the parameter `Purpose`, is overwritten. If `OverwriteData` is set to *'false'*, the prepared data is not overwritten. If there is no prepared data `OverwriteData` is ignored and data is saved in a 3D object model. The parameter `OverwriteData` can be used for choosing another set of generic parameters `GenParamName` and `GenParamValue`. The parameter `OverwriteData` has no influence if the parameter `Purpose` is set to *'shape_based_matching_3d'*, because for that, there are no generic parameters to set.

The generic parameters can optionally be used to influence the preparation. If desired, these parameters and their corresponding values can be specified by using `GenParamName` and `GenParamValue`, respectively. The following values for `GenParamName` are possible:

'max_area_holes': This parameter is only valid if `Purpose` is set to *'segmentation'*. The parameter specifies which area holes of the point coordinates are closed during a simple Delaunay triangulation. Only holes which are completely surrounded by the image region are closed. If *'max_area_holes'* is set to *0*, no holes are triangulated. If the parameter *'max_area_holes'* is set greater or equal than *1* pixel, the holes with an area less or equal than *'max_area_holes'* are closed by a meshing.

Suggested values: *1*, *10*, *100*.

Default: *10*.

'distance_to': This parameter is only valid if `Purpose` is set to *'distance_computation'*. The parameter specifies the type of data to which the distance shall be computed to. It is described in more detail in the documentation of `distance_object_model_3d`.

List of values: *'auto'*, *'triangles'*, *'points'*, *'primitive'*.

Default: *'auto'*.

'method': This parameter is only valid if `Purpose` is set to *'distance_computation'*. The parameter specifies the method to be used for the distance computation. It is described in more detail in the documentation of `distance_object_model_3d`.

List of values: *'auto'*, *'kd-tree'*, *'voxel'*, *'linear'*.

Default: *'auto'*.

'max_distance': This parameter is only valid if `Purpose` is set to *'distance_computation'*. The parameter specifies the maximum distance of interest for the distance computation. If it is set to *0*, no maximum distance is used. It is described in more detail in the documentation of `distance_object_model_3d`.

Suggested values: *0*, *0.1*, *1*, *10*.

Default: *0*.

'sampling_dist_rel': This parameter is only valid if `Purpose` is set to *'distance_computation'*. The parameter specifies the relative sampling distance when computing the distance to triangles with the method *'voxel'*. It is described in more detail in the documentation of `distance_object_model_3d`.

Suggested values: *0.03*, *0.01*.

Default: *0.03*.

'sampling_dist_abs': This parameter is only valid if `Purpose` is set to *'distance_computation'*. The parameter specifies the absolute sampling distance when computing the distance to triangles with the method *'voxel'*. It is described in more detail in the documentation of `distance_object_model_3d`.

Suggested values: *1*, *5*, *10*.

Default: None.

'xyz_map_width': This parameter is only valid if `Purpose` is set to *'gen_xyz_mapping'*. The parameter indicates that the point cloud is ordered row-wise and the passed value is used as the width of the image. The height of the image is calculated automatically. Only one of the two parameters *'xyz_map_width'* and *'xyz_map_height'* can be set.

Default: None.

'xyz_map_height': This parameter is only valid if `Purpose` is set to *'gen_xyz_mapping'*. The parameter indicates that the point cloud is ordered column-wise and the passed value is used as the height of the image. The width of the image is calculated automatically. Only one of the two parameters *'xyz_map_width'* and *'xyz_map_height'* can be set.

Default: None.

Parameters

▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **`Purpose`** (input_control) . . . . . string ⇝ *string*
Purpose of the 3D object model.
Default: 'shape_based_matching_3d'
Suggested values: `Purpose` ∈ {'shape_based_matching_3d', 'segmentation', 'distance_computation', 'gen_xyz_mapping'}

▷ **`OverwriteData`** (input_control) . . . . . string ⇝ *string*
Specify if already existing data should be overwritten.
Default: 'true'
List of values: `OverwriteData` ∈ {'true', 'false'}

▷ **`GenParamName`** (input_control) . . . . . attribute.name-array ⇝ *string* / real / integer
Names of the generic parameters.
Default: []
List of values: `GenParamName` ∈ {'max_area_holes', 'distance_to', 'method', 'max_distance', 'sampling_dist_rel', 'sampling_dist_abs', 'xyz_map_width', 'xyz_map_height'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value-array ⇝ *string* / real / integer
Values of the generic parameters.
Default: []
Suggested values: `GenParamValue` ∈ {0, 1, 100, 'auto', 'triangles', 'points', 'primitive', 'kd-tree', 'voxel', 'linear', 0.01, 0.03}

Example

```
read_object_model_3d ('object_model_3d', 'm', [], [], ObjectModel3D, Status)
prepare_object_model_3d (ObjectModel3D, 'gen_xyz_mapping', 'true',\
                         'xyz_map_width', Width)
object_model_3d_to_xyz (X, Y, Z, ObjectModel3D, 'from_xyz_map', [], [])
```

Result

The operator `prepare_object_model_3d` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_object_model_3d`, `xyz_to_object_model_3d`

Possible Successors

`create_shape_model_3d`, `create_surface_model`, `distance_object_model_3d`, `find_surface_model`, `fit_primitives_object_model_3d`, `refine_surface_model_pose`,

segment_object_model_3d, simplify_object_model_3d, sample_object_model_3d, surface_normals_object_model_3d

Module

3D Metrology

project_object_model_3d (: ModelContours : ObjectModel3D, CamParam, Pose, GenParamName, GenParamValue :)

Project a 3D object model into image coordinates.

The operator project_object_model_3d projects a 3D object model into the image coordinate system and returns the projected contours in ModelContours. This operator is particularly useful for the visualization of 3D object models. Note that primitives are not projected but silently ignored. The coordinates of the 3D object model are given in the model coordinate system (*mcs*), a 3D world coordinate system. First, they are transformed into the camera coordinate system (*ccs*) using the given Pose. Then, these coordinates are projected into the image coordinate system based on the internal camera parameters CamParam. Thereby the pose is needed in the form ${}^{ccs}\mathbf{P}_{mcs}$, see Transformations / Poses and "Solution Guide III-C - 3D Vision". Thus, the Pose describes the position and orientation of the world coordinate system with respect to the camera coordinate system. The internal camera parameters CamParam describe the projection characteristics of the camera (see Calibration).

There are some generic parameters that can optionally be used to influence the projection. If desired, these parameters and their corresponding values can be specified by using GenParamName and GenParamValue, respectively. The following values for GenParamName are possible:

'data': This parameter specifies which geometric data of the 3D object model should be projected. If *'data'* is set to *'faces'*, the faces of the 3D object model are projected. The faces are represented by their border lines in ModelContours. If *'data'* is set to *'lines'*, the 3D lines of the 3D object model are projected. If *'data'* is set to *'points'*, the points of the 3D object model are projected. The projected points can be represented in ModelContours in different ways. The point representation can be selected by using the generic parameter *'point_shape'* (see below). Finally, if *'data'* is set to *'auto'*, HALCON automatically chooses the most descriptive geometry data that is available in the 3D object model for visualization.

List of values: *'auto'*, *'faces'*, *'lines'*, *'points'*.

Default: *'auto'*.

'point_shape': This parameter specifies how points are represented in the output contour ModelContours. Consequently, this parameter only has an effect if the points of the 3D object model are selected for projection (see above). If *'point_shape'* is set to *'circle'*, points are represented by circles, whereas if *'point_shape'* is set to *'cross'*, points are represented by crosses. In both cases the size of the points (i.e., the size of the circles or the size of the crosses) can be specified by the generic parameter *'point_size'* (see below). The orientation of the crosses can be specified by the generic parameter *'point_orientation'* (see below).

List of values: *'circle'*, *'cross'*.

Default: *'circle'*.

'point_size': This parameter specifies the size of the point representation in the output contour ModelContours, i.e., the size of the circles or the size of the crosses depending on the selected *'point_shape'*. Consequently, this parameter only has an effect if the points of the 3D object model are selected for projection (see above). The size must be given in pixel units. If *'point_size'* is set to *0*, each point is represented by a contour that contains a single contour point.

Suggested values: *0*, *2*, *4*.

Default: *4*.

'point_orientation': This parameter specifies the orientation of the crosses in radians. Consequently, this parameter only has an effect if the points of the 3D object model are selected for projection and *'point_shape'* is set to *'cross'* (see above).

Suggested values: *0*, *0.39*, *0.79*.

Default: *0.79*.

'union_adjacent_contours': This parameter specifies if adjacent projected contours should be joined or not. Activating this option is equivalent to calling union_adjacent_contours_xld after this operator, but significantly faster.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'hidden_surface_removal': This parameter can be used to switch on or off the removal of hidden surfaces. If *'hidden_surface_removal'* is set to *'true'*, only those projected edges are returned that are not hidden by faces of the 3D object model. If *'hidden_surface_removal'* is set to *'false'*, all projected edges are returned. This is faster than a projection with *'hidden_surface_removal'* set to *'true'*.

If the system variable (see `set_system`) *'opengl_hidden_surface_removal_enable'* is set to *'true'* (which is the default if it is available) and *'hidden_surface_removal'* is set to *'true'*, the projection of the model is accelerated using the graphics card. Depending on the graphics card this is significantly faster than the non accelerated algorithm. Be aware that the results of the OpenGL projection are slightly different compared to the analytic projection. Notable, only the contours visible through `CamParam` are projected in this mode.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'min_face_angle': 3D edges are only projected if the angle between the two 3D faces that are incident with the 3D edge is at least *'min_face_angle'*. If *'min_face_angle'* is set to *0.0*, all edges are projected. If *'min_face_angle'* is set to π (equivalent to 180 degrees), only the silhouette of the 3D object model is returned. This parameter can be used to suppress edges within curved surfaces, e.g., the surface of a cylinder or cone.

Suggested values: *0.17*, *0.26*, *0.35*, *0.52*.

Default: *0.52*.

Attention

Cameras with hypercentric lenses are not supported. For displaying large faces with a non-zero distortion in `CamParam`, note that the distortion is only applied to the points of the model. In the projection, these points are subsequently connected by straight lines. For a good approximation of the distorted lines, please use a triangulation with sufficiently small triangles.

Parameters

▷ **`ModelContours`** (output_object) ..... xld_cont(-array) ⇝ *object*
Projected model contours.

▷ **`ObjectModel3D`** (input_control) ..... object_model_3d ⇝ *handle*
Handle of the 3D object model.

▷ **`CamParam`** (input_control) ..... campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **`Pose`** (input_control) ..... pose ⇝ *real* / integer
3D pose of the world coordinate system in camera coordinates.
Number of elements: Pose == 7

▷ **`GenParamName`** (input_control) ..... string(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: `GenParamName` ∈ {'true', 'false', 'hidden_surface_removal', 'min_face_angle', 'data', 'point_shape', 'point_size', 'point_orientation', 'union_adjacent_contours'}

▷ **`GenParamValue`** (input_control) ..... string(-array) ⇝ *string* / integer / real
Value of the generic parameter.
Default: []
Suggested values: `GenParamValue` ∈ {0.17, 0.26, 0.35, 0.52, 'true', 'false', 'auto', 'points', 'faces', 'lines', 'circle', 'cross', 1, 2, 3, 4, 0.785398}

Result

`project_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised. If the geometric data that was selected for the projection is not available in the 3D object model, the error 9514 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

read_object_model_3d, affine_trans_object_model_3d, prepare_object_model_3d

Possible Successors

clear_object_model_3d

See also

project_shape_model_3d, object_model_3d_to_xyz

Module

3D Metrology

projective_trans_object_model_3d (: : ObjectModel3D, HomMat3D : ObjectModel3DProjectiveTrans)

Apply an arbitrary projective 3D transformation to 3D object models.

projective_trans_object_model_3d applies an arbitrary projective 3D transformation to the points of 3D object models and returns the handles of the transformed 3D object models. The projective transformation is described by the homogeneous transformation matrix given in HomMat3D (see projective_trans_point_3d).

The transformation matrix can be created, e.g., using the operator vector_to_hom_mat3d.

Attention

projective_trans_object_model_3d transforms the attributes of type 3D points. Attributes of type shape model for shape-based 3D matching, of type 3D primitive, and of type normals are not transformed. Therefore, these attributes do not exist in the transformed 3D object model. All other attributes are copied without modification. To transform 3D primitives, the operator rigid_trans_object_model_3d must be used.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handles of the 3D object models.

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Homogeneous projective transformation matrix.

▷ **ObjectModel3DProjectiveTrans** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handles of the transformed 3D object models.

Result

If the parameters are valid, the operator projective_trans_object_model_3d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d

Possible Successors

project_object_model_3d, object_model_3d_to_xyz

See also

affine_trans_point_3d, rigid_trans_object_model_3d, affine_trans_object_model_3d

Module

3D Metrology

register_object_model_3d_global (: : ObjectModels3D, HomMats3D, From, To, GenParamName, GenParamValue : HomMats3DOut, Scores)

Improve the relative transformations between 3D object models based on their overlaps.

register_object_model_3d_global improves the relative transformations between 3D object models, which is called global registration. In particular, under the assumption that all input 3D objects models in ObjectModels3D have a known approximated spatial relation, all possible pairwise overlapping areas are calculated and optimized for a better alignment. The resulting offset is then synchronously minimized for all pairs. The entire process is then repeated iteratively from the newly resulting starting poses. The result in HomMats3DOut describes a transformation that can be applied with affine_trans_object_model_3d to the input 3D object models to transform all in a common reference frame. Scores contains for every 3D object model the number of found neighbors with a sufficient overlap. If no overlap is found for at least one object, an exception is raised.

Three types for the interpretation of the starting poses in HomMats3D are available, which is controlled by the parameters From and To:

First, if From is set to *'global'*, the parameter HomMats3D must contain a rigid transformation with 12 entries for each 3D object model in ObjectModels3D that describes its position in relation to a common global reference frame. In this case, To must be empty. This case is suitable, e.g., if transformations are applied by a turning table or a robot to either the camera or the object. In this case, all neighborhoods that are possible are considered for the global optimization.

Second, if From is set to *'previous'*, the parameter HomMats3D must contain a rigid transformation for each subsequent pair of 3D object models in ObjectModels3D (one less than for the first case). An example for this situation might be a matching applied consecutively to the previous frame (e.g., with register_object_model_3d_pair). To must be empty again. In this case, all neighborhoods that are possible are considered for the global optimization.

Third, you can describe any transformation in HomMats3D by setting From and To to the indices of the 3D object models for which the corresponding transformation is valid. That is, a given transformation describes the transformation that is needed to move the 3D object model with the index that is specified in From into the coordinate system of the 3D object model with the corresponding index that is specified in To. In this case, HomMats3D should contain all possible neighborhood relations between the objects, since no other than these neighborhoods are considered for the optimization. Please consider, that for all 3D object models at least one path of transformations to each other 3D object model must be contained in the such specified transformations.

If ObjectModels3D contains 3D-primitives, they will internally be transformed into point clouds and will be considered as such.

The accuracy of the returned poses is limited to around 0.1% of the size of the point clouds due to numerical reasons. The accuracy further depends on the noise of the data points, the number of data points and the shape of the point clouds.

The process of the global registration can be controlled further by the following generic parameters in GenParamName and GenParamValue:

'default_parameters': Allows to choose between two default parameter sets, i.e., it allows to switch between a *'fast'* and an *'accurate'* set of parameters.
List of values: *'fast'*, *'accurate'*.
Default: *'accurate'*.

'rel_sampling_distance': The relative sampling rate of the 3D object models. This value is relative to the object's diameter and refers to the minimal distance between two sampled points. A higher value leads to faster results, whereas a lower value leads to more accurate results.
Suggested values: *0.03*, *0.05*, *0.07*.
Default: *0.05* (*'default_parameters'* = *'accurate'*), *0.07* (*'default_parameters'* = *'fast'*).
Restriction: 0 < *'rel_sampling_distance'* < 1

'pose_ref_sub_sampling': Number of points that are skipped for the pose refinement. The value specifies the number of points that are skipped per selected point. Increasing this value allows faster convergence at the cost of less accurate results. The internally used method for the refinement is asymmetric and this parameter only affects the second model of each tested pair.
Suggested values: *1*, *2*, *20*.
Default: *2* (*'default_parameters'* = *'accurate'*), *10* (*'default_parameters'* = *'fast'*).
Restriction: *'pose_ref_sub_sampling'* > 0

'max_num_iterations': Number of iterations applied to adjust the initial alignment. The better the initial alignment is, the less iterations are necessary.

Suggested values: *1*, *3*, *10*.

Default: *3*.

Parameters

▷ **ObjectModels3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handles of several 3D object models.

▷ **HomMats3D** (input_control) . . . . . hom_mat3d-array ⇝ *real* / integer
Approximate relative transformations between the 3D object models.

▷ **From** (input_control) . . . . . number(-array) ⇝ *string* / integer
Type of interpretation for the transformations.
Default: 'global'
List of values: From ∈ {'global', 'previous', 0, 1, 2, 3, 4}

▷ **To** (input_control) . . . . . number(-array) ⇝ *integer*
Target indices of the transformations if From specifies the source indices, otherwise the parameter must be empty.
Default: []
List of values: To ∈ {0, 1, 2, 3, 4}

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of the generic parameters that can be adjusted for the global 3D object model registration.
Default: []
List of values: GenParamName ∈ {'default_parameters', 'rel_sampling_distance', 'pose_ref_sub_sampling', 'max_num_iterations'}

▷ **GenParamValue** (input_control) . . . . . number-array ⇝ *real* / integer / string
Values of the generic parameters that can be adjusted for the global 3D object model registration.
Default: []
Suggested values: GenParamValue ∈ {0.03, 0.05, 0.07, 0.1, 0.25, 0.5, 1, 2, 5, 10, 20, 'fast', 'accurate'}

▷ **HomMats3DOut** (output_control) . . . . . hom_mat3d-array ⇝ *real* / integer
Resulting Transformations.

▷ **Scores** (output_control) . . . . . number-array ⇝ *real*
Number of overlapping neighbors for each 3D object model.

Result

register_object_model_3d_global returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, register_object_model_3d_pair, gen_object_model_3d_from_points

Possible Successors

affine_trans_object_model_3d, union_object_model_3d, sample_object_model_3d, triangulate_object_model_3d

See also

register_object_model_3d_pair, find_surface_model, refine_surface_model_pose

Module

3D Metrology

register_object_model_3d_pair (: : ObjectModel3D1, ObjectModel3D2, Method, GenParamName, GenParamValue : Pose, Score)

Search for a transformation between two 3D object models.

register_object_model_3d_pair searches for a transformation between two 3D object models having an optimal alignment. This process is called registration. The transformation that is returned in Pose can be used to transform ObjectModel3D1 to the reference frame of the second object ObjectModel3D2. Score returns the ratio of the overlapping parts to the not overlapping parts of the two 3D object models. If the two objects are not overlapping, no pose is returned. The parameter Method decides if the initial relative position is calculated by *'matching'* or if only the pose refinement is performed in relation to the then assumed common global reference frame, which can be selected directly with *'icp'*.

The accuracy of the returned pose is limited to around 0.1% of the size of the point clouds due to numerical reasons. The accuracy further depends on the noise of the data points, the number of data points and the shape of the point clouds.

The matching process and the following refinement can be controlled using the following name-value pairs in GenParamName and GenParamValue:

'default_parameters': To allow an easy control over the parameters, three different sets of parameters are available. Selecting the *'fast'* parameter set allows a shorter calculation time. *'accurate'* will give more accurate results. *'robust'* additionally improves the quality of the resulting Score at the cost of calculation time.

List of values: *'fast'*, *'accurate'*, *'robust'*.

Default: *'accurate'*.

'rel_sampling_distance': This parameter controls the relative sampling rate of the 3D object models that is used to represent the surfaces for the computation. This value is relative to the diameter of the respective object and defines the minimal distance between two sampled points. A higher value will lead to faster and a lower value to more accurate results. This parameter can also be set for each object independently by using *'rel_sampling_distance_obj1'* and *'rel_sampling_distance_obj2'*.

Suggested values: *0.03*, *0.05*, *0.07*.

Default: *0.05*.

'key_point_fraction': This parameter controls the ratio of sampled points that are considered as key points for the matching process. The number is relative to the sampled points of the model. Reducing this ratio speeds up the process, whereas increasing leads to more robust results. This parameter can be also set for each object independently by using *'key_point_fraction_obj1'* and *'key_point_fraction_obj2'*.

Suggested values: *0.2*, *0.3*, *0.4*.

Default: *0.3*.

'pose_ref_num_steps': The number of iterative steps used for the pose refinement.

Suggested values: *5*, *7*, *10*.

Default: *5*.

'pose_ref_sub_sampling': Number of points that are skipped for the pose refinement. The value specifies the number of points that are skipped per selected point. Increasing this value allows faster convergence at the cost of less accurate results. This parameter is only relevant for the smaller of the two objects.

Suggested values: *1*, *2*, *20*.

Default: *2*.

'pose_ref_dist_threshold_rel': Maximum distance that two faces might have to be considered as potentially overlapping. This value is relative to the diameter of the larger object.

Suggested values: *0.05*, *0.1*, *0.15*.

Default: *0.1*.

'pose_ref_dist_threshold_abs': Maximum distance that two faces might have to be considered as potentially overlapping, as absolute value.

'model_invert_normals': Invert the normals of the smaller object, if its normals are inverted relative to the other object.

List of values: *'true'*, *'false'*.

Default: *'false'*.

Parameters

▷ **ObjectModel3D1** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the first 3D object model.

▷ **ObjectModel3D2** (input_control) . . . . . object_model_3d ⇝ *handle*
Handle of the second 3D object model.

▷ **Method** (input_control) . . . . . string ⇝ *string*
Method for the registration.
Default: 'matching'
List of values: `Method` ∈ {'matching', 'icp'}

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Names of the generic parameters.
Default: []
List of values: `GenParamName` ∈ {'default_parameters', 'rel_sampling_distance', 'rel_sampling_distance_obj1', 'rel_sampling_distance_obj2', 'key_point_fraction', 'key_point_fraction_obj1', 'key_point_fraction_obj2', 'pose_ref_num_steps', 'pose_ref_sub_sampling', 'pose_ref_dist_threshold_rel', 'pose_ref_dist_threshold_abs', 'model_invert_normals'}

▷ **GenParamValue** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Values of the generic parameters.
Default: []
Suggested values: `GenParamValue` ∈ {'fast', 'accurate', 'robust', 0.1, 0.25, 0.5, 1, 'true', 'false'}

▷ **Pose** (output_control) . . . . . pose ⇝ *real* / integer
Pose to transform `ObjectModel3D1` in the reference frame of `ObjectModel3D2`.

▷ **Score** (output_control) . . . . . number-array ⇝ *real*
Overlapping of the two 3D object models.

Example

```
* Generate two boxes
gen_box_object_model_3d ([0,0,0,0,0,0,0],3,2,1, ObjectModel3D1)
gen_box_object_model_3d ([0,0,0.5,15,0,0,0],3,2,1, ObjectModel3D2)
* Match them
register_object_model_3d_pair (ObjectModel3D1, ObjectModel3D2, 'matching',\
                               [], [], Pose, Score)
```

Result

`register_object_model_3d_pair` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_object_model_3d`, `gen_object_model_3d_from_points`, `xyz_to_object_model_3d`

Possible Successors

`register_object_model_3d_global`, `affine_trans_object_model_3d`, `union_object_model_3d`

See also

`register_object_model_3d_global`, `find_surface_model`

Module

3D Metrology

render_object_model_3d (: Image : ObjectModel3D, CamParam, Pose, GenParamName, GenParamValue :)

Render 3D object models to get an image.

render_object_model_3d renders the 3D object models of ObjectModel3D and returns the result in the image Image. To setup the scene to display, set CamParam and the individual Pose of the objects. Be aware that Pose can contain either one pose for each object or one pose for all objects.

The view of the output image is identical to that produced by disp_object_model_3d. The parameters and additional details are documented with disp_object_model_3d, except that the parameters *'object_index_persistence'*, and *'disp_background'* can not be set.

render_object_model_3d requires OpenGL 2.1, GLSL 1.2, and the OpenGL extensions GL_EXT_framebuffer_object and GL_EXT_framebuffer_blit. Otherwise the compatibility mode is automatically enabled. The compatibility mode requires OpenGL 1.1.

Attention

Cameras with hypercentric lenses are not supported.

Parameters

- ▷ **Image** (output_object) ..... multichannel-image ⇝ *object :* byte
  Rendered scene.
- ▷ **ObjectModel3D** (input_control) ..... object_model_3d(-array) ⇝ *handle*
  Handles of the 3D object models.
- ▷ **CamParam** (input_control) ..... campar ⇝ *real* / integer / string
  Camera parameters of the scene.
- ▷ **Pose** (input_control) ..... pose(-array) ⇝ *real* / integer
  3D poses of the objects.
- ▷ **GenParamName** (input_control) ..... string-array ⇝ *string*
  Names of the generic parameters.
  Default: []
  List of values: GenParamName ∈ {'alpha', 'attribute', 'color', 'colored', 'disp_lines', 'disp_pose', 'disp_normals', 'light_position', 'line_color', 'normal_color', 'quality', 'compatibility_mode_enable', 'point_size', 'color_attrib', 'color_attrib_start', 'color_attrib_end', 'red_channel_attrib', 'blue_channel_attrib', 'green_channel_attrib', 'rgb_channel_attrib_start', 'rgb_channel_attrib_end', 'lut'}
- ▷ **GenParamValue** (input_control) ..... string-array ⇝ *string* / integer / real
  Values of the generic parameters.
  Default: []
  List of values: GenParamValue ∈ {'true', 'false', 'coord_x', 'coord_y', 'coord_z', 'normal_x', 'normal_y', 'normal_z', 'red', 'green', 'blue', 'auto', 'faces', 'primitive', 'points', 'lines'}

Result

render_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: mutually exclusive (runs in parallel with other non-exclusive operators, but not with itself).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

find_surface_model, fit_primitives_object_model_3d, segment_object_model_3d, read_object_model_3d, xyz_to_object_model_3d

Possible Successors

disp_obj

See also

disp_object_model_3d, project_shape_model_3d, object_model_3d_to_xyz

Module

3D Metrology

rigid_trans_object_model_3d (: : ObjectModel3D, Pose : ObjectModel3DRigidTrans)

Apply a rigid 3D transformation to 3D object models.

rigid_trans_object_model_3d applies rigid 3D transformations, i.e., rotations and translations, to 3D object models and returns the handles of the transformed 3D object models. The transformations are described by the poses given in Pose, which are in the form ${}^{cst}\mathbf{P}_{mcsi}$, where *mcsi* denotes the coordinate system of the input object model and *cst* the coordinate system of the transformed model, e.g., the coordinate system of the scene (see Transformations / Poses and "Solution Guide III-C - 3D Vision"). A pose can be created using the operators create_pose, pose_invert, etc., or it can be the result of get_object_model_3d_params.

rigid_trans_object_model_3d transforms one or more 3D object models with the same pose if only one transformation matrix is passed in Pose (N:1). If a single 3D object model is passed in ObjectModel3D, it is transformed with all passed poses (1:N). If the number of poses corresponds to the number of 3D object models, every 3D object model is transformed individually with the respective pose (N:N). In those cases, N can be zero, i.e., no pose or no 3D object model can be passed to the operator. In this case, an empty tuple is returned in ObjectModel3DRigidTrans. This can be used to, for example, transform the results of find_surface_model without checking first if at least one match was returned.

Attention

rigid_trans_object_model_3d transforms the attributes of type 3D points, 3D point normals, and the prepared shape model for shape-based 3D matching, as well as 3D primitives. Precomputed data structures for 3D distance computation are not copied. All other attributes are copied without modification.

Parameters

- ▷ **ObjectModel3D** (input_control) ..... object_model_3d(-array) ⇝ *handle*
  Handles of the 3D object models.
- ▷ **Pose** (input_control) ..... pose(-array) ⇝ *real* / integer
  Poses.
- ▷ **ObjectModel3DRigidTrans** (output_control) ..... object_model_3d(-array) ⇝ *handle*
  Handles of the transformed 3D object models.

Result

rigid_trans_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_object_model_3d, xyz_to_object_model_3d, fit_primitives_object_model_3d

Possible Successors

project_object_model_3d, object_model_3d_to_xyz, get_object_model_3d_params

See also

affine_trans_point_3d, affine_trans_object_model_3d

Module

3D Metrology

sample_object_model_3d (: : ObjectModel3D, Method, SamplingParam, GenParamName, GenParamValue : SampledObjectModel3D)

Sample a 3D object model.

sample_object_model_3d creates a sampled version of the 3D object model ObjectModel3D and returns it in SampledObjectModel3D. Depending on the method used, SamplingParam controls the minimum

distance or the number of points in SampledObjectModel3D. The created 3D object model is returned in SampledObjectModel3D.

Using sample_object_model_3d is recommended if complex point clouds are to be thinned out for faster postprocessing or if primitives are to be converted to point clouds. Note that if the 3D object model is triangulated and should be simplified by preserving its original geometry as good as possible, simplify_object_model_3d should be used instead.

If the input object model ObjectModel3D contains only points, several sampling methods are available which can be selected using the parameter Method:

'fast': The default method *'fast'* adds all points from the input model which are not closer than SamplingParam to any point that was earlier added to the output model. If present, normals, XYZ-mapping and extended point attributes are copied to the output model.

'fast_compute_normals': The method *'fast_compute_normals'* selects the same points as the method *'fast'*, but additionally calculates the normals for all points that were selected. For this, the input object model must either contain normals, which are copied, or it must contain a XYZ-mapping attribute from which the normals are computed. The z-component of the calculated normal vectors is always positive. The XYZ-mapping is created by xyz_to_object_model_3d.

'accurate': The method *'accurate'* goes through the points of the 3D object model ObjectModel3D and calculates whether any other points are within a sphere with the radius SamplingParam around the examined point. If there are no other points, the original point is stored in SampledObjectModel3D. If there are other points, the center of gravity of these points (including the original point) is stored in SampledObjectModel3D. This procedure is repeated with the remaining points until there are no points left. Extended attributes of the input 3D object model are not copied, but normals and XYZ-mapping are copied. For this method, a noise removal is possible by specifying a value for *'min_num_points'* in GenParamName and GenParamValue, which removes all interpolated points that had less than the specified number of neighbor points in the original model.

'accurate_use_normals': The method *'accurate_use_normals'* requires normals in the input 3D object model and interpolates only points with similar normals. The similarity depends on the angle between the normals. The threshold of the angle can be specified in GenParamName and GenParamValue with *'max_angle_diff'*. The default value is *180* degrees. Additionally, outliers can be removed as described in the method *'accurate'*, by setting the generic parameter *'min_num_points'*.

'xyz_mapping': The method *'xyz_mapping'* can only be applied to 3D object models that contain an XYZ-mapping (for example, if it was created using xyz_to_object_model_3d). This mapping stores for each 3D point its original image coordinates. The method *'xyz_mapping'* subdivides those original images into squares with side length SamplingParam (which is given in pixel) and selects one 3D point per square.

The method behaves similar to applying zoom_image_factor onto the original XYZ-images. Note that this method does not use the 3D-coordinates of the points for the point selection, only their 2D image coordinates.

It is important to notice that for this method, the parameter SamplingParam corresponds to a distance in pixels, not to a distance in 3D space.

'xyz_mapping_compute_normals': The method *'xyz_mapping_compute_normals'* selects the same points as the method *'xyz_mapping'*, but additionally calculates the normals for all points that were selected. The z-component of the normal vectors is always positive. If the input object model contains normals, those normals are copied to the output. Otherwise, the normals are computed based on the XYZ-mapping.

'furthest_point': The method *'furthest_point'* iteratively adds the point of the input object to the output object that is furthest from all points already added to the output model. This usually leads to a reasonably uniform sampling. For this method, the desired number of points in the output model is passed in SamplingParam. If that number exceeds the number of points in the input object, then all points of the input object are returned.

The first point added to the output object is the point that is furthest away from the center of the axis aligned bounding box around the points of the input object.

'furthest_point_compute_normals': The method *'furthest_point_compute_normals'* selects the same points as the method *'furthest_point'*, but additionally calculates the normals for all points that were selected. The number of desired points in the output object is passed in SamplingParam.

To compute the normals, the input object model must either contain normals, which are copied, or it must contain a XYZ-mapping attribute from which the normals are computed. The z-component of the calculated normal vectors is always positive. The XYZ-mapping is created by xyz_to_object_model_3d.

If the input object model contains faces (triangles or polygons) or is a 3D primitive, the surface is sampled with the given distance. In this case, the method specified in `Method` is ignored. The directions of the computed normals depend on the face orientation of the model. Usually, the orientation of the faces does not vary within one CAD model, which results in a set of normals that is either pointing inwards or outwards. Note that planes and cylinders must have finite extent. If the input object model contains lines, the lines are sampled with the given distance `SamplingParam`.

The sampling process approximates surfaces by creating new points in the output object model. Therefore, any extended attributes from the input object model are discarded.

For mixed input object models, the sampling priority is (from top to bottom) faces, lines, primitives and points, i.e., only the objects of the highest priority are sampled.

The parameter `SamplingParam` accepts either one value, which is then used for all 3D object models passed in `ObjectModel3D`, or one value per input object model. If `SamplingParam` is a distance in 3D space the unit is the usual HALCON-internal unit *'m'*.

Parameters

- ▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
  Handle of the 3D object model to be sampled.
- ▷ **`Method`** (input_control) . . . . . string ⇝ *string*
  Selects between the different subsampling methods.
  Default: 'fast'
  List of values: `Method` ∈ {'fast', 'fast_compute_normals', 'accurate', 'accurate_use_normals', 'xyz_mapping', 'xyz_mapping_compute_normals', 'furthest_point', 'furthest_point_compute_normals'}
- ▷ **`SamplingParam`** (input_control) . . . . . real(-array) ⇝ *real* / integer
  Sampling distance or number of points.
  Number of elements: SamplingParam == 1 || SamplingParam == ObjectModel3D
  Default: 0.05
- ▷ **`GenParamName`** (input_control) . . . . . string-array ⇝ *string*
  Names of the generic parameters that can be adjusted.
  Default: []
  List of values: `GenParamName` ∈ {'min_num_points', 'max_angle_diff'}
- ▷ **`GenParamValue`** (input_control) . . . . . number-array ⇝ *real* / integer / string
  Values of the generic parameters that can be adjusted.
  Default: []
  Suggested values: `GenParamValue` ∈ {1, 2, 5, 10, 20, 0.1, 0.25, 0.5}
- ▷ **`SampledObjectModel3D`** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
  Handle of the 3D object model that contains the sampled points.
  Number of elements: SampledObjectModel3D == ObjectModel3D

Example

```
gen_box_object_model_3d ([0,0,0,0,0,0,0],3,2,1, ObjectModel3D)
sample_object_model_3d (ObjectModel3D, 'fast', 0.05, [], [], \
                        SampledObjectModel3D)
dev_get_window (WindowHandle)
visualize_object_model_3d (WindowHandle, SampledObjectModel3D, \
                           [], [], [], [], [], [], [], PoseOut)
```

Result

`sample_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_object_model_3d, gen_plane_object_model_3d, gen_sphere_object_model_3d, gen_cylinder_object_model_3d, gen_box_object_model_3d, gen_sphere_object_model_3d_center, xyz_to_object_model_3d

Possible Successors

get_object_model_3d_params, clear_object_model_3d

Alternatives

simplify_object_model_3d, smooth_object_model_3d

Module

3D Metrology

simplify_object_model_3d (: : ObjectModel3D, Method, Amount, GenParamName, GenParamValue : SimplifiedObjectModel3D)

Simplify a triangulated 3D object model.

simplify_object_model_3d simplifies the triangulated 3D object model ObjectModel3D by removing model points and returns the result in SimplifiedObjectModel3D. Note that in contrast to sample_object_model_3d points are removed such that the original geometry of the object model is represented as good as possible. Typically, this means that edges are preserved while the point density within smooth parts is reduced. This might be helpful, for example, to speed up subsequent operator calls by using a 3D object model of reduced complexity.

The triangulation of the input 3D object model is preserved as opposed to the operator sample_object_model_3d, which samples surfaces to equidistant unconnected 3D points.

Currently, the operator offers only a single simplification method (*'preserve_point_coordinates'*), which can be set in Method. This method ensures that the points in the simplified object model SimplifiedObjectModel3D have the same coordinates as the respective points in the input object model ObjectModel3D.

simplify_object_model_3d only works for triangulated object models. Whether an object model contains a triangulation can be queried with get_object_model_3d_params (GenParamName=*'has_triangles'*). Object models that do not contain a triangulation must be triangulated beforehand, e.g., by using triangulate_object_model_3d or prepare_object_model_3d (Purpose=*'segmentation'*).

The degree of simplification can be set with Amount. By default, Amount specifies the percentage of points of the input object model that should be contained in the output object model. Thus, the smaller the value of Amount in this case is chosen the stronger the object model will be simplified.

Alternatively, the meaning of the parameter Amount can be modified. For this, the generic parameter *'amount_type'* can be set to one of the following values:

'percentage_remaining' (default): Amount specifies the percentage of points of the input object model that should be contained in the output object model.
Value range: [*0.0* ... *100.0*].

'percentage_to_remove': Amount specifies the percentage of points of the input object model that should be removed.
Value range: [*0.0* ... *100.0*].

'num_points_remaining': Amount specifies the number of points of the input object model that should be contained in the output object model.
Value range: [*0* ... number of points in the input object model].

'num_points_to_remove': Amount specifies the number of points of the input object model that should be removed.
Value range: [*0* ... number of points in the input object model].

Sometimes triangular meshes flip during the simplification, i.e., the direction of their normal vectors changes by 180 degrees. This especially happens for artificially created CAD models that consist of planar parts. To avoid this flipping, the generic parameter *'avoid_triangle_flips'* can be set to *'true'* (the default is *'false'*). Note that in this case, the run-time of simplify_object_model_3d will increase.

Note that multiple calls of simplify_object_model_3d with a lower degree of simplification might result in a different simplified object model compared to a single call with a higher degree of simplification. Also note that isolated (i.e., non-triangulated) points will be removed. This might result in a number of points in SimplifiedObjectModel3D that slightly deviates from the degree of simplification that is specified in Amount.

Parameters

- ▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
  Handle of the 3D object model that should be simplified.
- ▷ **Method** (input_control) . . . . . string ⇝ *string*
  Method that should be used for simplification.
  Default: 'preserve_point_coordinates'
  List of values: Method ∈ {'preserve_point_coordinates'}
- ▷ **Amount** (input_control) . . . . . number ⇝ *real* / integer
  Degree of simplification (default: percentage of remaining model points).
- ▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
  Names of the generic parameters.
  Default: []
  List of values: GenParamName ∈ {'amount_type', 'avoid_triangle_flips'}
- ▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / real
  Values of the generic parameters.
  Default: []
  Suggested values: GenParamValue ∈ {'percentage_remaining', 'percentage_to_remove', 'num_points_remaining', 'num_points_to_remove', 'true', 'false'}
- ▷ **SimplifiedObjectModel3D** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
  Handle of the simplified 3D object model.

Example

```
read_object_model_3d ('mvtec_bunny.om3', 'm', [], [], ObjectModel3D, Status)
visualize_object_model_3d (WindowHandle, ObjectModel3D, [], [], [], [], \
                           [], [], [], Pose)
simplify_object_model_3d (ObjectModel3D, 'preserve_point_coordinates', \
                          5.0, 'amount_type', 'percentage_remaining', \
                          SimplifiedObjectModel3D)
visualize_object_model_3d (WindowHandle, SimplifiedObjectModel3D, [], \
                           Pose, [], [], [], [], [], Pose)
```

Result

simplify_object_model_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

prepare_object_model_3d, read_object_model_3d, triangulate_object_model_3d, xyz_to_object_model_3d

Possible Successors

disp_object_model_3d, smallest_bounding_box_object_model_3d

Alternatives

sample_object_model_3d, smooth_object_model_3d

References

Michael Garland, Paul S. Heckbert: Surface Simplification Using Quadric Error Metrics, Proceedings of the 24th

Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97), 209-216, ACM Press, 1997

Module

3D Metrology

smooth_object_model_3d (: : ObjectModel3D, Method, GenParamName, GenParamValue : SmoothObjectModel3D)

Smooth the 3D points of a 3D object model.

The operator smooth_object_model_3d smoothes the 3D points in ObjectModel3D and returns the smoothed points in SmoothObjectModel3D. Currently, the operator offers three methods for smoothing that can be selected in Method: *'mls'*, *'xyz_mapping'* and *'xyz_mapping_compute_normals'*. *'mls'* applies a Moving Least Squares (MLS) algorithm on the 3D points. As a side effect of the smoothing, the method extends SmoothObjectModel3D by corresponding normals. *'xyz_mapping'* smoothes the coordinates of the 3D points using a 2D filter and the 2D mapping contained in ObjectModel3D. *'xyz_mapping_compute_normals'* applies the same smoothing as *'xyz_mapping'*, but additionally extends SmoothObjectModel3D by normals.

Additional parameters can be set with GenParamName and GenParamValue. The parameter names settable for Method=*'mls'* use the prefix *'mls'*. Analogically, parameter names settable for Method=*'xyz_mapping'* and Method=*'xyz_mapping_compute_normals'* use the prefix *'xyz_mapping'*.

MLS smoothing

By selecting Method=*'mls'*, for each point P, the MLS smoothing algorithm fits a planar surface or a higher order polynomial surface to its k-neighborhood (the k nearest points). The surface fitting is essentially a standard *weighted least squares* parameter estimation of the plane or polynomial surface parameters, respectively. The closest neighbors of P have higher contribution than the other points, which is controlled by the following *weighting function* with a parameter σ:

$$w(P') = exp\left(-\frac{\|P' - P\|^2}{\sigma^2}\right)$$

The point is then projected on the surface. This process is repeated for all points resulting in a smoothed point set. The fitted surfaces have well defined normals (i.e., they can easily be computed from the surface parameters). Therefore, the points are augmented by the corresponding normals as side effect of the smoothing.

Additional parameters can be adjusted for the *MLS smoothing* specifically using the following parameter names and values for GenParamName and GenParamValue:

'mls_kNN': Specify the number of nearest neighbors k that are used to fit the MLS surface to each point.

Suggested values: *40*, *60*, *80*, *100*, *400*.

Default: *60*.

'mls_order': Specify the order of the MLS polynomial surface. For *'mls_order'=1* the surface is a plane.

Suggested values: *1*, *2*,*3*.

Default: *2*.

'mls_abs_sigma': Specify the weighting parameter σ as a fixed absolute value in meter. The value to be selected depends on the scale of the point data. As a rule of thumb, σ can be selected to be the typical distance between a point P and its $k/2$-th neighbor $P_{k/2}$. Note that setting an absolute weighting parameter for point data with varying density might result in different smoothing results for points that are situated in parts of the point data with different densities. This problem can be avoided by using *'mls_relative_sigma'* instead that is scale independent, which makes it also a more convenient way to specify the neighborhood weighting. Note that if *'mls_abs_sigma'* is passed, any value set in *'mls_relative_sigma'* is ignored.

Suggested values: *0.0001*, *0.001*, *0.01*, *0.1*, *1.0*.

'mls_relative_sigma': Specify a multiplication factor σ_{rel} that is used to compute σ_P for a point P by the formula:

$$\sigma_P = \sigma_{rel}\|P_{k/2} - P\|,$$

where $P_{k/2}$ is the $k/2$-th neighbor of P. Note that, unlike σ, which is a global parameter for all points, σ_P is computed for each point P and therefore adapts the weighting function to its neighborhood. This avoids

problems that might appear while trying to set a global parameter σ (*'mls_abs_sigma'*) to a point data with highly varying point density. Note however that if *'mls_abs_sigma'* is set, *'mls_relative_sigma'* is ignored.
Suggested values: *0.1*, *0.5*, *1.0*, *1.5*, *2.0*.
Default: *1.0*.

'mls_force_inwards': If this parameter is set to *'true'*, all surface normals are oriented such that they point "in the direction of the origin". Expressed mathematically, it is ensured that the scalar product between the normal vector and the vector from the respective surface point to the origin is positive. This may be necessary if the resulting `SmoothObjectModel3D` is used for surface-based matching, either as model in `create_surface_model` or as 3D scene in find_surface_model, because here, the consistent orientation of the normals is important for the matching process. If *'mls_force_inwards'* is set to *'false'*, the normal vectors are oriented arbitrarily.
List of values: *'true'*, *'false'*.
Default: *'true'*.

2D mapping smoothing

By selecting `Method`=*'xyz_mapping'* or `Method`=*'xyz_mapping_compute_normals'*, the coordinates of the 3D points are smoothed using a 2D filter and the 2D mapping contained in `ObjectModel3D`. Additionally, for `Method`=*'xyz_mapping_compute_normals'*, `SmoothObjectModel3D` is extended by normals computed from the XYZ-mapping. If no 2D mapping is available, an exception is raised. As the filter operates on the 2D depth image, using `Method`=*'xyz_mapping'* or `Method`=*'xyz_mapping_compute_normals'* is usually faster than using `Method`=*'mls'*. Invalid points (e.g., duplicated points with coordinates [0,0,0]) should be removed from the 3D object model before applying the operator, e.g., by using `select_points_object_model_3d` with attribute *'point_coord_z'* or *'num_neighbors_fast X'*.

Additional parameters can be adjusted for the *2D mapping smoothing* specifically using the following parameter names and values for `GenParamName` and `GenParamValue`:

'xyz_mapping_filter': Specify the filter used for smoothing the 2D mapping. The sizes of the corresponding filter mask are set with *'xyz_mapping_mask_width'* and *'xyz_mapping_mask_height'*.
In the default filter mode *'median_separate'*, the filter method used on the 2D image is comparable to `median_separate`. This mode is usually faster than *'median'*, but can also lead to less accurate results or artifacts at surface discontinuities.
Using filter mode *'median'*, the used filter method is comparable to `median_image`.
List of values: *'median_separate'*, *'median'*.
Default: *'median_separate'*.

'xyz_mapping_mask_width', *'xyz_mapping_mask_height'*: Specify the width and height of the used filter mask.
For *'xyz_mapping_filter'='median_separate'* or *'xyz_mapping_filter'='median'*, even values for *'xyz_mapping_mask_width'* or *'xyz_mapping_mask_height'* are increased to the next odd value automatically.
For *'xyz_mapping_filter'='median'*, the used filter mask must be quadratic (*'xyz_mapping_mask_width'* = *'xyz_mapping_mask_height'*). Thus, when setting only *'xyz_mapping_mask_width'* or *'xyz_mapping_mask_height'*, the other parameter is set to the same value automatically. If two different values are set, an error is raised.
Suggested values: *3*, *5*, *7*, *9*.
Default: *3*.

Parameters

▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model containing 3D point data.

▷ **`Method`** (input_control) . . . . . string ⇝ *string*
Smoothing method.
Default: 'mls'
List of values: `Method` ∈ {'mls', 'xyz_mapping', 'xyz_mapping_compute_normals'}

▷ **`GenParamName`** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of generic smoothing parameters.
Default: []
List of values: `GenParamName` ∈ {'mls_kNN', 'mls_order', 'mls_abs_sigma', 'mls_relative_sigma', 'mls_force_inwards', 'xyz_mapping_filter', 'xyz_mapping_mask_width', 'xyz_mapping_mask_height'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of generic smoothing parameters.
Default: []
Suggested values: GenParamValue ∈ {10, 20, 40, 60, 0.1, 0.5, 1.0, 2.0, 0, 1, 2, 3, 5, 7, 9}

▷ **SmoothObjectModel3D** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model with the smoothed 3D point data.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

surface_normals_object_model_3d, sample_object_model_3d, simplify_object_model_3d

Module

3D Metrology

surface_normals_object_model_3d (: : ObjectModel3D, Method, GenParamName, GenParamValue : ObjectModel3DNormals)

Calculate the 3D surface normals of a 3D object model.

The operator surface_normals_object_model_3d calculates the 3D surface normals for the object ObjectModel3D using the method specified by Method. The calculated normals are appended to the input object and the resulting object is returned in ObjectModel3DNormals.

For Method *'mls'*, the normals estimation method Moving Least Squares (MLS) is applied. The MLS method for normals estimation is essentially identical with the MLS method used by smooth_object_model_3d with the exception that in surface_normals_object_model_3d the 3D points are not smoothed, i.e., the original 3D points of ObjectModel3D remain unchanged. For more details on the MLS as well as a full list and descriptions of the supported MLS parameters refer to smooth_object_model_3d.

If the object ObjectModel3D contains triangles, the Method *'triangles'* can be used to obtain point normals from the normals of the triangles neighboring a point. The normals of the neighboring triangles are weighted according to the angle which the triangle encloses at the point. The triangle normals are returned in the extended attributes '&triangle_normal_x', '&triangle_normal_y' and '&triangle_normal_z'. If the extended attributes already exist, they will not be overwritten.

If the object ObjectModel3D contains a 2D mapping (for example a 3D object model that was created with xyz_to_object_model_3d), the Method *'xyz_mapping'* can be used to obtain point normals from the neighborhood of the points in the 2D mapping. In an 11x11 neighborhood of the points in the 2D mapping, a plane is fit through the corresponding 3D points. The normal of this plane then gets switched in a direction consistent with the 2D mapping, for example along the viewing direction of the sensor or in the opposite direction.

Note that for points where the normal vector cannot be estimated, it is set to the zero vector. This happens, for example, if the 3D object model contains an identical point more than *'mls_kNN'* times.

Parameters

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model containing 3D point data.

▷ **Method** (input_control) . . . . . string ⇝ *string*
Normals calculation method.
Default: 'mls'
List of values: Method ∈ {'mls', 'triangles', 'xyz_mapping'}

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of generic smoothing parameters.
Default: []
List of values: GenParamName ∈ {'mls_kNN', 'mls_order', 'mls_abs_sigma', 'mls_relative_sigma', 'mls_force_inwards'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of generic smoothing parameters.
Default: []
Suggested values: GenParamValue ∈ {10, 20, 40, 60, 0.1, 0.5, 1.0, 2.0, 0, 1, 2, 'true', 'false'}

▷ **ObjectModel3DNormals** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model with calculated 3D normals.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

sample_object_model_3d

Possible Successors

create_surface_model, fuse_object_model_3d

Alternatives

smooth_object_model_3d

Module

3D Metrology

triangulate_object_model_3d (: : ObjectModel3D, Method, GenParamName, GenParamValue : TriangulatedObjectModel3D, Information)

Create a surface triangulation for a 3D object model.

The operator triangulate_object_model_3d generates a surface of triangular faces for the 3D object model ObjectModel3D and returns the resulting surface in TriangulatedObjectModel3D. Currently, the operator offers four methods for the triangulation that can be selected in Method: *'polygon_triangulation'*, *'xyz_mapping'*, *'greedy'* and *'implicit'*. *'polygon_triangulation'* is a simple method for the conversion of a polygonal to a triangular face representation in a 3D object model. *'xyz_mapping'* triangulates the points in 2D according to a 2D mapping. The other two methods are rather complex algorithms that are used to calculate triangular faces from pure 3D point data with unknown surface topology. A detailed comparison of the *'greedy'* and *'implicit'* algorithm is provided in the paragraph *"Comparison of the triangulation methods"* below.

Polygon triangulation

By selecting Method=*'polygon_triangulation'*, all polygons in ObjectModel3D are triangulated. No generic parameters are supported for this method. If no polygons are available, an exception is raised. A triangular mesh representing the same surface as ObjectModel3D is returned in TriangulatedObjectModel3D.

2D mapping triangulation

By selecting Method=*'xyz_mapping'*, the points are triangulated in 2D according to a 2D mapping contained in ObjectModel3D. The used method is the same as in prepare_object_model_3d for Purpose='segmentation'. If no 2D mapping is available, an exception is raised.

As a post-processing step, triangles whose normal differs strongly from a specified direction can be removed, refer to the description of GenParamName below. This is helpful in cases where the 2D neighborhood used for the triangulation does not reflect the 3D neighborhood well, e.g., when parts of the surface are hidden along the viewing direction of the sensor, or to remove typical noise along the viewing direction of the sensor.

By setting GenParamName to the following value, the additional parameter specific for the *2D mapping triangulation* can be set with GenParamValue:

'xyz_mapping_max_area_holes' specifies which area holes of the point coordinates are closed during a simple Delaunay triangulation. Only holes which are completely surrounded by the image region are closed. If *'xyz_mapping_max_area_holes'* is set to 0, no holes are triangulated. The parameter corresponds to the GenParamName *'max_area_holes'* of prepare_object_model_3d.

Suggested values: *1*, *10*, *100*.

Default: *10*.

'xyz_mapping_max_view_angle' specifies the maximum allowed angle difference between the triangle normal and the viewing direction of the sensor. The smaller this value is set, the fewer triangles are returned. The viewing direction of the sensor is assumed to be the z-axis in the coordinate system of `ObjectModel3D` if not specified differently using `GenParamName` set to *'xyz_mapping_max_view_dir_x'*, *'xyz_mapping_max_view_dir_y'*, and *'xyz_mapping_max_view_dir_z'*. The angle has to be specified between 0 and 90 degrees.

Suggested values: *'rad(60)'*, *'rad(85)'*, *'rad(90)'*.

Default: *'rad(90)'*.

'xyz_mapping_max_view_dir_x', *'xyz_mapping_max_view_dir_y'*, *'xyz_mapping_max_view_dir_z'* specify the viewing direction of the sensor for use with the `GenParamName` *'xyz_mapping_max_view_angle'*, in the coordinate system of `ObjectModel3D`. If not all three coordinate directions are set simultaneously or the direction equals the zero vector, an exception is raised.

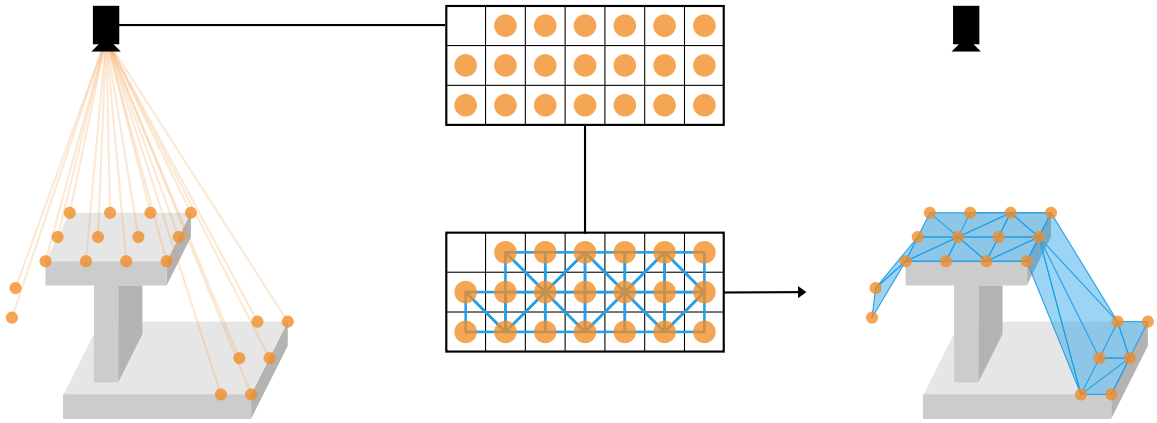
Suggested values: *[1, 0, 0]*, *[0, 0, 1]*.

Default: *[0, 0, 1]*.

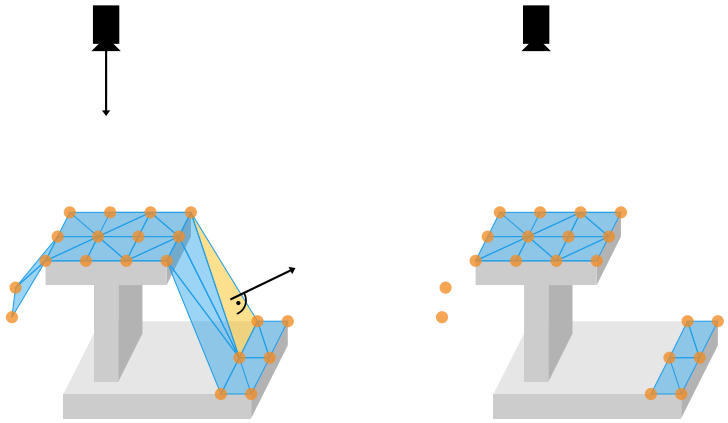
'xyz_mapping_output_all_points' controls, if all input points are returned, regardless whether they were used in the output triangulation or not. Mainly provided for reasons of backward compatibility. When *'xyz_mapping_output_all_points'* is set to *'false'*, the old point indices are stored as an extended attribute named *'original_point_indices'* in the 3D object model `TriangulatedObjectModel3D`. This attribute can subsequently be queried with `get_object_model_3d_params` or be processed with other operators that use extended attributes.

List of values: *'false'*, *'true'*.

Default: *'false'*.



(1)



(2)

(1) In order to triangulate the 3D object model, a 2D mapping of the model is used. The triangulation is based on the respective 2D neighborhood. Thereby it is possible, that unwanted triangles are created along the sensor's direction of view, e.g., because of hidden object parts or clutter data. (2) Whether or not a triangle is

returned, is decided by computing the difference between the normal direction of each triangle and the viewing direction. The maximum deviation is specified in *'xyz_mapping_max_view_angle'*.

Greedy triangulation

By selecting `Method`=*'greedy'*, a so called greedy triangulation algorithm is invoked. It requires 3D point data containing normals. If `ObjectModel3D` does not contain the normals, they are calculated internally, in an identical manner to calling `surface_normals_object_model_3d` with its default parameters before triangulation. The algorithm constructs a surface, which passes through the points and whose surface normals must be conform to the corresponding point normals up to a given tolerance. The surface is represented by triangular faces, which are constructed from triplets of *neighboring* points. In order to determine which triplets qualify for a surface triangle, the algorithm applies for each point pair the following local neighborhood test, denoted as **surface neighborhood criteria (SNC)**:

If a point P is lying on a surface, with N being the orientation (normal) of the surface, then a point P' with normal N' is considered to lay on this surface if:

1. the distance between both points is smaller or equal to r, i.e., $\Delta(P, P') \leq r$
2. both normals have similar orientation, i.e., the angle $\angle(N, N') \leq \alpha$ or - if no strict consistency of the normals is enforced - $\angle(N, -N') \leq \alpha$
3. the vector $\delta P = P' - P$ is close to orthogonal with respect to N, i.e., the angle $|90° - \angle(N, \delta P)| \leq \beta$
4. if P' does not meet 3. but it is not further away from the plane defined by $[P, N]$ than d, then it is accepted as well.

The four parameters r (see *'greedy_radius_type'* and *'greedy_radius_value'*), α (see *'greedy_neigh_orient_tol'*), β (see *'greedy_neigh_latitude_tol'*), and d (see *'greedy_neigh_vertical_tol'*) control the criteria and have the following meaning:

The parameter α essentially controls the curvature of the generated surface: for small values of α the generated surface will be locally flatter; larger values of α permit the generation of more curved surface fragments.

The other three parameters define a portion of a sphere that defines the valid SNC neighborhood. The sphere has a radius r, it is centered in P, and its equatorial plane is incident with the plane $[P, N]$. Only points that are within the sphere (first SNC criteria) are considered. Furthermore, they need to have a latitude within [-β; β] (third SNC criteria) with respect to the equator unless they are lying within the thin layer defined on the both sides of the equatorial plane by the distance parameter d (fourth SNC criteria). In contrast, points lying in any of both pole segments of the sphere (i.e., with higher latitude than β and a distance from the equatorial plane beyond d) are not considered as neighbors.

The parameter r prevents the algorithm from constructing too big triangles. This is particularly important for point sets that represent several disconnected surface pieces or a surface with holes that must not be closed. The latitude window defined by β enables neighbors which deviate from $[P, N]$ due to noise or curvature to be considered as well. Similarly, the parameter d enables neighbors right "above" or "below" the equatorial plane to be accepted, which essentially accounts for data noise.

Here is some advice for selecting the appropriate values for these parameters:

- If the resulting surface triangulation looks very disconnected or exhibits many holes, this might be a hint that r is too small and thus restricts the generation of triangles that are large enough to close the holes. Try to increase r.
- If the normals data is noisy (i.e., neighboring normals are deviating to a large extend from each other), then increase α. The source of noisy normals is typically caused either by the sensor, which delivers both the point and the normals data, or an imprecise normals estimation routine, which computes the normals from the point data.
- If the point data represents a very curved surface, i.e., it exhibits a very fine structure like, e.g., little buckles, fine waves or folds, or sharp turns, then make sure the generation of curved data is facilitated by an increasing α and/or β.
- In contrast, if the data is rather planar but has lots of outliers (i.e., points laying next to the surface, which have completely different orientations and thus most probably do not belong to it), then decrease α to exclude them from the surface generation.
- If the point data is very noisy and resembles more a crust than a single-layer surface, then increase β and/or d to make sure that neighbors for P can still be found even if they are further away from the optimal plane $[P, N]$.

- In contrast, if the data is rather noise-free, but two surfaces are running close to each other and are nearly parallel, e.g., surfaces representing the front and the back side of a thin, plate-like object, then decrease β and d to avoid interference between the surfaces.

The **greedy triangulation** algorithm starts by initializing a surface with one triangle constructed from three SNC-eligible, neighboring points. If all valid neighborhoods show local inconsistencies like collinear or 'double' points, an error will be raised. A prior call of `sample_object_model_3d` with Method set to 'fast' and a small SamplingParam will remove most local inconsistencies from `ObjectModel3D`. Having found one triangle, the algorithm then greedily constructs new triangles as long as further points can be reached by the SNC rules from any point on the surface boundaries. If no points can be reached from the current surface, but there are unprocessed points in the 3D object model, a new surface is initialized. Because the SNC rules are essentially defined only in the small local neighborhoods of the points, the resulting surface can have global topological artifacts like holes and flips. The latter occur, when - while it is growing - a surface meets itself but with inverted face orientations (i.e., the surface was *flipped* somewhere while it was growing). These artifacts are handled in special post-processing steps: *hole filling* and *flip resolving*, respectively.

Finally, a *mesh morphology* can be performed to additionally remove artifacts that occurred on the final surface boundaries. The mesh morphology consists of several mesh erosion cycles and several subsequent mesh dilation cycles. With each erosion cycle, all triangles reachable from the surface boundaries are removed and the surface boundaries shrink. Then, with each dilation cycle all triangles reachable from the surface boundaries are appended again to the surface and the boundaries expand. Note that this is only possible for triangles, which were removed by an erosion cycle before that. Therefore, once the original boundaries of the surface (i.e., those which existed before the mesh erosion cycles) are reached, the dilation cannot advance any further and hence the dilation cycles cannot be more than the erosion cycles. Applying mesh erosion and dilation subsequently is analogous to performing opening to standard HALCON regions. At last, the mesh morphology can delete surface pieces which have too few triangles.

The individual algorithm steps are summarized here:

1. Triangulation of all points reachable by SNC
2. Hole filling (see *'greedy_hole_filling'*)
3. Flip resolving (see *'greedy_fix_flips'*)
4. Mesh morphology (see *'greedy_mesh_erosion'*, *'greedy_mesh_dilation'*, and *'greedy_remove_small_surfaces'*)

By setting `GenParamName` to one of the following values, additional parameters specific for the *greedy triangulation* can be set with `GenParamValue`:

'greedy_kNN' specifies the size k of the neighborhood. While looking for reachable SNC neighbors for a surface boundary point, the algorithm considers only its closest k neighbors.

Suggested values: *20*, *30*, *40*, *50*, *60*.

Default: *40*.

'greedy_radius_type': if set to *'fixed'*,*'greedy_radius_value'* specifies the SNC radius r in meter units.

If set to *'z_factor'*, r is calculated for each point P by multiplying its z-coordinate by the value specified by *'greedy_radius_value'*. This representation of r is appropriate for data where the density of the points correlates with their distance from the sensor they were recorded with. This is typically the case with depth sensors or TOF cameras.

If set to *'auto'*, the algorithm determines internally whether to use a *'fixed'* or a *'z_factor'* radius and estimates its value. The estimated value is then multiplied by the value specified in *'greedy_radius_value'*. This way, the user specifies a scale factor for the estimated radius.

List of values: *'auto'*, *'fixed'*, *'z_factor'*.

Default: *'auto'*.

'greedy_radius_value': see *'greedy_radius_type'*.

Suggested values: *0.01*, *0.05*, *0.5*, *0.66*, *1.0*, *1.5*, *2.0*, *3.0*, *4.0*

'greedy_neigh_orient_tol': sets the SNC parameter α in degree units. α controls the surface curvature as described with the SNC rules above.

Suggested values: *10*, *20*, *30*, *40*.

Default: *30*.

'greedy_neigh_orient_consistent': enforces that the normals of two neighboring points have the same orientation (i.e., they do not show in opposite directions). If enabled, this parameter disables the second part of the SNC criteria for α, i.e., if $\angle(N, N') > \alpha$, the criteria fails even if $\angle(N, -N') \leq \alpha$.

List of values: *'true'*, *'false'*.

Default: *'false'*.

'greedy_neigh_latitude_tol': sets the SNC parameter β in degree units. β controls the surface neighborhood latitude window as described with the SNC rules above.

Suggested values: *10*, *20*, *30 40*.

Default: *30*.

'greedy_neigh_vertical_tol': sets the SNC parameter d as a factor of the radius r.

Suggested values: *0.01*, *0.1*, *0.2*, *0.3*.

Default: *0.1*.

'greedy_hole_filling': sets the length of surface boundaries (in number of point vertices) that should be considered for the hole filling. If *'false'* is specified, then the hole filling step is disabled.

Suggested values: *'false'*, *20*, *40*, *60*.

Default: *40*.

'greedy_fix_flips': enables/disables the flip resolving step of the algorithm.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'greedy_prefetch_neighbors': enables/disables prefetching of lists of the k nearest neighbors for all points. This prefetching improves the algorithm speed, but has high memory requirements ($O(kn)$, where k is the number specified by *'greedy_kNN'*, and n is the number of points in `ObjectModel3D`). For very large data, it might be impossible to preallocate such a big amount of memory, results in a memory error message. In such a case the prefetching must be disabled.

List of values: *'true'*, *'false'* **Default:** *'true'*.

'greedy_mesh_erosion': specifies the number of erosion cycles applied to the final mesh.

Suggested values: *0*, *1*, *2*, *3*.

Default: *0*.

'greedy_mesh_dilation': specifies the number of dilation cycles. The mesh dilation is applied after the mesh erosion. If *'greedy_mesh_dilation'* is set to a greater value than *'greedy_mesh_erosion'*, it will be reduced internally to the value of *'greedy_mesh_erosion'*.

Suggested values: *0*, *1*, *2*, *3* **Default:** *0*.

'greedy_remove_small_surfaces': controls the criteria for removing small surface pieces. If set to *'false'*, the small surface removal is disabled. If set to a value between *0.0* and *1.0*, all surfaces having less triangles than *'greedy_remove_small_surfaces'*×`num_triangles` will be removed, where `num_triangles` is the total number of triangles generated by the algorithm. If set to a value greater than *1*, all surfaces having less triangles than *'greedy_remove_small_surfaces'* will be removed.

Suggested values: *'false'*, *0.01*, *0.05*, *0.1*, *10*, *100*, *1000*, *10000*.

Default: *'false'*.

'greedy_timeout': using a timeout, it is possible to interrupt the operator after a defined period of time in seconds. This is especially useful in cases, where a maximum cycle time has to be ensured. The temporal accuracy of this interrupt is about 10 ms. Passing values less then zero is not valid. Setting *'greedy_timeout'* to *'false'* deactivates the timeout, which corresponds to the default. The temporal accuracy depends on several factors including the size of the model, the speed of your computer, and the *'timer_mode'* set via `set_system`.

Suggested values: *'false'*, *0.1*, *0.5*, *1*, *10*, *100*.

Default: *'false'*.

'greedy_suppress_timeout_error': by default, if a timeout occurs the operator returns a timeout error code. By setting *'greedy_suppress_timeout_error'* to *'true'* instead, the operator returns no error and the intermediate results of the triangulation are returned in `TriangulatedObjectModel3D`. With the error suppressed, the occurrence of a timeout can be checked by querying the list of values returned in `Information` (in *'verbose'* mode) by looking for the value corresponding to *'timeout_occured'*.

List of values: *'false'*, *'true'*.

Default: *'false'*.

'greedy_output_all_points': controls, if all input points are returned, regardless whether they were used in the output triangulation or not. Mainly provided for reasons of backward compatibility. When *'greedy_output_all_points'* is set to *'false'*, the old point indices are stored as an extended attribute named *'original_point_indices'* in the 3D object model `TriangulatedObjectModel3D`. This attribute can subsequently be queried with `get_object_model_3d_params` or be processed with other operators that use extended attributes.

List of values: *'false'*, *'true'*.

Default: *'false'*.

'information': specifies, which intermediate results shall be reported in `Information`. By default (*'information'*=*'num_triangles'*), the number of generated triangles is reported. For *'information'*=*'verbose'*, a list of name-value information pairs is returned. Currently, the following information is reported:

Name	Value	Description
'num_triangles'	<number of triangles>	returns the number of generated triangular faces.
'specified_radius_type'	*'auto'* | *'fixed'* | *'z_factor'* | *'none'*	returns the radius type as specified by the user.
'specified_radius_value'	<specified radius value>	returns the radius value specified by the user.
'used_radius_type'	*'fixed'* | *'z_factor'* | *'sampling'*	returns the radius type used internally; if the user specified *'auto'* for *'specified_radius_type'*, this field returns the radius type that was selected internally; if `ObjectModel3D` is a 3D primitive, the user specified radius value is internally used as a sampling step and *'used_radius_type'* returns *'sampling'*.
'used_radius_value'	<used radius value>	returns the radius value used internally; if *'used_radius_type'='fixed'*, the absolute neighborhood radius in meters is reported; if *'used_radius_type'='z_factor'*, the multiplication factor is reported, which is used to compute the neighborhood radius from the z-coordinate of the neighborhood center point; if *'used_radius_type'='sampling'*, then the sub-sampling factor is reported, which is used to generate the triangulation of 3D primitives, in particular: cylinder and sphere.
'neigh_orient_tol'	<α>	returns the surface curvature parameter α in degrees that was used for the triangulation.
'neigh_latitude_tol'	<β>	returns the angular tolerance window in degrees that was used to select surface neighbors.
'neigh_vertical_tol'	<d>	returns the neighborhood parameter d as a factor of the used radius.
'fix_flips'	*'true'* | *'false'*	returns whether the flip fixing was enabled.
'hole_filling'	*'false'* | <max hole boundary length>	returns *'false'* when the hole filling was disabled, or the specified maximal hole boundary length in number of points.
'timeout'	*'false'* | <timeout>	*'false'* when the timeout was disabled, or the specified timeout in seconds.
'timeout_occured'	*'yes'* | *'no'*	returns whether a timeout occurred.

List of values: *'num_triangles'*, *'verbose'*.
Default: *'num_triangles'*.

Implicit triangulation

By selecting `Method`=*'implicit'* an implicit triangulation algorithm based on a Poisson solver (see the paper in References) is invoked. It constructs a water-tight surface, i.e., it is completely closed. The implicit triangulation requires 3D point data containing normals. Additionally, it is required that the 3D normals are pointing strictly inwards or strictly outwards regarding the volume enclosed by the surface to be reconstructed. Unlike the *'greedy'* algorithm, the *'implicit'* algorithm does not construct the surface through the input 3D points. Instead, it constructs

a surface that approximates the original 3D data and creates a new set of 3D points lying on this surface.

First, the algorithm organizes the point data in an adaptive octree structure: the volume of the bounding box containing the point data is split in the middle in each dimension resulting in eight sub-volumes, or *octree voxels*. Voxels still containing enough point data can be split in further eight sub-voxels. Voxels that contain no or just few points must not be split further. This splitting is repeated recursively in regions of dense 3D point data until the resulting voxels contain no or just few points. The recursion level of the voxel splits, reached with the smallest voxels, is denoted as *depth* of the octree.

In the next step, the algorithm estimates the values of the so-called *implicit indicator function* of the surface, based on the assumption that the points from `ObjectModel3D` are lying on the surface of an object and the normals of the points in `ObjectModel3D` are pointing inwards that object (see the paper in References). This assumption explains the requirement of mutually consistent normal orientations. The implicit function has a value of 1 in voxel corners that are strictly inside the body and 0 for voxel corners strictly outside of it. Due to noisy data, voxel corners that are close to the boundary of the object cannot be 'labeled' unambiguously. Therefore, they receive a value between 0 and 1.

The *implicit surface* defined by the indicator function is a surface, such that each point lying on it has an indicator value of 0.5. The implicit algorithm uses a standard *marching cubes algorithm* to compute the intersection points of the implicit surface with the sides of the octree voxels. The intersection points result in the new set of 3D points spanning the surface returned in `TriangulatedObjectModel3D`. As a consequence, the resolution of the surface details reconstructed in `TriangulatedObjectModel3D` depends directly on the resolution of the octree (i.e., on its *depth*).

By setting `GenParamName` to one of the following values, additional parameters specific for the *implicit triangulation* can be set with `GenParamValue`:

'implicit_octree_depth': sets the depth of the octree. The octree depth controls the resolution of the surface generation - a higher depth leads to a higher surface resolution. The octree depth has an exponential effect on the runtime and an exponential effect on the memory requirements of the octree. Therefore, the depth is limited to *12*.

Restriction: $5 \leq$ *'implicit_octree_depth'* ≤ 12.

Suggested values: *5*, *6*, *8*, *10*, *11*, *12*.

Default: *6*.

'implicit_solver_depth': enables an alternative algorithm, which can prepare the implicit function up to a user specified octree depth, before the original algorithm takes over the rest of the computations. This algorithm requires less memory than the original one, but is a bit slower.

Restriction: *'implicit_solver_depth'* $\leq$ *'implicit_octree_depth'*.

Suggested values: *2*, *4*, *6*, *8*, *10*, *11*, *12*.

Default: *6*.

'implicit_min_num_samples': sets the minimal number of point samples required per octree voxel node. If the number of points in a voxel is less than this value, the voxel is not split any further. For noise free data, this value can be set low (e.g., between 1-5). For noisy data, this value should be set higher (e.g., 10-20), such that the noisy data is accumulated in single voxel nodes to smooth the noise.

Suggested values: *1 5*, *10*, *15*, *20*, *30*.

Default: *1*.

'information': specifies, which intermediate results shall be reported in `Information`. By default (*'information'='num_triangles'*), the number of generated triangles is reported. For *'information'='verbose'*, a list of name-value information pairs is returned. Currently, the following information is reported:

Name	Value	Description
'num_triangles'	<number of triangles>	returns the number of generated triangular faces.
'num_points'	<number of points>	returns the number of generated points.

List of values: *'num_triangles'*, *'verbose'*.

Default: *'num_triangles'*.

Comparison of the triangulation methods

In this paragraph, a simple comparison of both supported triangulation methods is provided:

Property	Greedy triangulation	Implicit triangulation
required data:	3D points with 3D normals	3D points with 3D normals, the normals must point consistently inwards
resulting surface:	open, triangulation of the input points	closed (water-tight), approximation of the input points
resulting point data:	the input point data is preserved	new point data is generated
noise handling:	moderate point and normal noise handled properly	point and normal noise handled implicitly; moderate and high noise levels are accepted
triangulation resolution:	explicit, controlled by surface neighborhood parameters	implicit, controlled by octree depth and minimal number of point samples per node
time complexity:	$O(Nk \log N)$	$O(ND^3)$
memory complexity:	$O(Nk)$, with neighborhood prefetching $O(N)$, without neighborhood prefetching	$O(D^3)$ *where:* N: *number of points* k: *size of the neighborhood* D: *depth of the octree*

Depending on the number of points in `ObjectModel3D`, noise, and specific structure of the data, both algorithms deliver different results and perform with different time and memory complexity. The greedy algorithm works fast, requires less memory, and returns a high level of details in the reconstructed surface for rather small data sets (up to, e.g., 500.000 points). Since the algorithm must basically process every single point in the data, its time performance cannot be decoupled from the point number and it can be rather time consuming for more than 500.000 points. If large point sets need to be triangulated with this method anyway, it is recommended to first sub-sample them via `sample_object_model_3d`.

In contrast, as described above, the implicit algorithm organizes all points in an underlying octree. Therefore, the details returned by it, its speed, and its memory consumption are dominated by the depth of the octree. While higher levels of surface details can only be achieved at disproportionately higher time and memory costs, the octree offers the advantage that it handles large point sets more efficiently. With the octree, the performance of the implicit algorithm depends mostly on the depth of the octree and to a lesser degree on the number of points to be processed. One further disadvantage of the implicit algorithm is its requirement that the adjacent point normals are strictly consistent. This requirement can seldom be fulfilled by usual normal estimation routines.

Parameters

▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model containing 3D point data.

▷ **`Method`** (input_control) . . . . . string ⇝ *string*
Triangulation method.
Default: 'greedy'
List of values: `Method` ∈ {'greedy', 'implicit', 'polygon_triangulation', 'xyz_mapping'}

▷ **`GenParamName`** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic triangulation parameters.
Default: []
List of values: `GenParamName` ∈ {'information', 'implicit_octree_depth', 'implicit_solver_depth', 'implicit_min_num_samples', 'greedy_radius_type', 'greedy_radius_value', 'greedy_kNN', 'greedy_neigh_orient_tol', 'greedy_neigh_orient_consistent', 'greedy_neigh_vertical_tol', 'greedy_neigh_latitude_tol', 'greedy_hole_filling', 'greedy_fix_flips', 'greedy_mesh_erosion', 'greedy_mesh_dilation', 'greedy_remove_small_surfaces', 'greedy_prefetch_neighbors', 'greedy_timeout', 'greedy_suppress_timeout_error', 'greedy_output_all_points', 'xyz_mapping_max_area_holes', 'xyz_mapping_output_all_points', 'xyz_mapping_max_view_angle', 'xyz_mapping_max_view_dir_x', 'xyz_mapping_max_view_dir_y', 'xyz_mapping_max_view_dir_z'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of the generic triangulation parameters.
Default: []
Suggested values: GenParamValue ∈ {6, 8, 12, 'true', 'false', 'auto', 'fixed', 'z_factor', 'verbose', 'num_triangles'}

▷ **TriangulatedObjectModel3D** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model with the triangulated surface.

▷ **Information** (output_control) . . . . . number(-array) ⇝ *integer* / string
Additional information about the triangulation process.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_object_model_3d, gen_plane_object_model_3d, gen_sphere_object_model_3d, gen_cylinder_object_model_3d, gen_box_object_model_3d, gen_sphere_object_model_3d_center, sample_object_model_3d

Possible Successors

write_object_model_3d, render_object_model_3d, project_object_model_3d, simplify_object_model_3d

References

M. Kazhdan, M. Bolitho, and H. Hoppe: "Poisson Surface Reconstruction." Symposium on Geometry Processing (June 2006).

Module

3D Metrology

xyz_to_object_model_3d (X, Y, Z : : : ObjectModel3D)

Transform 3D points from images to a 3D object model.

The operator xyz_to_object_model_3d transforms an image triple that contains the X, Y, and Z-coordinates of 3D points to a 3D object model. Thereby, only points in the intersecting domains of all three images are used and the images need to be of same size. The size of these images can be queried from the model by get_object_model_3d_params with *'mapping_size'*. The handle of the created 3D object model is returned in ObjectModel3D. The created 3D object model contains the coordinates of the points, as well as a mapping attribute that contains the original row and column of each 3D point. Points where one of the coordinates is infinity or "Not a Number" (NaN) are ignored and not added to the 3D object model.

Parameters

▷ **X** (input_object) . . . . . singlechannelimage ⇝ *object : real*
Image with the X-Coordinates and the ROI of the 3D points.

▷ **Y** (input_object) . . . . . singlechannelimage ⇝ *object : real*
Image with the Y-Coordinates of the 3D points.

▷ **Z** (input_object) . . . . . singlechannelimage ⇝ *object : real*
Image with the Z-Coordinates of the 3D points.

▷ **ObjectModel3D** (output_control) . . . . . object_model_3d ⇝ *handle*
Handle of the 3D object model.

Result

The operator xyz_to_object_model_3d returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`disparity_image_to_xyz`, `get_sheet_of_light_result`

Alternatives

`gen_object_model_3d_from_points`, `get_sheet_of_light_result_object_model_3d`

See also

`read_object_model_3d`

Module

3D Metrology

Chapter 5

3D Reconstruction

5.1 Binocular Stereo

```
binocular_disparity ( ImageRect1, ImageRect2 : Disparity,
    Score : Method, MaskWidth, MaskHeight, TextureThresh,
    MinDisparity, MaxDisparity, NumLevels, ScoreThresh, Filter,
    SubDisparity : )
```

Compute the disparities of a rectified image pair using correlation techniques.

`binocular_disparity` computes pixel-wise correspondences between two rectified images using correlation techniques. Different from `binocular_distance` the results are not transformed into distance values.

The algorithm requires a reference image `ImageRect1` and a search image `ImageRect2` which must be rectified, i.e., corresponding epipolar lines are parallel and lie on identical image rows ($r_1 = r_2$). In case this assumption is violated the images can be rectified by using the operators `calibrate_cameras`, `gen_binocular_rectification_map`, and `map_image`. Hence, given a pixel in the reference image `ImageRect1` the homologous pixel in `ImageRect2` is selected by searching along the corresponding row in `ImageRect2` and matching a local neighborhood within a rectangular window of size `MaskWidth` and `MaskHeight`. The pixel correspondences are returned in the single-channel `Disparity` image $d(r_1, c_1)$ which specifies for each pixel $(r1, c1)$ of the reference image `ImageRect1` a suitable matching pixel $(r2, c2)$ of `ImageRect2` according to the equation $c_2 = c_1 + d(r_1, c_1)$. A quality measure for each disparity value is returned in `Score`, containing the best result of the matching function S of a reference pixel. For the matching, the gray values of the original unprocessed images are used.

The used matching function is defined by the parameter `Method` allocating three different kinds of correlation:

- *'sad'*: Summed Absolute Differences

  $$S(r,c,d) = \frac{1}{N} \sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} \mid g_1(r',c') - g_2(r',c'+d) \mid,$$

  with $0 \leq S(r,c,d) \leq 255$.
- *'ssd'*: Summed Squared Differences

  $$S(r,c,d) = \frac{1}{N} \sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} (g_1(r',c') - g_2(r',c'+d))^2,$$

  with $0 \leq S(r,c,d) \leq 65025$.
- *'ncc'*: Normalized Cross Correlation

  $$S(r,c,d) = \frac{\sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} (g_1(r',c') - \bar{g_1}(r,c))(g_2(r',c'+d) - \bar{g_2}(r,c+d))}{\sqrt{\left(\sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} (g_1(r',c') - \bar{g_1}(r,c))^2\right)\left(\sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} (g_2(r',c'+d) - \bar{g_2}(r,c+d))^2\right)}},$$

  with $-1.0 \leq S(r,c,d) \leq 1.0$.

with

$r1$, $c1$, $r2$, $c2$: row and column coordinates of the corresponding pixels of the two input images,

$g1$, $g2$: gray values of the unprocessed input images,

$N = (2m+1)(2n+1)$: size of correlation window

$\bar{g}(r,c) = \frac{1}{N} \sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} g(r',c')$: mean value within the correlation window of width $2m+1$ and height $2n+1$.

Note that the methods *'sad'* and *'ssd'* compare the gray values of the pixels within a mask window directly, whereas *'ncc'* compensates for the mean gray value and its variance within the mask window. Therefore, if the two images differ in brightness and contrast, this method should be preferred. For images with similar brightness and contrast *'sad'* and *'ssd'* are to be preferred as they are faster because of less complex internal computations.

It should be noted, that the quality of correlation for rising S is falling in methods *'sad'* and *'ssd'* (the best quality value is 0) but rising in method *'ncc'* (the best quality value is 1.0).

The size of the correlation window, referenced by $2m+1$ and $2n+1$, has to be odd numbered and is passed in `MaskWidth` and `MaskHeight`. The search space is confined by the minimum and maximum disparity value `MinDisparity` and `MaxDisparity`. Due to pixel values not defined beyond the image border the resulting domain of `Disparity` and `Score` is not set along the image border within a margin of height (`MaskHeight`-1)/2 at the top and bottom border and of width (`MaskWidth`-1)/2 at the left and right border. For the same reason, the maximum disparity range is reduced at the left and right image border.

Since matching turns out to be highly unreliable when dealing with poorly textured areas, the minimum statistical spread of gray values within the correlation window can be defined in `TextureThresh`. This threshold is applied on both input images `ImageRect1` and `ImageRect2`. In addition, `ScoreThresh` guarantees the matching quality and defines the maximum (*'sad'*,*'ssd'*) or, respectively, minimum (*'ncc'*) score value of the correlation function. Setting `Filter` to *'left_right_check'*, moreover, increases the robustness of the returned matches, as the result relies on a concurrent direct and reverse match, whereas *'none'* switches it off.

The number of pyramid levels used to improve the time response of `binocular_disparity` is determined by `NumLevels`. Following a coarse-to-fine scheme disparity images of higher levels are computed and segmented into rectangular subimages of similar disparity to reduce the disparity range on the next lower pyramid level. `TextureThresh` and `ScoreThresh` are applied on every level and the returned domain of the `Disparity` and `Score` images arises from the intersection of the resulting domains of every single level. Generally, pyramid structures are the more advantageous the more the disparity image can be segmented into regions of homogeneous disparities and the bigger the disparity range is specified. As a drawback, coarse pyramid levels might loose important texture information which can result in deficient disparity values.

Finally, the value *'interpolation'* for parameter `SubDisparity` performs subpixel refinement of disparities. It is switched off by setting the parameter to *'none'*.

Parameters

- ▷ **`ImageRect1`** (input_object) ..... singlechannelimage ⇝ *object :* byte
  Rectified image of camera 1.
- ▷ **`ImageRect2`** (input_object) ..... singlechannelimage ⇝ *object :* byte
  Rectified image of camera 2.
- ▷ **`Disparity`** (output_object) ..... singlechannelimage ⇝ *object :* real
  Disparity map.
- ▷ **`Score`** (output_object) ..... singlechannelimage ⇝ *object :* real
  Evaluation of the disparity values.
- ▷ **`Method`** (input_control) ..... string ⇝ *string*
  Matching function.
  Default: 'ncc'
  List of values: `Method` ∈ {'sad', 'ssd', 'ncc'}
- ▷ **`MaskWidth`** (input_control) ..... integer ⇝ *integer*
  Width of the correlation window.
  Default: 11
  Suggested values: `MaskWidth` ∈ {5, 7, 9, 11, 21}
  Restriction: 3 <= MaskWidth && odd(MaskWidth)

▷ **MaskHeight** (input_control) . . . . . integer ⇝ *integer*
Height of the correlation window.
Default: 11
Suggested values: MaskHeight ∈ {5, 7, 9, 11, 21}
Restriction: 3 <= MaskHeight && odd(MaskHeight)

▷ **TextureThresh** (input_control) . . . . . real ⇝ *real* / integer
Variance threshold of textured image regions.
Default: 0.0
Suggested values: TextureThresh ∈ {0.0, 10.0, 30.0}
Restriction: 0.0 <= TextureThresh

▷ **MinDisparity** (input_control) . . . . . integer ⇝ *integer*
Minimum of the expected disparities.
Default: -30
Value range: -32768 ≤ MinDisparity ≤ 32767

▷ **MaxDisparity** (input_control) . . . . . integer ⇝ *integer*
Maximum of the expected disparities.
Default: 30
Value range: -32768 ≤ MaxDisparity ≤ 32767

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer*
Number of pyramid levels.
Default: 1
Suggested values: NumLevels ∈ {1, 2, 3, 4}
Restriction: 1 <= NumLevels

▷ **ScoreThresh** (input_control) . . . . . real ⇝ *real* / integer
Threshold of the correlation function.
Default: 0.5
Suggested values: ScoreThresh ∈ {-1.0, 0.0, 0.3, 0.5, 0.7}

▷ **Filter** (input_control) . . . . . string(-array) ⇝ *string*
Downstream filters.
Default: 'none'
List of values: Filter ∈ {'none', 'left_right_check'}

▷ **SubDisparity** (input_control) . . . . . string ⇝ *string*
Subpixel interpolation of disparities.
Default: 'none'
List of values: SubDisparity ∈ {'none', 'interpolation'}

Example

```
* Set internal and external stereo parameters.
* Note that, typically, these values are the result of a prior
* calibration.
gen_cam_par_area_scan_division (0.01, -665, 5.2e-006, 5.2e-006, \
                                622, 517, 1280, 1024, CamParam1)
gen_cam_par_area_scan_division (0.01, -731, 5.2e-006, 5.2e-006, \
                                654, 519, 1280, 1024, CamParam2)
create_pose (0.1535,-0.0037,0.0447,0.17,319.84,359.89, \
             'Rp+T', 'gba', 'point', RelPose)

* Compute the mapping for rectified images.
gen_binocular_rectification_map (Map1, Map2, CamParam1, CamParam2, RelPose, \
                                 1, 'viewing_direction', 'bilinear', \
                                 CamParamRect1,CamParamRect2, \
                                 Cam1PoseRect1, Cam2PoseRect2,RelPoseRect)

* Compute the disparities in online images.
while (1)
  grab_image_async (Image1, AcqHandle1, -1)
  map_image (Image1, Map1, ImageRect1)
```

```
  grab_image_async (Image2, AcqHandle2, -1)
  map_image (Image2, Map2, ImageRect2)

  binocular_disparity(ImageRect1, ImageRect2, Disparity, Score, 'sad', \
                      11, 11, 20, -40, 20, 2, 25, 'left_right_check', \
                      'interpolation')
endwhile
```

Result

`binocular_disparity` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`map_image`

Possible Successors

`threshold`, `disparity_to_distance`, `disparity_image_to_xyz`

Alternatives

`binocular_disparity_mg`, `binocular_disparity_ms`, `binocular_distance`, `binocular_distance_mg`, `binocular_distance_ms`

See also

`map_image`, `gen_binocular_rectification_map`, `binocular_calibration`

Module

3D Metrology

binocular_disparity_mg (ImageRect1, ImageRect2 : Disparity, Score : GrayConstancy, GradientConstancy, Smoothness, InitialGuess, CalculateScore, MGParamName, MGParamValue :)

Compute the disparities of a rectified stereo image pair using multigrid methods.

`binocular_disparity_mg` calculates the disparity between two rectified stereo images `ImageRect1` and `ImageRect2` and returns it in `Disparity`. In contrast to `binocular_disparity`, a variational approach based on multigrid methods is used. This approach returns disparity values also for image parts that contain no texture. In contrast to `binocular_distance_mg`, the results are not transformed into distance values.

The input images must be a pair of rectified stereo images, i.e., corresponding points must have the same vertical coordinate. The images can have different widths, but must have the same height. The runtime of the operator is approximately linear in the size of the images.

The disparity is the amount by which each point in the first image `ImageRect1` needs to be moved to reach its corresponding point in the second image `ImageRect2`. Two points are called corresponding if they are the image of the same point in the original scene. The calculated disparity field is dense and estimates the disparity also for points that do not have a corresponding point. The disparity is calculated only for those lines that are part of the domains of both input images. More exactly, the domain of the disparity map is calculated as the intersection of heights of the smallest enclosing rectangles of the domains of the input images.

The calculated disparity field is usually not perfect. If the parameter `CalculateScore` is set to *'true'*, a quality measure for the disparity is estimated for each pixel and returned in `Score`, which is a gray value image with a range from 0 to 10, where 0 is the best quality and 10 the worst. For this, the reverse disparity field from the second to the first image is calculated and compared to the returned disparity field. Because of this, the runtime roughly doubles when computing the score.

The operator uses a variational approach, where an energy value is assigned to each possible disparity field. Disparity fields with a lower energy are better than those with a high energy. The operator calculates a disparity field with the minimum energy and returns it.

The energy assigned to a disparity field consists of a data term and a smoothness term. The data term models the fact that corresponding points are images of the same part of the scene and thus have equal gray values. The smoothness term models the fact that the imaged scene and with it its disparity field is piecewise smooth, which leads to an interpolation of the disparity into areas with low information from the data term, e.g., areas with no texture.

The details of the assumptions are as follows:

Constancy of the gray values: It is assumed that corresponding points have the same gray value, i.e., that $I_1(x,y) = I_2(x+u(x,y),y)$.

Constancy of the gray value gradients: It is assumed that corresponding points have the same gray value gradient, i.e., that $\nabla I_1(x,y) = \nabla I_2(x+u(x,y),y))$. Discrepancies from this assumption are modeled using the L_2 norm of the difference of the two gradients. The gray value gradient has the advantage of being invariant to additive illumination changes between the two images.

Statistical robustness in the data term: To reduce the influence of outliers, i.e., points that violate the constancy assumptions, they are penalized in a statistically robust manner via the total variation $\Psi(x) = \sqrt{x+\epsilon^2}$, where $\epsilon = 0.01$ is a fixed regularization constant.

Smoothness of the disparity field: It is assumed that the resulting disparity field is piecewise smooth. This is modeled by the L_2 norm of the derivative of the disparity field.

Statistical robustness in the smoothness term: Analogously to the data term, the statistically robust total variation is applied to the smoothness term to reduce the influence of outliers. This is especially important for preserving edges in the disparity field that appear on object boundaries.

The energy functional is the integral of a linear combination of the above terms over the area of the first image. The coefficients of the linear combination are parameters of the operator and allow a fine tuning of the model to a specific situation. `GrayConstancy` determines the influence of the gray value constancy, `GradientConstancy` the influence of the constancy of the gray value gradient, and `Smoothness` the influence of the smoothness term. The first two parameters need to be adapted to the gray value interval of the images. The proposed parameters are valid for images with a gray value range of 0 to 255.

Let $I_1(x,y)$ be the gray value of the first image at the coordinates (x,y), $I_2(x,y)$ the gray value of the second image, and $u(x,y)$ the value of the disparity at the coordinate (x,y). Then, the energy functional is then given by

$$\begin{aligned} E = \int \Psi \quad (\quad & \texttt{GrayConstancy} \cdot \underbrace{(I_2(x+u(x,y),y) - I_1(x,y))^2}_{\text{gray value constancy}} + \\ & \texttt{GradientConstancy} \cdot \underbrace{|\nabla I_2(x+u(x,y)) - \nabla I_1(x,y)|^2)}_{\text{gradient constancy}} \\ + \quad & \texttt{Smoothness} \cdot \Psi \underbrace{(|\nabla u(x,y)|^2)}_{\text{smoothness}} \, dx\, dy \end{aligned}$$

It is assumed that the disparity field u that minimizes the functional E satisfies the above assumptions and is thus a good approximation of the disparity between the two images.

The above functional is minimized by finding the roots of the Euler-Lagrange equation (ELE) of the integral. This is comparable to finding the extremal values of a one-dimensional function by searching the roots of its derivative. The ELE is a nonlinear partial differential equation over the region of the integral, which needs to be 0 for extrema of E. Since the functional typically does not have any maxima, the corresponding roots of the ELE correspond to the minima of the functional.

The following techniques are used to find the roots of the ELE:

Fixed point iteration: The ELE is solved by converting it to a fixed point iteration that iteratively approaches the solution. The number of iterations can be used to balance between speed and accuracy of the solution. Each step of the fixed point iteration consists of solving a linear partial differential equation.

Coarse-to-fine process: A Gaussian image pyramid of the stereo images is created. The ELE is first solved on a coarse level of the pyramid and the solution is taken as the initial value of the fixed point iteration of the next level. This has a number of advantages and disadvantages:

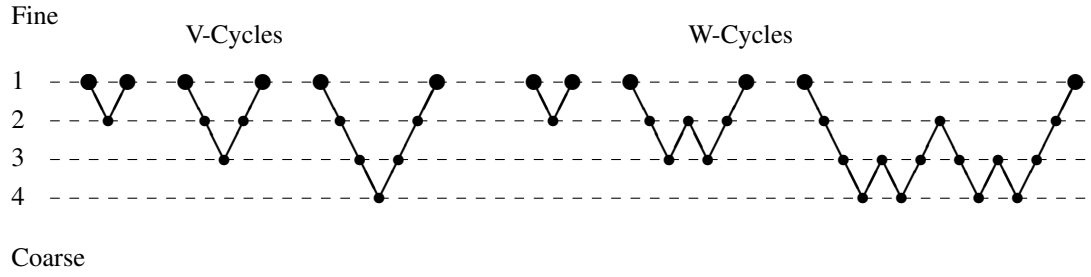
1. Since the fixed point iteration of the next level receives a good initial value, fewer iterations are necessary to archive a good accuracy. The iteration must perform only small corrections of the disparity.

2. Large disparities on the original images become small disparities on the coarse grid levels and can thus be calculated more easily.

3. The robustness against noise in the images is increased because most kinds of noise disappear on the coarse version of the images.

4. Problems arise with small structures that have a large disparity difference to their surroundings since they disappear on coarse versions of the image and thus the disparity of the surroundings is calculated. This error will not be corrected on the finer levels of the image pyramid since only small corrections are calculated there.

Multigrid methods: The linear partial differential equations that arise in the fixed point iteration at each pyramid level are converted into a linear system of equations through linearization. These linear systems are solved using iterative solvers. Multigrid methods are among the most efficient solvers for the kind of linear systems that arise here. They use the fact that classic iterative solvers, like the Gauss-Seidel solver, quickly reduce the high frequency parts of the error, but only slowly reduce the low frequency parts. Multigrid methods thus calculate the error on a coarser grid where the low frequency part of the error appears as high frequencies and can be reduced quickly by the classical solvers. This is done hierarchically, i.e., the computation of the error on a coarser resolution level itself uses the same strategy and efficiently computes its error (i.e., the error of the error) by correction steps on an even coarser resolution level. Depending on whether one or two error correction steps are performed per cycle, a so called V or W cycle is obtained. The corresponding strategies for stepping through the resolution hierarchy are as follows for two to four resolution levels:



Here, iterations on the original problem are denoted by large markers, while small markers denote iterations on error correction problems.

Algorithmically, a correction cycle can be described as follows:

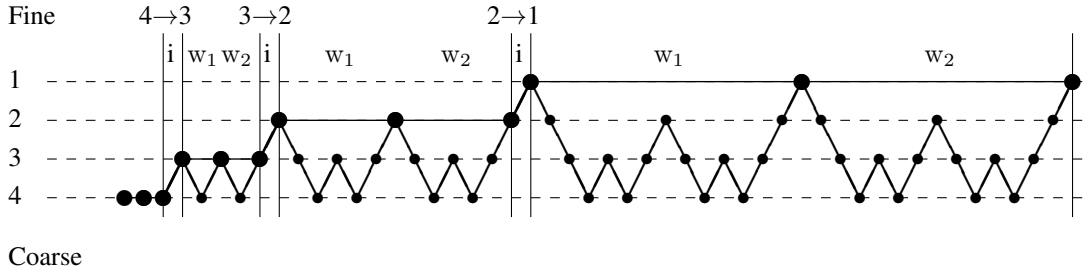
1. In the first step, several (few) iterations using an interactive linear or nonlinear basic solver are performed (e.g., a variant of the Gauss-Seidel solver). This step is called pre-relaxation step.

2. In the second step, the current error is computed to correct the current solution (the solution after step 1). For efficiency reasons, the error is calculated on a coarser resolution level. This step, which can be performed iteratively several times, is called coarse grid correction step.

3. In a final step, again several (few) iterations using the interactive linear or nonlinear basic solver of step 1 are performed. This step is called post-relaxation step.

In addition, the solution can be initialized in a hierarchical manner. Starting from a very coarse variant of the original linear equation system, the solution is successively refined. To do so, interpolated solutions of coarser variants of the equation system are used as the initialization of the next finer variant. On each resolution level itself, the V or W cycles described above are used to efficiently solve the linear equation system on that resolution level. The corresponding multigrid methods are called full multigrid methods in the literature. The full multigrid algorithm can be visualized as follows:

Full multigrid algorithm



This example represents a full multigrid algorithm that uses two W correction cycles per resolution level of the hierarchical initialization. The interpolation steps of the solution from one resolution level to the next are denoted by i and the two W correction cycles by w_1 and w_2. Iterations on the original problem are denoted by large markers, while small markers denote iterations on error correction problems.

Depending on the selected multigrid solver, a number of parameters for fine tuning the solver are available and are described in the following.

The parameter `InitialGuess` gives a initial value for the initialization of the fixed point iteration on the coarsest grid. Usually 0 is sufficient, but to avoid local minima other values can be used.

Using the parameters `MGParamName` and `MGParamValue`, the solver is controlled, i.e., the coarse-to-fine process, the fixed point iteration, and the multigrid solver. It is usually sufficient to use one of the predefined parameter sets, which are available by setting `MGParamName` = *'default_parameters'* and `MGParamValue` = *'very_accurate'*, *'accurate'*, *'fast_accurate'*, or *'fast'*.

If the parameters should be specified individually, `MGParamName` and `MGParamValue` must be set to tuples of the same length. The values corresponding to the parameters specified in `MGParamName` must be specified at the corresponding position in `MGParamValue`. The parameters are evaluated in the given order. Therefore, it is possible to first select a group of default parameters (see above) and then change only some of the parameters. in the following, the possible parameters are described.

`MGParamName` = *'mg_solver'* sets the solver for the linear system. Possible values for `MGParamValue` are *'multigrid'* for a simple multigrid solver, *'full_multigrid'* for a full multigrid solver, and *'gauss_seidel'* for the plain Gauss-Seidel solver. The multigrid methods have the advantage of a faster convergence, but incur the overhead of coarsening the linear system.

`MGParamName` = *'mg_cycle_type'* selects the type of recursion for the multigrid solvers. Possible values for `MGParamValue` are *'v'* for a V-Cycle, *'w'* for a W-Cycle, and *'none'* for no recursion.

`MGParamName` = *'mg_pre_relax'* sets the number of iterations of the pre-relaxation step in multigrid solvers, or the number of iterations for the Gauss-Seidel solver, depending on which is selected.

`MGParamName` = *'mg_post_relax'* sets the number of iterations of the post-relaxation step.

Increasing the number of pre- and post-relaxation steps increases the computation time asymptotically linearly. However, no additional restriction and prolongation operations (zooming down and up of the error correction images) are performed. Consequently, a moderate increase in the number of relaxation steps only leads to a slight increase in the computation times.

`MGParamName` = *'initial_level'* sets the coarsest level of the image pyramid where the coarse-to-fine process starts. The value can be positive, in which case it directly gives the initial level. Level 0 is the finest level with the original images. If the value is negative, then it is used relative to the maximum number of pyramid levels. The coarsest available pyramid level is the one where both images have a size of at least 4 pixels in both directions. As described below, the default value of *'initial_level'* is *-2*. This facilitates the calculation of the correct disparity for images that have very large disparities. In some cases, e.g., for repeating textures, this may lead to the fact that too large disparities are calculated for some parts of the image. In this case, *'initial_level'* should be set to a smaller value.

The standard parameters zoom the image with a factor of 0.6 per pyramid level. If a guess of the maximum disparity d exists, then the initial level s should be selected so that 0.6^{-s} is greater than d.

`MGParamName` = *'iterations'* sets the number of iterations of the fixed point iteration per pyramid level. The exact number of iterations is $steps = \min(10, \text{iterations} + \text{level}^2)$, where level is the current level in the image pyramid. If this value is set to 0, then no iteration is performed on the finest pyramid level 0. Instead, the result of

level 1 is scaled to the original image size and returned, which can be used if speed is crucial. The runtime of the operator is approximately linear in the number of iterations.

MGParamName = *'pyramid_factor'* determines the factor by which the images are scaled when creating the image pyramid for the coarse-to-fine process. The width and height of the next smaller image is scaled by the given factor. The value must lie between 0.1 and 0.9.

The predefined parameter sets for MGParamName = *'default_parameters'* contain the following values:

'default_parameters' = *'very_accurate'*: *'mg_solver'* = *'full_multigrid'*, *'mg_cycle_type'* = *'w'*, *'mg_pre_relax'* = 5, *'mg_post_relax'* = 5, *'initial_level'* = -2, *'iterations'* = 5, *'pyramid_factor'* = 0.6.

'default_parameters' = *'accurate'*: *'mg_solver'* = *'full_multigrid'*, *'mg_cycle_type'* = *'w'*, *'mg_pre_relax'* = 5, *'mg_post_relax'* = 5, *'initial_level'* = -2, *'iterations'* = 2, *'pyramid_factor'* = 0.6.

'default_parameters' = *'fast_accurate'*: *'mg_solver'* = *'full_multigrid'*, *'mg_cycle_type'* = *'v'*, *'mg_pre_relax'* = 2, *'mg_post_relax'* = 2, *'initial_level'* = -2, *'iterations'* = 1, *'pyramid_factor'* = 0.6. These are the default parameters of the algorithm if the default parameter set is not specified.

'default_parameters' = *'fast'*: *'mg_solver'* = *'full_multigrid'*, *'mg_cycle_type'* = *'v'*, *'mg_pre_relax'* = 1, *'mg_post_relax'* = 1, *'initial_level'* = -2, *'iterations'* = 0, *'pyramid_factor'* = 0.6.

Weaknesses of the operator: Large jumps in the disparity, which correspond to large jumps in the distance of the observed objects, are smoothed rather strongly. This leads to problems with thin objects that have a large distance to their background.

Distortions can occur at the left and right border of the image in the parts that are visible in only one of the images.

Additionally, general problems of stereo vision should be avoided, including horizontally repetitive patterns, areas with little texture as well as reflections.

Parameters

▷ **ImageRect1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Rectified image of camera 1.

▷ **ImageRect2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Rectified image of camera 2.

▷ **Disparity** (output_object) ..... singlechannelimage(-array) ⇝ *object :* real
Disparity map.

▷ **Score** (output_object) ..... singlechannelimage(-array) ⇝ *object :* real
Score of the calculated disparity if CalculateScore is set to *'true'*.

▷ **GrayConstancy** (input_control) ..... real ⇝ *real*
Weight of the gray value constancy in the data term.
Default: 1.0
Suggested values: GrayConstancy ∈ {0.0, 1.0, 2.0, 10.0}
Restriction: GrayConstancy >= 0.0

▷ **GradientConstancy** (input_control) ..... real ⇝ *real*
Weight of the gradient constancy in the data term.
Default: 30.0
Suggested values: GradientConstancy ∈ {0.0, 1.0, 5.0, 10.0, 30.0, 50.0, 70.0}
Restriction: GradientConstancy >= 0.0

▷ **Smoothness** (input_control) ..... real ⇝ *real*
Weight of the smoothness term in relation to the data term.
Default: 5.0
Suggested values: Smoothness ∈ {1.0, 3.0, 5.0, 10.0}
Restriction: Smoothness > 0.0

▷ **InitialGuess** (input_control) ..... real ⇝ *real*
Initial guess of the disparity.
Default: 0.0
Suggested values: InitialGuess ∈ {-30.0, -20.0, -10.0, 0.0, 10.0, 20.0, 30.0}

▷ **CalculateScore** (input_control) ..... string ⇝ *string*
Should the quality measure should be returned in Score?
Default: 'false'
Suggested values: CalculateScore ∈ {'true', 'false'}

▷ **MGParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Parameter name(s) for the multigrid algorithm.
Default: 'default_parameters'
List of values: MGParamName ∈ {'default_parameters', 'mg_solver', 'mg_cycle_type', 'mg_pre_relax', 'mg_post_relax', 'initial_level', 'pyramid_factor', 'iterations'}

▷ **MGParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer
Parameter value(s) for the multigrid algorithm.
Default: 'fast_accurate'
Suggested values: MGParamValue ∈ {'very_accurate', 'accurate', 'fast_accurate', 'fast', 'v', 'w', 'none', 'gauss_seidel', 'multigrid', 'full_multigrid', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, -1, -2, -3, -4, -5}

Example

```
read_image (BaseballL, 'stereo/epipolar/baseball_l')
read_image (BaseballR, 'stereo/epipolar/baseball_r')
binocular_disparity_mg (BaseballL, BaseballR, Disparity, Score, \
                        0.25, 30, 5, 0, 'true', \
                        'default_parameters','fast_accurate')
```

Result

If the parameter values are correct, binocular_disparity_mg returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

map_image

Possible Successors

threshold, disparity_to_distance, disparity_image_to_xyz

Alternatives

binocular_disparity, binocular_disparity_ms, binocular_distance, binocular_distance_mg, binocular_distance_ms

See also

map_image, gen_binocular_rectification_map, binocular_calibration

Module

3D Metrology

binocular_disparity_ms (ImageRect1, ImageRect2 : Disparity, Score : MinDisparity, MaxDisparity, SurfaceSmoothing, EdgeSmoothing, GenParamName, GenParamValue :)

Compute the disparities of a rectified stereo image pair using multi-scanline optimization.

binocular_disparity_ms calculates the disparity between two rectified stereo images ImageRect1 and ImageRect2 using multi-scanline optimization. The resulting disparity image is returned in Disparity. In contrast to binocular_distance_ms, the results are not transformed into distance values.

For this task, the three operators binocular_disparity, binocular_disparity_mg, and binocular_disparity_ms can be used. binocular_disparity returns robust results in regions of

sufficient texture but fails where there is none. binocular_disparity_mg interpolates low-texture regions but blurs discontinuities. binocular_disparity_ms preserves discontinuities and interpolates partially.

binocular_disparity_ms requires a reference image ImageRect1 and a search image ImageRect2 which both must be rectified, i.e., corresponding pixels must have the same row coordinate. If this assumption is violated, the images can be rectified by using the operators calibrate_cameras, gen_binocular_rectification_map, and map_image.

ImageRect1 and ImageRect2 can have different widths but must have the same height. Given a pixel in ImageRect1, the homologous pixel in ImageRect2 is selected by searching along the corresponding row in ImageRect2 and matching both pixels based on a similarity measure. The disparity is the number of pixels by which each pixel in ImageRect1 needs to be moved to reach the homologous pixel in ImageRect2.

The search space is confined by the minimum and maximum disparity values MinDisparity and MaxDisparity. If the minimum and maximum disparity values are set to an empty tuple, they are automatically set to the maximum possible range for the given images ImageRect1.

To calculate the disparities from the similarity measure, the intermediate results are optimized by a multi-scanline method. The optimization increases the robustness in low-texture areas without blurring discontinuities in the disparity image. The optimization is controlled by the parameters SurfaceSmoothing and EdgeSmoothing. SurfaceSmoothing controls the smoothness within surfaces. High values suppress disparity differences of one pixel. EdgeSmoothing controls the occurrence and the shape of edges. Low values allow many edges, high values lead to fewer and rounder edges. For both parameters, reasonable values usually range between 0 and 100. If both parameters are set to zero, no optimization is performed.

The calculation of the disparities can be controlled by generic parameters. The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'consistency_check' Activates an optional post-processing step to increase robustness. Concurrent direct and reverse matches between reference patterns in ImageRect1 and ImageRect2 are required for a disparity value to be returned. The check is switched off by setting GenParamValue to *'false'*.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'disparity_offset' Adapts the quality of the coarse-to-fine approach at discontinuities. The higher the value set in GenParamValue, the more runtime is required.

Suggested values: *2*, *3*, *4*.

Default: *3*.

'method': Determines the method used to calculate the disparities. The following parameters GenParamValue can be set:

- *'accurate'*: Most accurate calculation method, but requires more runtime and memory compared to the remaining methods.
- *'fast'*: Uses a coarse-to-fine scheme to improve the runtime. The coarse-to-fine scheme works in a similar way to the scheme explained in binocular_disparity.
  The coarse-to-fine method requires significantly less memory and is significantly faster than the *'accurate'* method, especially for large images or a large range of MinDisparity and MaxDisparity. The coarse-to-fine scheme has the further advantage that it automatically estimates the range of MinDisparity and MaxDisparity while traversing through the pyramid. As a consequence, neither MinDisparity nor MaxDisparity needs to be set. However, the generated disparity images are less accurate for the *'fast'* method than for the default *'accurate'* approach. Especially at sharp disparity jumps the *'fast'* method preserves discontinuities less accurately.
- *'very_fast'*: Also uses a coarse-to-fine scheme to improve the runtime even further. However, this approach makes numerous assumptions that may lead to a smoothing of the disparities at discontinuities. Per default, the number of levels of the coarse-to-fine scheme is estimated automatically. It is possible to set the number of levels explicitly (see *'num_levels'*).

The runtime of the operator is approximately linear to the image width, the image height, and the disparity range. Consequently, the disparity range should be chosen as narrow as possible for large images. The runtime of the coarse-to-fine scheme (which is used for *'fast'* or *'very_fast'*) is approximately linear to the image width and the image height. For small images and small disparity ranges the runtime of the coarse-to-fine scheme may be larger than that of the *'accurate'* scheme.

List of values: *'accurate'*, *'fast'*, *'very_fast'*.

Default: *'accurate'*.

'num_levels': Determines the number of pyramid levels that are used for the coarse-to-fine scheme. By setting `GenParamValue` to *'auto'*, the number of pyramid levels is automatically calculated.

Suggested values: *2*, *3*, *'auto'*.

Default: *'auto'*.

'similarity_measure': Sets the similarity measure to be used. For both options *'census_dense'* (default) and *'census_sparse'*, the similarity measure is based on the Census transform. A Census transformed image contains for every pixel information about the intensity topology within a support window around it.

- *'census_dense'*: Uses a dense 9 x 7 pixels window and is more suitable for fine structures.
- *'census_sparse'*: Uses a sparse 15 x 15 pixels window where only a subset of the pixels is evaluated. Is more robust in low-texture areas.

List of values: *'census_dense'*, *'census_sparse'*.

Default: *'census_dense'*.

'sub_disparity': Activates sub-pixel refinement of disparities when set to *'true'*. Can be deactivated by setting *'false'*.

List of values: *'true'*, *'false'*.

Default: *'true'*.

The resulting disparity is returned in the single-channel image `Disparity`. A quality measure for each disparity value is returned in `Score`, containing the best (lowest) result of the optimized similarity measure of a reference pixel.

Parameters

▷ **`ImageRect1`** (input_object) ..... singlechannelimage ⇝ *object :* byte
Rectified image of camera 1.

▷ **`ImageRect2`** (input_object) ..... singlechannelimage ⇝ *object :* byte
Rectified image of camera 2.

▷ **`Disparity`** (output_object) ..... singlechannelimage ⇝ *object :* real
Disparity map.

▷ **`Score`** (output_object) ..... singlechannelimage ⇝ *object :* real
Score of the calculated disparity.

▷ **`MinDisparity`** (input_control) ..... integer ⇝ *integer*
Minimum of the expected disparities.
Default: -30
Value range: -32768 ≤ `MinDisparity` ≤ 32768
Restriction: MinDisparity <= MaxDisparity

▷ **`MaxDisparity`** (input_control) ..... integer ⇝ *integer*
Maximum of the expected disparities.
Default: 30
Value range: -32768 ≤ `MaxDisparity` ≤ 32768
Restriction: MinDisparity <= MaxDisparity

▷ **`SurfaceSmoothing`** (input_control) ..... integer ⇝ *integer*
Smoothing of surfaces.
Default: 50
Suggested values: `SurfaceSmoothing` ∈ {20, 50, 100}
Restriction: SurfaceSmoothing >= 0

▷ **`EdgeSmoothing`** (input_control) ..... integer ⇝ *integer*
Smoothing of edges.
Default: 50
Suggested values: `EdgeSmoothing` ∈ {20, 50, 100}
Restriction: EdgeSmoothing >= 0

▷ **`GenParamName`** (input_control) ..... attribute.name(-array) ⇝ *string*
Parameter name(s) for the multi-scanline algorithm.
Default: []
List of values: `GenParamName` ∈ {'method', 'similarity_measure', 'consistency_check', 'sub_disparity', 'num_levels', 'disparity_offset'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *string*
Parameter value(s) for the multi-scanline algorithm.
Default: []
Suggested values: GenParamValue ∈ {'accurate', 'fast', 'very_fast', 'census_dense', 'census_sparse', 'true', 'false', 'auto'}

Example

```
read_image (BaseballL, 'stereo/epipolar/baseball_l')
read_image (BaseballR, 'stereo/epipolar/baseball_r')
binocular_disparity_ms (BaseballL, BaseballR, Disparity, Score, \
                        -40, -10, 50, 50, [], [])
```

Result

If the parameter values are correct, binocular_disparity_ms returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

map_image

Possible Successors

threshold, disparity_to_distance, disparity_image_to_xyz

Alternatives

binocular_disparity, binocular_disparity_mg, binocular_distance, binocular_distance_mg, binocular_distance_ms

See also

map_image, gen_binocular_rectification_map, binocular_calibration

Module

3D Metrology

binocular_distance (ImageRect1, ImageRect2 : Distance, Score : CamParamRect1, CamParamRect2, RelPoseRect, Method, MaskWidth, MaskHeight, TextureThresh, MinDisparity, MaxDisparity, NumLevels, ScoreThresh, Filter, SubDistance :)

Compute the distance values for a rectified stereo image pair using correlation techniques.

binocular_distance computes the distance values for a rectified stereo image pair using correlation techniques. The operator first calculates the disparities between the two images ImageRect1 and ImageRect2 similar to binocular_disparity. The resulting disparities are transformed into distance values of the corresponding 3D world points to the rectified stereo camera system as in disparity_to_distance. The distances are returned in the single-channel image Distance in which each gray value represents the distance of the respective 3D world point to the stereo camera system.

The algorithm requires a reference image ImageRect1 and a search image ImageRect2 which must be rectified, i.e., corresponding epipolar lines are parallel and lie on identical image rows ($r_1 = r_2$). In case this assumption is violated the images can be rectified by using the operators calibrate_cameras, gen_binocular_rectification_map and map_image. Hence, given a pixel in the reference image ImageRect1 the homologous pixel in ImageRect2 is selected by searching along the corresponding row

in ImageRect2 and matching a local neighborhood within a rectangular window of size MaskWidth and MaskHeight. For each defined reference pixel the pixel correspondences are transformed into distances of the world points defined by the intersection of the lines of sight of a corresponding pixel pair to the $z = 0$ plane of the rectified stereo system.

For this transformation the rectified internal camera parameters CamParamRect1 of camera 1 and CamParamRect2 of camera 2, and the pose with the external parameters RelPoseRect have to be defined. Latter one is of the form ${}^{ccsR1}\mathbf{P}_{ccsR2}$ and characterizes the relative pose of both cameras to each other. More precisely, it specifies the point transformation from the rectified camera system 2 (*ccsR2*) into the rectified camera system 1 (*ccsR1*), see Transformations / Poses and "Solution Guide III-C - 3D Vision". These parameters can be obtained from the operator calibrate_cameras and gen_binocular_rectification_map. After all, a quality measure for each distance value is returned in Score, containing the best result of the matching function S of a reference pixel. For the matching, the gray values of the original unprocessed images are used.

- *'sad'*: Summed Absolute Differences
  $S(r,c,d) = \frac{1}{N} \sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} | g_1(r',c') - g_2(r',c'+d) |,$
  with $0 \leq S(r,c,d) \leq 255$.
- *'ssd'*: Summed Squared Differences
  $S(r,c,d) = \frac{1}{N} \sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} (g_1(r',c') - g_2(r',c'+d))^2,$
  with $0 \leq S(r,c,d) \leq 65025$.
- *'ncc'*: Normalized Cross Correlation
  $S(r,c,d) = \frac{\sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} (g_1(r',c') - \bar{g_1}(r,c))(g_2(r',c'+d) - \bar{g_2}(r,c+d))}{\sqrt{\left(\sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} (g_1(r',c') - \bar{g_1}(r,c))^2\right)\left(\sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} (g_2(r',c'+d) - \bar{g_2}(r,c+d))^2\right)}},$
  with $-1.0 \leq S(r,c,d) \leq 1.0$.

with

$r1$, $c1$, $r2$, $c2$: row and column coordinates of the corresponding pixels of the two input images,

$g1$, $g2$: gray values of the unprocessed input images,

$N = (2m+1)(2n+1)$: size of correlation window

$\bar{g}(r,c) = \frac{1}{N} \sum_{r'=r-m}^{r+m} \sum_{c'=c-n}^{c+n} g(r',c')$: mean value within the correlation window of width $2m+1$ and height $2n+1$.

Note that the methods *'sad'* and *'ssd'* compare the gray values of the pixels within a mask window directly, whereas *'ncc'* compensates for the mean gray value and its variance within the mask window. Therefore, if the two images differ in brightness and contrast, this method should be preferred. For images with similar brightness and contrast *'sad'* and *'ssd'* are to be preferred as they are faster because of less complex internal computations. See binocular_disparity for further details.

It should be noted that the quality of correlation for rising S is falling in methods *'sad'* and *'ssd'* (the best quality value is 0) but rising in method *'ncc'* (the best quality value is 1.0).

The size of the correlation window ($2m+1$ and $2n+1$) has to be odd numbered and is passed in MaskWidth and MaskHeight. The search space is confined by the minimum and maximum disparity value MinDisparity and MaxDisparity. Due to pixel values not defined beyond the image border the resulting domain of Distance and Score is generally not set along the image border within a margin of height MaskHeight/2 at the top and bottom border and of width MaskWidth/2 at the left and right border. For the same reason, the maximum disparity range is reduced at the left and right image border.

Since matching turns out to be highly unreliable when dealing with poorly textured areas, the minimum variance within the correlation window can be defined in TextureThresh. This threshold is applied on both input images ImageRect1 and ImageRect2. In addition, ScoreThresh guarantees the matching quality and defines the maximum (*'sad'*,*'ssd'*) or, respectively, minimum (*'ncc'*) score value of the correlation function. Setting Filter to *'left_right_check'*, moreover, increases the robustness of the returned matches, as the result relies on a concurrent direct and reverse match, whereas *'none'* switches it off.

The number of pyramid levels used to improve the time response of binocular_distance is determined by NumLevels. Following a coarse-to-fine scheme disparity images of higher levels are computed and segmented into rectangular subimages to reduce the disparity range on the next lower pyramid level. TextureThresh and ScoreThresh are applied on every level and the returned domain of the Distance and Score images arises from the intersection of the resulting domains of every single level. Generally, pyramid structures are the more advantageous the more the distance image can be segmented into regions of homogeneous distance values and the bigger the disparity range must be specified. As a drawback, coarse pyramid levels might loose important texture information which can result in deficient distance values.

Finally, the value *'interpolation'* for parameter SubDistance increases the refinement and accuracy of the distance values. It is switched off by setting the parameter to *'none'*.

Attention

If using cameras with telecentric lenses, the Distance is not defined as the distance of a point to the camera but as the distance from the point to the plane, defined by the y-axes of both cameras and their baseline (see gen_binocular_rectification_map).

For a stereo setup of mixed type (i.e., for a stereo setup in which one of the original cameras is a perspective camera and the other camera is a telecentric camera; see gen_binocular_rectification_map), the rectifying plane of the two cameras is in a position with respect to the object that would lead to very unintuitive distances. Therefore, binocular_distance does not support a stereo setup of mixed type. For stereo setups of mixed type, please use reconstruct_surface_stereo, in which the reference coordinate system can be chosen arbitrarily. Alternatively, binocular_disparity and disparity_image_to_xyz might be used.

Additionally, stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported.

Parameters

- ▷ **ImageRect1** (input_object) ..... singlechannelimage ⇝ *object :* byte
  Rectified image of camera 1.
- ▷ **ImageRect2** (input_object) ..... singlechannelimage ⇝ *object :* byte
  Rectified image of camera 2.
- ▷ **Distance** (output_object) ..... singlechannelimage ⇝ *object :* real
  Distance image.
- ▷ **Score** (output_object) ..... singlechannelimage ⇝ *object :* real
  Evaluation of a distance value.
- ▷ **CamParamRect1** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameters of the rectified camera 1.
- ▷ **CamParamRect2** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameters of the rectified camera 2.
- ▷ **RelPoseRect** (input_control) ..... pose ⇝ *real* / integer
  Point transformation from the rectified camera 2 to the rectified camera 1.
  Number of elements: 7
- ▷ **Method** (input_control) ..... string ⇝ *string*
  Matching function.
  Default: 'ncc'
  List of values: Method ∈ {'sad', 'ssd', 'ncc'}
- ▷ **MaskWidth** (input_control) ..... integer ⇝ *integer*
  Width of the correlation window.
  Default: 11
  Suggested values: MaskWidth ∈ {5, 7, 9, 11, 21}
  Restriction: 3 <= MaskWidth && odd(MaskWidth)
- ▷ **MaskHeight** (input_control) ..... integer ⇝ *integer*
  Height of the correlation window.
  Default: 11
  Suggested values: MaskHeight ∈ {5, 7, 9, 11, 21}
  Restriction: 3 <= MaskHeight && odd(MaskHeight)
- ▷ **TextureThresh** (input_control) ..... real ⇝ *real* / integer
  Variance threshold of textured image regions.
  Default: 0.0
  Suggested values: TextureThresh ∈ {0.0, 2.0, 5.0, 10.0}
  Restriction: 0.0 <= TextureThresh

▷ **MinDisparity** (input_control) ..... integer ⇝ *integer*
Minimum of the expected disparities.
Default: 0
Value range: -32768 ≤ MinDisparity ≤ 32767

▷ **MaxDisparity** (input_control) ..... integer ⇝ *integer*
Maximum of the expected disparities.
Default: 30
Value range: -32768 ≤ MaxDisparity ≤ 32767

▷ **NumLevels** (input_control) ..... integer ⇝ *integer*
Number of pyramid levels.
Default: 1
Suggested values: NumLevels ∈ {1, 2, 3, 4}
Restriction: 1 <= NumLevels

▷ **ScoreThresh** (input_control) ..... real ⇝ *real* / integer
Threshold of the correlation function.
Default: 0.0
Suggested values: ScoreThresh ∈ {0.0, 2.0, 5.0, 10.0}

▷ **Filter** (input_control) ..... string(-array) ⇝ *string*
Downstream filters.
Default: 'none'
List of values: Filter ∈ {'none', 'left_right_check'}

▷ **SubDistance** (input_control) ..... string(-array) ⇝ *string*
Distance interpolation.
Default: 'none'
List of values: SubDistance ∈ {'none', 'interpolation'}

Example

```
* Set internal and external stereo parameters.
* Note that, typically, these values are the result of a prior
* calibration.
gen_cam_par_area_scan_division (0.01, -665, 5.2e-006, 5.2e-006, \
                                622, 517, 1280, 1024, CamParam1)
gen_cam_par_area_scan_division (0.01, -731, 5.2e-006, 5.2e-006, \
                                654, 519, 1280, 1024, CamParam2)
create_pose (0.1535,-0.0037,0.0447,0.17,319.84,359.89, \
             'Rp+T', 'gba', 'point', RelPose)
* Compute the mapping for rectified images.
gen_binocular_rectification_map (Map1, Map2, CamParam1, CamParam2, \
                                 RelPose, 1, 'viewing_direction', \
                                 'bilinear', CamParamRect1, CamParamRect2, \
                                 Cam1PoseRect1, Cam2PoseRect2, RelPoseRect)
* Compute the distance values in online images.
while (1)
  grab_image_async (Image1, AcqHandle1, -1)
  map_image (Image1, Map1, ImageRect1)

  grab_image_async (Image2, AcqHandle2, -1)
  map_image (Image2, Map2, ImageRect2)

  binocular_distance (ImageRect1, ImageRect2, Distance, Score, \
                      CamParamRect1, CamParamRect2, RelPoseRect, 'sad', \
                      11, 11, 20, -40, 20, 2, 25, \
                      'left_right_check', 'interpolation')
endwhile
```

Result

binocular_disparity returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

map_image

Possible Successors

threshold

Alternatives

binocular_distance_mg, binocular_distance_ms, binocular_disparity, binocular_disparity_mg, binocular_disparity_ms

See also

map_image, gen_binocular_rectification_map, binocular_calibration, distance_to_disparity, disparity_to_distance, disparity_image_to_xyz

Module

3D Metrology

binocular_distance_mg (ImageRect1, ImageRect2 : Distance, Score : CamParamRect1, CamParamRect2, RelPoseRect, GrayConstancy, GradientConstancy, Smoothness, InitialGuess, CalculateScore, MGParamName, MGParamValue :)

Compute the distance values for a rectified stereo image pair using multigrid methods.

binocular_distance_mg computes the distance values for a rectified stereo image pair using multigrid methods. The operator first calculates the disparities between two rectified images ImageRect1 and ImageRect2 similar to binocular_disparity_mg. The resulting disparity values are then transformed into distance values of the corresponding 3D world points to the rectified stereo camera system as in disparity_to_distance. The distances are returned in the single-channel image Distance in which each gray value represents the distance of the respective 3D world point to the stereo camera system. Different from binocular_distance this operator uses a variational approach based on multigrid methods. This approach returns distance values also for image parts that contain no texture.

The input images ImageRect1 and ImageRect2 must be a pair of rectified stereo images, i.e., corresponding points must have the same row coordinate. In case this assumption is violated the images can be rectified by using the operators calibrate_cameras, gen_binocular_rectification_map and map_image.

For the transformation of the disparity to the distance, the internal camera parameters of the rectified camera 1 CamParamRect1 and of the rectified camera 2 CamParamRect2, as well as the relative pose of the cameras RelPoseRect must be specified. The relative pose defines a point transformation from the rectified camera system 2 to the rectified camera system 1. These parameters can be obtained from the operators calibrate_cameras and gen_binocular_rectification_map.

A detailed description of the algorithm and of the remaining parameters can be found in the documentation of binocular_disparity_mg.

Attention

If using cameras with telecentric lenses, the Distance is not defined as the distance of a point to the camera but as the distance from the point to the plane, defined by the y-axes of both cameras and their baseline (see gen_binocular_rectification_map).

For a stereo setup of mixed type (i.e., for a stereo setup in which one of the original cameras is a perspective camera and the other camera is a telecentric camera; see gen_binocular_rectification_map), the rectifying plane of the two cameras is in a position with respect to the object that would lead to very unintuitive distances. Therefore, binocular_distance_mg does not support a stereo setup of mixed type. For stereo setups of mixed type, please use reconstruct_surface_stereo, in which the reference coordinate system can be chosen arbitrarily. Alternatively, binocular_disparity_mg and disparity_image_to_xyz might be used.

Additionally, stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported.

Parameters

- ▷ **ImageRect1** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Rectified image of camera 1.
- ▷ **ImageRect2** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Rectified image of camera 2.
- ▷ **Distance** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
  Distance image.
- ▷ **Score** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
  Score of the calculated disparity if `CalculateScore` is set to *'true'*.
- ▷ **CamParamRect1** (input_control) . . . . . campar ⇝ *real* / integer / string
  Internal camera parameters of the rectified camera 1.
- ▷ **CamParamRect2** (input_control) . . . . . campar ⇝ *real* / integer / string
  Internal camera parameters of the rectified camera 2.
- ▷ **RelPoseRect** (input_control) . . . . . pose ⇝ *real* / integer
  Point transformation from the rectified camera 2 to the rectified camera 1.
  Number of elements: 7
- ▷ **GrayConstancy** (input_control) . . . . . real ⇝ *real*
  Weight of the gray value constancy in the data term.
  Default: 1.0
  Suggested values: `GrayConstancy` ∈ {0.0, 1.0, 2.0, 10.0}
  Restriction: GrayConstancy >= 0.0
- ▷ **GradientConstancy** (input_control) . . . . . real ⇝ *real*
  Weight of the gradient constancy in the data term.
  Default: 30.0
  Suggested values: `GradientConstancy` ∈ {0.0, 1.0, 5.0, 10.0, 30.0, 50.0, 70.0}
  Restriction: GradientConstancy >= 0.0
- ▷ **Smoothness** (input_control) . . . . . real ⇝ *real*
  Weight of the smoothness term in relation to the data term.
  Default: 5.0
  Suggested values: `Smoothness` ∈ {1.0, 3.0, 5.0, 10.0}
  Restriction: Smoothness > 0.0
- ▷ **InitialGuess** (input_control) . . . . . real ⇝ *real*
  Initial guess of the disparity.
  Default: 0.0
  Suggested values: `InitialGuess` ∈ {-30.0, -20.0, -10.0, 0.0, 10.0, 20.0, 30.0}
- ▷ **CalculateScore** (input_control) . . . . . string ⇝ *string*
  Should the quality measure be returned in `Score`?
  Default: 'false'
  Suggested values: `CalculateScore` ∈ {'true', 'false'}
- ▷ **MGParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
  Parameter name(s) for the multigrid algorithm.
  Default: 'default_parameters'
  List of values: `MGParamName` ∈ {'default_parameters', 'mg_solver', 'mg_cycle_type', 'mg_pre_relax', 'mg_post_relax', 'initial_level', 'pyramid_factor', 'iterations'}
- ▷ **MGParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer
  Parameter value(s) for the multigrid algorithm.
  Default: 'fast_accurate'
  Suggested values: `MGParamValue` ∈ {'very_accurate', 'accurate', 'fast_accurate', 'fast', 'v', 'w', 'none', 'gauss_seidel', 'multigrid', 'full_multigrid', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, -1, -2, -3, -4, -5}

Result

If the parameter values are correct, `binocular_distance_mg` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input images are available) the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

map_image

Possible Successors

threshold

Alternatives

binocular_distance, binocular_distance_ms, binocular_disparity, binocular_disparity_mg, binocular_disparity_ms

See also

map_image, gen_binocular_rectification_map, binocular_calibration, disparity_to_distance, distance_to_disparity, disparity_image_to_xyz

Module

3D Metrology

binocular_distance_ms (ImageRect1, ImageRect2 : Distance, Score : CamParamRect1, CamParamRect2, RelPoseRect, MinDisparity, MaxDisparity, SurfaceSmoothing, EdgeSmoothing, GenParamName, GenParamValue :)

Compute the distance values for a rectified stereo image pair using multi-scanline optimization.

binocular_distance_ms computes the distance values for a rectified stereo image pair using multi-scanline optimization. The operator first calculates the disparities between two rectified images ImageRect1 and ImageRect2 similar to binocular_disparity_ms. The resulting disparity values are then transformed into distance values of the corresponding 3D world points to the rectified stereo camera system as in disparity_to_distance. The distances are returned in the single-channel image Distance in which each gray value represents the distance of the respective 3D world point to the stereo camera system.

binocular_disparity_ms requires a reference image ImageRect1 and a search image ImageRect2 which both must be rectified, i.e., corresponding pixels must have the same row coordinate. If this assumption is violated, the images can be rectified by using the operators calibrate_cameras, gen_binocular_rectification_map, and map_image.

For the transformation of the disparity to the distance, the internal camera parameters of the rectified camera 1 CamParamRect1 and of the rectified camera 2 CamParamRect2, as well as the relative pose of the cameras RelPoseRect must be specified. The relative pose defines a point transformation from the rectified camera system 2 to the rectified camera system 1. These parameters can be obtained from the operators calibrate_cameras and gen_binocular_rectification_map.

A detailed description of the remaining parameters can be found in the documentation of binocular_disparity_ms.

Attention

If using cameras with telecentric lenses, the Distance is not defined as the distance of a point to the camera but as the distance from the point to the plane, defined by the y-axes of both cameras and their baseline (see gen_binocular_rectification_map).

For a stereo setup of mixed type (i.e., for a stereo setup in which one of the original cameras is a perspective camera and the other camera is a telecentric camera; see gen_binocular_rectification_map), the rectifying plane of the two cameras is in a position with respect to the object that would lead to very unintuitive distances. Therefore, binocular_distance_ms does not support a stereo setup of mixed type. For stereo setups of mixed type, please use reconstruct_surface_stereo, in which the reference coordinate system can be

chosen arbitrarily. Alternatively, binocular_disparity_ms and disparity_image_to_xyz might be used.

Additionally, stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported.

Parameters

- ▷ **ImageRect1** (input_object) . . . . . singlechannelimage ⇝ *object :* byte
  Rectified image of camera 1.
- ▷ **ImageRect2** (input_object) . . . . . singlechannelimage ⇝ *object :* byte
  Rectified image of camera 2.
- ▷ **Distance** (output_object) . . . . . singlechannelimage ⇝ *object :* real
  Distance image.
- ▷ **Score** (output_object) . . . . . singlechannelimage ⇝ *object :* real
  Score of the calculated disparity.
- ▷ **CamParamRect1** (input_control) . . . . . campar ⇝ *real* / integer / string
  Internal camera parameters of the rectified camera 1.
- ▷ **CamParamRect2** (input_control) . . . . . campar ⇝ *real* / integer / string
  Internal camera parameters of the rectified camera 2.
- ▷ **RelPoseRect** (input_control) . . . . . pose ⇝ *real* / integer
  Point transformation from the rectified camera 2 to the rectified camera 1.
  Number of elements: 7
- ▷ **MinDisparity** (input_control) . . . . . integer ⇝ *integer*
  Minimum of the expected disparities.
  Default: -30
  Value range: -32768 ≤ MinDisparity ≤ 32768
  Restriction: MinDisparity <= MaxDisparity
- ▷ **MaxDisparity** (input_control) . . . . . integer ⇝ *integer*
  Maximum of the expected disparities.
  Default: 30
  Value range: -32768 ≤ MaxDisparity ≤ 32768
  Restriction: MinDisparity <= MaxDisparity
- ▷ **SurfaceSmoothing** (input_control) . . . . . integer ⇝ *integer*
  Smoothing of surfaces.
  Default: 50
  Suggested values: SurfaceSmoothing ∈ {20, 50, 100}
  Restriction: SurfaceSmoothing >= 0
- ▷ **EdgeSmoothing** (input_control) . . . . . integer ⇝ *integer*
  Smoothing of edges.
  Default: 50
  Suggested values: EdgeSmoothing ∈ {20, 50, 100}
  Restriction: EdgeSmoothing >= 0
- ▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
  Parameter name(s) for the multi-scanline algorithm.
  Default: []
  List of values: GenParamName ∈ {'similarity_measure', 'disparity_offset', 'num_levels', 'consistency_check', 'sub_disparity'}
- ▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer
  Parameter value(s) for the multi-scanline algorithm.
  Default: []
  Suggested values: GenParamValue ∈ {'census_dense', 'census_sparse', 'true', 'false'}

Result

If the parameter values are correct, binocular_distance_ms returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

map_image

Possible Successors

threshold

Alternatives

binocular_distance, binocular_distance_mg, binocular_disparity, binocular_disparity_mg, binocular_disparity_ms

See also

map_image, gen_binocular_rectification_map, binocular_calibration, disparity_to_distance, distance_to_disparity, disparity_image_to_xyz

Module

3D Metrology

disparity_image_to_xyz (Disparity : X, Y, Z : CamParamRect1, CamParamRect2, RelPoseRect :)

Transform a disparity image into 3D points in a rectified stereo system.

Given the disparity image Disparity of a rectified binocular stereo system, disparity_image_to_xyz computes the corresponding 3D points. Their coordinates relative to the rectified camera 1 are stored as gray values in the images X, Y, and Z, i.e., the pixels at the position (Row,Column) in X, Y, and Z contain the x, y, and z coordinate, respectively, of the pixel (Row,Column) in the disparity image.

The rectified binocular camera system is specified by its internal camera parameters CamParamRect1 of the rectified camera 1 and CamParamRect2 of the rectified camera 2, and the external parameters RelPoseRect. The latter one is a pose in the form ${}^{ccsR1}\mathbf{P}_{ccsR2}$, thus it defines the relative pose of the rectified camera coordinate system 2 (*ccsR2*) relative to the rectified camera coordinate system 1 (*ccsR1*) (see Transformations / Poses and "Solution Guide III-C - 3D Vision"). These camera parameters can be obtained from the operators calibrate_cameras and gen_binocular_rectification_map.

Attention

Stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported.

Parameters

- ▷ **Disparity** (input_object) ..... singlechannelimage(-array) ⇝ *object :* real
  Disparity image.
- ▷ **X** (output_object) ..... singlechannelimage(-array) ⇝ *object :* real
  X coordinates of the points in the rectified camera system 1.
- ▷ **Y** (output_object) ..... singlechannelimage(-array) ⇝ *object :* real
  Y coordinates of the points in the rectified camera system 1.
- ▷ **Z** (output_object) ..... singlechannelimage(-array) ⇝ *object :* real
  Z coordinates of the points in the rectified camera system 1.
- ▷ **CamParamRect1** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameters of the rectified camera 1.
- ▷ **CamParamRect2** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameters of the rectified camera 2.
- ▷ **RelPoseRect** (input_control) ..... pose ⇝ *real* / integer
  Pose of the rectified camera 2 in relation to the rectified camera 1.
  Number of elements: 7

Example

```
disparity_image_to_xyz (ImageDisparity, ImgX, ImgY, ImgZ, RectCamParL, \
                        RectCamParR, RectLPosRectR)
get_region_points (ImageDisparity, Rows, Columns)
get_grayval (ImgX, Rows, Columns, XValues)
get_grayval (ImgY, Rows, Columns, YValues)
get_grayval (ImgZ, Rows, Columns, ZValues)
```

Result

The operator `disparity_image_to_xyz` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input image available) is set via the operator `set_system ('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

`binocular_disparity`

Possible Successors

`threshold`, `write_image`

Alternatives

`disparity_to_point_3d`, `binocular_distance`

See also

`binocular_calibration`, `gen_binocular_rectification_map`, `intersect_lines_of_sight`

Module

3D Metrology

disparity_to_distance (: : CamParamRect1, CamParamRect2, RelPoseRect, Disparity : Distance)

Transform a disparity value into a distance value in a rectified binocular stereo system.

`disparity_to_distance` transforms a disparity value into a distance of an object point to the binocular stereo system. The cameras of this system must be rectified and are defined by the rectified internal parameters `CamParamRect1` of camera 1 and `CamParamRect2` of camera 2, and the external parameters `RelPoseRect`. Latter specifies the relative pose of both cameras to each other by defining a point transformation from rectified camera system 2 to rectified camera system 1. These parameters can be obtained from the operator `calibrate_cameras` and `gen_binocular_rectification_map`. The disparity value `Disparity` is defined by the column difference of the image coordinates of two corresponding points on an epipolar line according to the equation $d = c_2 - c_1$(see also `binocular_disparity`). This value characterises a set of 3D object points of an equal distance to a plane being parallel to the rectified image plane of the stereo system. The distance to the subset plane $z = 0$ which is parallel to the rectified image plane and contains the optical centers of both cameras is returned in `Distance`.

Attention

If using cameras with telecentric lenses, the `Distance` is not defined as the distance of a point to the camera but as the distance from the point to the plane, defined by the y-axes of both cameras and their baseline (see `gen_binocular_rectification_map`).

For a stereo setup of mixed type (i.e., for a stereo setup in which one of the original cameras is a perspective camera and the other camera is a telecentric camera; see `gen_binocular_rectification_map`), the rectifying

plane of the two cameras is in a position with respect to the object that would lead to very unintuitive distances. Therefore, disparity_to_distance does not support stereo setups of mixed type. For stereo setups of mixed type, disparity_to_point_3d should be used instead.

Additionally, stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported.

Parameters

- ▷ **CamParamRect1** (input_control) . . . . . campar ⇝ *real* / integer / string
  Rectified internal camera parameters of camera 1.
- ▷ **CamParamRect2** (input_control) . . . . . campar ⇝ *real* / integer / string
  Rectified internal camera parameters of camera 2.
- ▷ **RelPoseRect** (input_control) . . . . . pose ⇝ *real* / integer
  Point transformation from the rectified camera 2 to the rectified camera 1.
  Number of elements: 7
- ▷ **Disparity** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Disparity between the images of the world point.
- ▷ **Distance** (output_control) . . . . . real(-array) ⇝ *real*
  Distance of a world point to the rectified camera system.

Result

disparity_to_distance returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

binocular_calibration, gen_binocular_rectification_map, map_image, binocular_disparity

Alternatives

binocular_distance

See also

distance_to_disparity, disparity_to_point_3d

Module

3D Metrology

disparity_to_point_3d (: : CamParamRect1, CamParamRect2, RelPoseRect, Row1, Col1, Disparity : X, Y, Z)

Transform an image point and its disparity into a 3D point in a rectified stereo system.

Given an image point of the rectified camera 1, specified by its image coordinates (Row1,Col1), and its disparity in a rectified binocular stereo system, disparity_to_point_3d computes the corresponding three dimensional object point. The disparity value Disparity defines the column difference of the image coordinates of two corresponding features on an epipolar line according to the equation $d = c_2 - c_1$. The rectified binocular camera system is specified by its internal camera parameters CamParamRect1 of camera 1 and CamParamRect2 of camera 2, and the external parameters RelPoseRect defining the pose of the rectified camera 2 in relation to the rectified camera 1. These camera parameters can be obtained from the operators calibrate_cameras and gen_binocular_rectification_map. The 3D point is returned in Cartesian coordinates (X,Y,Z) of the rectified camera system 1.

Attention

Stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported.

Parameters

- ▷ **CamParamRect1** (input_control) . . . . . campar ⇝ *real* / integer / string
  Rectified internal camera parameters of camera 1.
- ▷ **CamParamRect2** (input_control) . . . . . campar ⇝ *real* / integer / string
  Rectified internal camera parameters of camera 2.
- ▷ **RelPoseRect** (input_control) . . . . . pose ⇝ *real* / integer
  Pose of the rectified camera 2 in relation to the rectified camera 1.
  Number of elements: 7
- ▷ **Row1** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Row coordinate of a point in the rectified image 1.
- ▷ **Col1** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Column coordinate of a point in the rectified image 1.
- ▷ **Disparity** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Disparity of the images of the world point.
- ▷ **X** (output_control) . . . . . real(-array) ⇝ *real*
  X coordinate of the 3D point.
- ▷ **Y** (output_control) . . . . . real(-array) ⇝ *real*
  Y coordinate of the 3D point.
- ▷ **Z** (output_control) . . . . . real(-array) ⇝ *real*
  Z coordinate of the 3D point.

Result

`disparity_to_point_3d` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`binocular_calibration`, `gen_binocular_rectification_map`

Alternatives

`disparity_image_to_xyz`

See also

`binocular_disparity`, `binocular_distance`, `intersect_lines_of_sight`

Module

3D Metrology

distance_to_disparity (: : CamParamRect1, CamParamRect2, RelPoseRect, Distance : Disparity)

Transform a distance value into a disparity in a rectified stereo system.

`distance_to_disparity` transforms a distance of a 3D point to the binocular stereo system into a disparity value. The cameras of this system must be rectified and are defined by the rectified internal parameters `CamParamRect1` of the camera 1 and `CamParamRect2` of the camera 2 and the external parameters `RelPoseRect`. The latter specifies the relative pose of both camera systems to each other by defining a point transformation from the rectified camera system 2 to the rectified camera system 1. These parameters can be obtained from the operator `calibrate_cameras` and `gen_binocular_rectification_map`. The distance value is passed in `Distance` and the resulting disparity value `Disparity` is defined by the column difference of the image coordinates of two corresponding features on an epipolar line according to the equation $d = c_2 - c_1$.

Attention

If using cameras with telecentric lenses, the `Distance` is not defined as the distance of a point to the camera

but as the distance from the point to the plane, defined by the y-axes of both cameras and their baseline (see gen_binocular_rectification_map).

For stereo setups of mixed type (i.e., for a stereo setup in which one of the original cameras is a perspective camera and the other camera is a telecentric camera; see gen_binocular_rectification_map), the rectifying plane of the two cameras is in a position with respect to the object that would lead to very unintuitive distances. Therefore, distance_to_disparity does not support stereo setups of mixed type.

Additionally, stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported.

Parameters

- ▷ **CamParamRect1** (input_control) . . . . . campar ⇝ *real* / integer / string
  Rectified internal camera parameters of camera 1.
- ▷ **CamParamRect2** (input_control) . . . . . campar ⇝ *real* / integer / string
  Rectified internal camera parameters of camera 2.
- ▷ **RelPoseRect** (input_control) . . . . . pose ⇝ *real* / integer
  Point transformation from the rectified camera 2 to the rectified camera 1.
  Number of elements: 7
- ▷ **Distance** (input_control) . . . . . real(-array) ⇝ *real*
  Distance of a world point to camera 1.
- ▷ **Disparity** (output_control) . . . . . number(-array) ⇝ *real* / integer
  Disparity between the images of the point.

Result

distance_to_disparity returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

binocular_calibration, gen_binocular_rectification_map

Possible Successors

binocular_disparity

Module

3D Metrology

essential_to_fundamental_matrix (: : EMatrix, CovEMat, CamMat1, CamMat2 : FMatrix, CovFMat)

Compute the fundamental matrix from an essential matrix.

The fundamental matrix is the entity describing the epipolar constraint in image coordinates (C,R) and the essential matrix is its counterpart for 3D direction vectors (X,Y,1):

$$\begin{pmatrix} C_2 \\ R_2 \\ 1 \end{pmatrix}^T \cdot \texttt{FMatrix} \cdot \begin{pmatrix} C_1 \\ R_1 \\ 1 \end{pmatrix} = 0 \quad \text{and} \quad \begin{pmatrix} X_2 \\ Y_2 \\ 1 \end{pmatrix}^T \cdot \texttt{EMatrix} \cdot \begin{pmatrix} X_1 \\ Y_1 \\ 1 \end{pmatrix} = 0 \quad .$$

Image coordinates result from 3D direction vectors by multiplication with the camera matrix $CamMat$:

$$\begin{pmatrix} col \\ row \\ 1 \end{pmatrix} = CamMat \cdot \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \quad .$$

Therefore, the fundamental matrix FMatrix is calculated from the essential matrix EMatrix and the camera matrices CamMat1, CamMat2 by the following formula:

$$\texttt{FMatrix} \quad = \quad \texttt{CamMat2}^{-T} \cdot \texttt{EMatrix} \cdot \texttt{CamMat1}^{-1} \quad .$$

The transformation of the essential matrix to the fundamental matrix goes along with the propagation of the covariance matrices CovEMat to CovFMat. If CovEMat is empty CovFMat will be empty too.

The conversion operator essential_to_fundamental_matrix is used especially for a subsequent visualization of the epipolar line structure via the fundamental matrix, which depicts the underlying stereo geometry.

Parameters

▷ **EMatrix** (input_control) . . . hom_mat2d ⇝ *real* / integer
Essential matrix.

▷ **CovEMat** (input_control) . . . number-array ⇝ *real* / integer
9×9 covariance matrix of the essential matrix.
Default: []

▷ **CamMat1** (input_control) . . . hom_mat2d ⇝ *real* / integer
Camera matrix of the 1. camera.

▷ **CamMat2** (input_control) . . . hom_mat2d ⇝ *real* / integer
Camera matrix of the 2. camera.

▷ **FMatrix** (output_control) . . . hom_mat2d ⇝ *real*
Computed fundamental matrix.

▷ **CovFMat** (output_control) . . . real-array ⇝ *real*
9×9 covariance matrix of the fundamental matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

vector_to_essential_matrix

Alternatives

rel_pose_to_fundamental_matrix

Module

3D Metrology

gen_binocular_proj_rectification (: Map1, Map2 : FMatrix, CovFMat, Width1, Height1, Width2, Height2, SubSampling, Mapping : CovFMatRect, H1, H2)

Compute the projective rectification of weakly calibrated binocular stereo images.

A binocular stereo setup is called weakly calibrated if the fundamental matrix, which describes the projective relation between the two images, is known. Rectification is the process of finding a suitable set of transformations, that transform both images such that all corresponding epipolar lines become collinear and parallel to the horizontal axes. The rectified images can be thought of as acquired by a stereo configuration where the left and right image plane are identical and the difference between both image centers is a horizontal translation. Note that rectification can only be performed if both of the epipoles are located outside the images.

Typically, the fundamental matrix is calculated beforehand with match_fundamental_matrix_ransac and FMatrix is the basis for the computation of the two homographies H1 and H2, which describe the rectifications for the left image and the right image respectively. Since a projective rectification is an underdetermined problem, additional constraints are defined: the algorithm chooses the set of homographies that minimizes the projective distortion induced by the homographies in both images. For the computation of this cost function the

dimensions of the images must be provided in Width1, Height1, Width2, Height2. After rectification the fundamental matrix is always of the canonical form

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} .$$

In the case of a known covariance matrix CovFMat of the fundamental matrix FMatrix, the covariance matrix CovFMatRect of the above rectified fundamental matrix is calculated. This can help for an improved stereo matching process because the covariance matrix defines in terms of probabilities the image domain where to find a corresponding match.

Similar to the operator gen_binocular_rectification_map the output images Map1 and Map2 describe the transformation, also called mapping, of the original images to the rectified ones. The parameter Mapping specifies whether bilinear interpolation (*'bilinear_map'*) should be applied between the pixels in the input image or whether the gray value of the nearest neighboring pixel should be taken (*'nn_map'*). The size and resolution of the maps and of the transformed images can be adjusted by the parameter SubSampling, which applies a sub-sampling factor to the original images. For example, a factor of two will halve the image sizes. If just the two homographies are required Mapping can be set to *'no_map'* and no maps will be returned. For speed reasons, this option should be used if for a specific stereo configuration the images must be rectified only once. If the stereo setup is fixed, the maps should be generated only once and both images should be rectified with map_image; this will result in the smallest computational cost for on-line rectification.

When using the maps, the transformed images are of the same size as their maps. Each pixel in the map contains the description of how the new pixel at this position is generated. The images Map1 and Map2 are single channel images if Mapping is set to *'nn_map'* and five channel images if it is set to *'bilinear_map'*. In the first channel, which is of type int4, the pixels contain the linear coordinates of their reference pixels in the original image. With Mapping equal to *'no_map'* this reference pixel is the nearest neighbor to the back-transformed pixel coordinates of the map. In the case of bilinear interpolation the reference pixel is the next upper left pixel relative to the back-transformed coordinates. The following scheme shows the ordering of the pixels in the original image next to the back-transformed pixel coordinates, where the reference pixel takes the number 2.

2	3
4	5

The channels 2 to 5, which are of type uint2, contain the weights of the relevant pixels for the bilinear interpolation.

Based on the rectified images, the disparity be computed using binocular_disparity. In contrast to stereo with fully calibrated cameras, using the operator gen_binocular_rectification_map and its successors, metric depth information can not be derived for weakly calibrated cameras. The disparity map gives just a qualitative depth ordering of the scene.

Parameters

▷ **Map1** (output_object) ..... image(-array) ⇝ *object :* int4 / uint2
Image coding the rectification of the 1. image.

▷ **Map2** (output_object) ..... image(-array) ⇝ *object :* int4 / uint2
Image coding the rectification of the 2. image.

▷ **FMatrix** (input_control) ..... hom_mat2d ⇝ *real* / integer
Fundamental matrix.

▷ **CovFMat** (input_control) ..... number-array ⇝ *real* / integer
9×9 covariance matrix of the fundamental matrix.
Default: []

▷ **Width1** (input_control) ..... integer ⇝ *integer*
Width of the 1. image.
Default: 512
Suggested values: Width1 ∈ {128, 256, 512, 1024}
Restriction: Width1 > 0

▷ **Height1** (input_control) ..... integer ⇝ *integer*
Height of the 1. image.
Default: 512
Suggested values: Height1 ∈ {128, 256, 512, 1024}
Restriction: Height1 > 0

▷ **Width2** (input_control) ..... integer ⇝ *integer*
Width of the 2. image.
Default: 512
Suggested values: Width2 ∈ {128, 256, 512, 1024}
Restriction: Width2 > 0

▷ **Height2** (input_control) ..... integer ⇝ *integer*
Height of the 2. image.
Default: 512
Suggested values: Height2 ∈ {128, 256, 512, 1024}
Restriction: Height2 > 0

▷ **SubSampling** (input_control) ..... number ⇝ *integer* / real
Subsampling factor.
Default: 1
List of values: SubSampling ∈ {1, 2, 3, 1.5}

▷ **Mapping** (input_control) ..... string ⇝ *string*
Type of mapping.
Default: 'no_map'
List of values: Mapping ∈ {'no_map', 'nn_map', 'bilinear_map'}

▷ **CovFMatRect** (output_control) ..... number-array ⇝ *real*
9×9 covariance matrix of the rectified fundamental matrix.

▷ **H1** (output_control) ..... hom_mat2d ⇝ *real*
Projective transformation of the 1. image.

▷ **H2** (output_control) ..... hom_mat2d ⇝ *real*
Projective transformation of the 2. image.

Example

```
* Rectify an image pair using a map.
get_image_size (Image1, Width1, Height1)
get_image_size (Image2, Width2, Height2)
points_harris (Image1, 3, 1, 0.2, 10000, Row1, Col1)
points_harris (Image2, 3, 1, 0.2, 10000, Row2, Col2)
match_fundamental_matrix_ransac (Image1, Image2, Row1, Col1, Row2, Col2, \
                                 'ncc', 21, 0, 200, 20, 50, 0, 0.9, \
                                 'gold_standard', 0.3, 1, FMatrix, \
                                 CovFMat, Error, Points1, Points2)
gen_binocular_proj_rectification (Map1, Map2, FMatrix, [], Width1, \
                                  Height1, Width2, Height2, 1, \
                                  'bilinear_map', CovFMatRect, H1, H2)
map_image (Image1, Map1, Image1Rect)
map_image (Image2, Map2, Image2Rect)

* Rectify an image pair without using a map.
get_image_size (Image1, Width1, Height1)
get_image_size (Image2, Width2, Height2)
points_harris (Image1, 3, 1, 0.2, 10000, Row1, Col1)
points_harris (Image2, 3, 1, 0.2, 10000, Row2, Col2)
match_fundamental_matrix_ransac (Image1, Image2, Row1, Col1, Row2, Col2, \
                                 'ncc', 21, 0, 200, 20, 50, 0, 0.9, \
                                 'gold_standard', 0.3, 1, FMatrix, \
                                 CovFMat, Error, Points1, Points2)
gen_binocular_proj_rectification (Map1, Map2, FMatrix, [], Width1, \
                                  Height1, Width2, Height2, 1, \
                                  'no_map', CovFMatRect, H1, H2)
* Determine the maximum extent of the two rectified images.
projective_trans_point_2d (H1, [0,0,Height1,Height1], \
                           [0,Width1,0,Width1], [1,1,1,1], R1, C1, W1)
R1 := int(floor(R1/W1))
C1 := int(floor(C1/W1))
```

```
projective_trans_point_2d (H2, [0,0,Height2,Height2], \
                           [0,Width2,0,Width2], [1,1,1,1], R2, C2, W2)
R2 := int(floor(R2/W2))
C2 := int(floor(C2/W2))
WidthRect := max([C1,C2])
HeightRect := max([R1,R2])
projective_trans_image_size (Image1, Image1Rect, H1, 'bilinear', \
                             WidthRect, HeightRect, 'false')
projective_trans_image_size (Image2, Image2Rect, H2, 'bilinear', \
                             WidthRect, HeightRect, 'false')
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

match_fundamental_matrix_ransac, vector_to_fundamental_matrix

Possible Successors

map_image, projective_trans_image, binocular_disparity

Alternatives

gen_binocular_rectification_map

References

J. Gluckmann and S.K. Nayar: “Rectifying transformations that minimize resampling effects”; IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2001, vol I, pages 111-117.

Module

3D Metrology

gen_binocular_rectification_map (: Map1, Map2 : CamParam1, CamParam2, RelPose, SubSampling, Method, MapType : CamParamRect1, CamParamRect2, CamPoseRect1, CamPoseRect2, RelPoseRect)

Generate transformation maps that describe the mapping of the images of a binocular camera pair to a common rectified image plane.

Given a pair of stereo images, rectification determines a transformation of each image plane in a way that pairs of conjugate epipolar lines become collinear and parallel to the horizontal image axes. This is required for an efficient calculation of disparities or distances with operators such as binocular_disparity or binocular_distance. The rectified images can be thought of as acquired by a new stereo rig, obtained by rotating and, in case of telecentric area scan and line scan cameras, translating the original cameras. The projection centers (i.e., in the telecentric case, the direction of the optical axes) are maintained. For perspective cameras, the image planes are additionally transformed into a common plane, which means that the focal lengths are set equal, and the optical axes are parallel. For a stereo setup of mixed type (i.e., one perspective and one telecentric camera), the image planes are also transformed into a common plane, as described below.

To achieve the transformation map for rectified images gen_binocular_rectification_map requires the internal camera parameters CamParam1 of camera 1 and CamParam2 of camera 2, as well as the relative pose RelPose, ${}^{ccs1}\mathbf{P}_{ccs2}$, defining a point transformation from camera coordinate system 2 (*ccs2*) into camera coordinate system 1 (*ccs1*), see Transformations / Poses and “Solution Guide III-C - 3D Vision”. These parameters can be obtained, e.g., from the operator calibrate_cameras.

The internal camera parameters, modified by the rectification, are returned in CamParamRect1 for camera 1 and CamParamRect2 for camera 2, respectively. The rotation and, in case of telecentric cameras, translation of the rectified camera in relation to the original camera is specified by CamPoseRect1 and CamPoseRect2, respectively. These poses are in the form ${}^{ccsX}\mathbf{P}_{ccsRX}$ with *ccsX*: camera coordinate system of camera X and *ccsRX*: camera coordinate system of camera X for the rectified image. Finally, RelPoseRect returns ${}^{ccsR1}\mathbf{P}_{ccsR2}$, the

relative pose of the rectified camera coordinate system 2 (*ccsR2*) relative to the rectified camera coordinate system 1 (*ccsR1*).

Rectification Method

For perspective area scan cameras, `RelPoseRect` only has a translation in x. Generally, the transformations are defined in a way that the rectified camera 1 is left of the rectified camera 2. This means that the optical center of camera 2 has a positive x coordinate of the rectified coordinate system of camera 1.

The projection onto a common plane has many degrees of freedom, which are implicitly restricted by selecting a certain method in `Method`:

- *'viewing_direction'* uses the baseline as the x-axis of the common image plane. The mean of the viewing directions (z-axes) of the two cameras is used to span the x-z plane of the rectified system. The resulting rectified z-axis is the orientation of the common image plane and as such located in this plane and orthogonal to the baseline. In many cases, the resulting rectified z-axis will not differ much from the mean of the two old z-axes. The new focal length is determined in such a way that the old principal points have the same distance to the new common image plane. The different z-axes directions are illustrated in the schematic below.

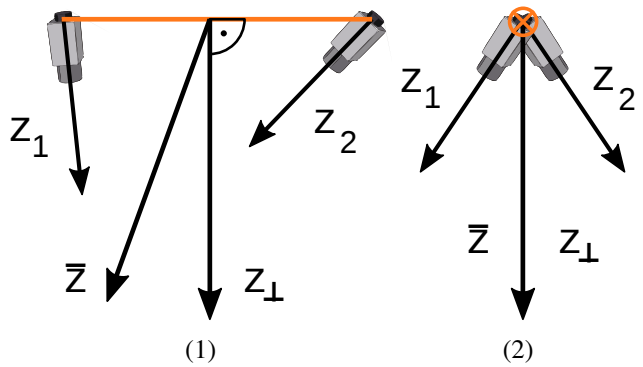


Illustration for the different z-axes directions using *'viewing_direction'*. (1): View facing the base line (in orange). (2): View along the base line (pointing into the page, in orange).

- *'geometric'* specifies the orientation of the common image plane by the cross product of the baseline and the line of intersection of the original image planes. The new focal length is determined in such a way that the old principal points have the same distance to the new common image plane.

For telecentric area scan and line scan cameras, the parameter `Method` is ignored. The relative pose of both cameras is not uniquely defined in such a system since the cameras return identical images no matter how they are translated along their optical axis. Yet, in order to define an absolute distance measurement to the cameras, a standard position of both cameras is considered. This position is defined as follows: Both cameras are translated along their optical axes until their distance is one meter and until the line between the cameras (baseline) forms the same angle with both optical axes (i.e., the baseline and the optical axes form an isosceles triangle). The optical axes remain unchanged. The relative pose of the rectified cameras `RelPoseRect` may be different from the relative pose of the original cameras `RelPose`.

For a stereo setup of mixed type (i.e., one perspective and one telecentric camera), the parameter `Method` is ignored. The rectified image plane is determined uniquely from the geometry of the perspective camera and the relative pose of the two cameras. The normal of the rectified image plane is the vector that points from the projection center of the perspective camera to the point on the optical axis of the telecentric camera that has the shortest distance from the projection center of the perspective camera. This is also the z-axis of the rectified perspective camera. The geometric base of the mixed camera system is a line that passes through the projection center of the perspective camera and has the same direction as the z-axis of the telecentric camera, i.e., the base is parallel to the viewing direction of the telecentric camera. The x-axis of the rectified perspective camera is given by the base and the y-axis is constructed to form a right-handed coordinate system. To rectify the telecentric camera, its optical axis must be shifted to the base and the image plane must be tilted by $90°$ or $-90°$. To achieve this, a special type of object-side telecentric camera that is able to handle this special rectification geometry (indicated by a negative image plane distance `ImagePlaneDist`) must be used for the rectified telecentric camera. The representation of this special camera type should be regarded as a black box because it is used only for rectification purposes in HALCON (for this reason, it is not documented in `camera_calibration`). The rectified telecentric camera has the same orientation as the original telecentric camera, while its origin is translated to a point on the base.

Rectification Maps

The mapping functions for the images of camera 1 and camera 2 are returned in the images `Map1` and `Map2`. `MapType` is used to specify the type of the output maps. If *'nearest_neighbor'* is chosen, both maps consist of one image containing one channel, in which for each pixel of the resulting image the linearized coordinate of the pixel of the input image is stored that is the nearest neighbor to the transformed coordinates. If *'bilinear'* interpolation is chosen, both maps consists of one image containing five channels. In the first channel for each pixel in the resulting image the linearized coordinates of the pixel in the input image is stored that is in the upper left position relative to the transformed coordinates. The four other channels contain the weights of the four neighboring pixels of the transformed coordinates which are used for the bilinear interpolation, in the following order:

2	3
4	5

The second channel, for example, contains the weights of the pixels that lie to the upper left relative to the transformed coordinates. If *'coord_map_sub_pix'* is chosen, both maps consist of one vector field image, in which for each pixel of the resulting image the subpixel precise coordinates in the input image are stored.

The size and resolution of the maps and of the transformed images can be adjusted by the `SubSampling` parameter which applies a sub-sampling factor to the original images.

If you want to re-use the created map in another program, you can save it as a multi-channel image with the operator `write_image`, using the format *'tiff'*.

Attention

Stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported.

Parameters

- ▷ **`Map1`** (output_object) . . . . . (multichannel-)image ⇝ *object :* int4 / uint2 / vector_field
  Image containing the mapping data of camera 1.
- ▷ **`Map2`** (output_object) . . . . . (multichannel-)image ⇝ *object :* int4 / uint2 / vector_field
  Image containing the mapping data of camera 2.
- ▷ **`CamParam1`** (input_control) . . . . . campar ⇝ *real* / integer / string
  Internal parameters of camera 1.
- ▷ **`CamParam2`** (input_control) . . . . . campar ⇝ *real* / integer / string
  Internal parameters of camera 2.
- ▷ **`RelPose`** (input_control) . . . . . pose ⇝ *real* / integer
  Point transformation from camera 2 to camera 1.
  Number of elements: 7
- ▷ **`SubSampling`** (input_control) . . . . . real ⇝ *real*
  Subsampling factor.
  Default: 1.0
  Suggested values: `SubSampling` ∈ {0.5, 0.66, 1.0, 1.5, 2.0, 3.0, 4.0}
- ▷ **`Method`** (input_control) . . . . . string ⇝ *string*
  Type of rectification.
  Default: 'viewing_direction'
  List of values: `Method` ∈ {'viewing_direction', 'geometric'}
- ▷ **`MapType`** (input_control) . . . . . string ⇝ *string*
  Type of mapping.
  Default: 'bilinear'
  List of values: `MapType` ∈ {'nearest_neighbor', 'bilinear', 'coord_map_sub_pix'}
- ▷ **`CamParamRect1`** (output_control) . . . . . campar ⇝ *real* / integer / string
  Rectified internal parameters of camera 1.
- ▷ **`CamParamRect2`** (output_control) . . . . . campar ⇝ *real* / integer / string
  Rectified internal parameters of camera 2.
- ▷ **`CamPoseRect1`** (output_control) . . . . . pose ⇝ *real* / integer
  Point transformation from the rectified camera 1 to the original camera 1.
  Number of elements: 7
- ▷ **`CamPoseRect2`** (output_control) . . . . . pose ⇝ *real* / integer
  Point transformation from the rectified camera 1 to the original camera 1.
  Number of elements: 7

▷ **RelPoseRect** (output_control) . . . . . pose ⇝ *real* / integer
Point transformation from the rectified camera 2 to the rectified camera 1.
Number of elements: 7

Example

```
* Set internal and external stereo parameters.
* Note that, typically, these values are the result of a prior
* calibration.
gen_cam_par_area_scan_division (0.01, -665, 5.2e-006, 5.2e-006, \
                                622, 517, 1280, 1024, CamParam1)
gen_cam_par_area_scan_division (0.01, -731, 5.2e-006, 5.2e-006, \
                                654, 519, 1280, 1024, CamParam2)
create_pose (0.1535,-0.0037,0.0447,0.17,319.84,359.89, \
             'Rp+T', 'gba', 'point', RelPose)

* Compute the mapping for rectified images.
gen_binocular_rectification_map (Map1, Map2, CamParam1, CamParam2, \
                                 RelPose, 1,'viewing_direction', 'bilinear',\
                                 CamParamRect1, CamParamRect2, \
                                 CamPoseRect1, CamPoseRect2, \
                                 RelPoseRect)

* Compute the disparities in online images.
while (1)
  grab_image_async (Image1, AcqHandle1, -1)
  map_image (Image1, Map1, ImageMapped1)

  grab_image_async (Image2, AcqHandle2, -1)
  map_image (Image2, Map2, ImageMapped2)

  binocular_disparity(ImageMapped1, ImageMapped2, Disparity, Score, \
                      'sad', 11, 11, 20, -40, 20, 2, 25, \
                      'left_right_check', 'interpolation')
endwhile
```

Result

`gen_binocular_rectification_map` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`binocular_calibration`

Possible Successors

`map_image`

Alternatives

`gen_image_to_world_plane_map`

See also

`map_image`, `binocular_disparity`, `binocular_distance`, `binocular_disparity_mg`, `binocular_distance_mg`, `binocular_disparity_ms`, `binocular_distance_ms`, `gen_image_to_world_plane_map`, `contour_to_world_plane_xld`, `image_points_to_world_plane`

Module

3D Metrology

intersect_lines_of_sight (: : CamParam1, CamParam2, RelPose, Row1, Col1, Row2, Col2 : X, Y, Z, Dist)

Get a 3D point from the intersection of two lines of sight within a binocular camera system.

Given two lines of sight from different cameras, specified by their image points (Row1,Col1) of camera 1 and (Row2,Col2) of camera 2, intersect_lines_of_sight computes the 3D point of intersection of these lines. The binocular camera system is specified by its internal camera parameters CamParam1 of the projective camera 1 and CamParam2 of the projective camera 2, and the external parameters RelPose. Latter one is of the form ${}^{ccs1}\mathbf{P}_{ccs2}$ and characterizes the relative pose of both cameras to each other, thus defining a point transformation from camera coordinate system 2 (*ccs2*) into camera coordinate system 1 (*ccs1*), see Transformations / Poses and "Solution Guide III-C - 3D Vision". These camera parameters can be obtained, e.g., from the operator calibrate_cameras, if the coordinates of the image points (Row1,Col1) and (Row2,Col2) refer to the respective original image coordinate system. In case of rectified image coordinates (e.g., obtained from rectified images), the rectified camera parameters must be passed, as they are returned by the operator gen_binocular_rectification_map. The 'point of intersection' is defined by the point with the shortest distance to both lines of sight. This point is returned in Cartesian coordinates (X,Y,Z) of camera system 1 and its distance to the lines of sight is passed in Dist.

Attention

Stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported.

Parameters

- ▷ **CamParam1** (input_control) ..... campar ⇝ *real* / integer / string
  Internal parameters of the projective camera 1.
- ▷ **CamParam2** (input_control) ..... campar ⇝ *real* / integer / string
  Internal parameters of the projective camera 2.
- ▷ **RelPose** (input_control) ..... pose ⇝ *real* / integer
  Point transformation from camera 2 to camera 1.
  Number of elements: 7
- ▷ **Row1** (input_control) ..... number(-array) ⇝ *real* / integer
  Row coordinate of a point in image 1.
- ▷ **Col1** (input_control) ..... number(-array) ⇝ *real* / integer
  Column coordinate of a point in image 1.
- ▷ **Row2** (input_control) ..... number(-array) ⇝ *real* / integer
  Row coordinate of the corresponding point in image 2.
- ▷ **Col2** (input_control) ..... number(-array) ⇝ *real* / integer
  Column coordinate of the corresponding point in image 2.
- ▷ **X** (output_control) ..... real(-array) ⇝ *real*
  X coordinate of the 3D point.
- ▷ **Y** (output_control) ..... real(-array) ⇝ *real*
  Y coordinate of the 3D point.
- ▷ **Z** (output_control) ..... real(-array) ⇝ *real*
  Z coordinate of the 3D point.
- ▷ **Dist** (output_control) ..... real(-array) ⇝ *real*
  Distance of the 3D point to the lines of sight.

Result

intersect_lines_of_sight returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

binocular_calibration

See also

disparity_to_point_3d

Module

3D Metrology

match_essential_matrix_ransac (Image1, Image2 : : Rows1, Cols1, Rows2, Cols2, CamMat1, CamMat2, GrayMatchMethod, MaskSize, RowMove, ColMove, RowTolerance, ColTolerance, Rotation, MatchThreshold, EstimationMethod, DistanceThreshold, RandSeed : EMatrix, CovEMat, Error, Points1, Points2)

Compute the essential matrix for a pair of stereo images by automatically finding correspondences between image points.

Given a set of coordinates of characteristic points (Rows1, Cols1) and (Rows2, Cols2) in the stereo images Image1 and Image2 along with known internal camera parameters, specified by the camera matrices CamMat1 and CamMat2, match_essential_matrix_ransac automatically determines the geometry of the stereo setup and finds the correspondences between the characteristic points. The geometry of the stereo setup is represented by the essential matrix EMatrix and all corresponding points have to fulfill the epipolar constraint.

The operator match_essential_matrix_ransac is designed to deal with a linear camera model. The internal camera parameters are passed by the arguments CamMat1 and CamMat2, which are 3×3 upper triangular matrices describing an affine transformation. The relation between a vector (X,Y,1), representing the direction from the camera to the viewed 3D space point and its (projective) 2D image coordinates (col,row,1) is:

$$\begin{pmatrix} col \\ row \\ 1 \end{pmatrix} = CamMat \cdot \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \quad \text{where} \quad CamMat = \begin{pmatrix} f/s_x & s & c_x \\ 0 & f/s_y & c_y \\ 0 & 0 & 1 \end{pmatrix} .$$

Note the column/row ordering in the point coordinates which has to be compliant with the x/y notation of the camera coordinate system. The focal length is denoted by f, s_x, s_y are scaling factors, s describes a skew factor and (c_x, c_y) indicates the principal point. Mainly, these are the elements known from the camera parameters as used for example in calibrate_cameras. Alternatively, the elements of the camera matrix can be described in a different way, see e.g. stationary_camera_self_calibration. Multiplied by the inverse of the camera matrices the direction vectors in 3D space are obtained from the (projective) image coordinates. For known camera matrices the epipolar constraint is given by:

$$\begin{pmatrix} X_2 \\ Y_2 \\ 1 \end{pmatrix}^T \cdot EMatrix \cdot \begin{pmatrix} X_1 \\ Y_1 \\ 1 \end{pmatrix} = 0 .$$

The matching process is based on characteristic points, which can be extracted with point operators like points_foerstner or points_harris. The matching itself is carried out in two steps: first, gray value correlations of mask windows around the input points in the first and the second image are determined and an initial matching between them is generated using the similarity of the windows in both images. Then, the RANSAC algorithm is applied to find the essential matrix that maximizes the number of correspondences under the epipolar constraint.

The size of the mask windows is MaskSize $\times$ MaskSize. Three metrics for the correlation can be selected. If GrayMatchMethod has the value *'ssd'*, the sum of the squared gray value differences is used, *'sad'* means the sum of absolute differences, and *'ncc'* is the normalized cross correlation. For details please refer to binocular_disparity. The metric is minimized (*'ssd'*, *'sad'*) or maximized (*'ncc'*) over all possible point pairs. A thus found matching is only accepted if the value of the metric is below the value of MatchThreshold (*'ssd'*, *'sad'*) or above that value (*'ncc'*).

To increase the speed of the algorithm, the search area for the matching operations can be limited. Only points within a window of $2 \cdot$ RowTolerance $\times 2 \cdot$ ColTolerance points are considered. The offset of the center of the search window in the second image with respect to the position of the current point in the first image is given by RowMove and ColMove.

If the second camera is rotated around the optical axis with respect to the first camera the parameter Rotation may contain an estimate for the rotation angle or an angle interval in radians. A good guess will increase the quality of the gray value matching. If the actual rotation differs too much from the specified estimate the matching will typically fail. In this case, an angle interval should be specified, and Rotation is a tuple with two elements. The larger the given interval the slower the operator is since the RANSAC algorithm is run over all angle increments within the interval.

After the initial matching is completed a randomized search algorithm (RANSAC) is used to determine the essential matrix EMatrix. It tries to find the essential matrix that is consistent with a maximum number of correspondences. For a point to be accepted, the distance to its corresponding epipolar line must not exceed the threshold DistanceThreshold.

The parameter EstimationMethod decides whether the relative orientation between the cameras is of a special type and which algorithm is to be applied for its computation. If EstimationMethod is either *'normalized_dlt'* or *'gold_standard'* the relative orientation is arbitrary. Choosing *'trans_normalized_dlt'* or *'trans_gold_standard'* means that the relative motion between the cameras is a pure translation. The typical application for this special motion case is the scenario of a single fixed camera looking onto a moving conveyor belt. In order to get a unique solution in the correspondence problem the minimum required number of corresponding points is six in the general case and three in the special, translational case.

The essential matrix is computed by a linear algorithm if *'normalized_dlt'* or *'trans_normalized_dlt'* is chosen. With *'gold_standard'* or *'trans_gold_standard'* the algorithm gives a statistically optimal result, and returns the covariance of the essential matrix CovEMat as well. Here, *'normalized_dlt'* and *'gold_standard'* stand for direct-linear-transformation and gold-standard-algorithm respectively. Note, that in general the found correspondences differ depending on the deployed estimation method.

The value Error indicates the overall quality of the estimation procedure and is the mean Euclidean distance in pixels between the points and their corresponding epipolar lines.

Point pairs consistent with the mentioned constraints are considered to be in correspondences. Points1 contains the indices of the matched input points from the first image and Points2 contains the indices of the corresponding points in the second image.

For the operator match_essential_matrix_ransac a special configuration of scene points and cameras exists: if all 3D points lie in a single plane and additionally are all closer to one of the two cameras then the solution in the essential matrix is not unique but twofold. As a consequence both solutions are computed and returned by the operator. This means that the output parameters EMatrix, CovEMat and Error are of double length and the values of the second solution are simply concatenated behind the values of the first one.

The parameter RandSeed can be used to control the randomized nature of the RANSAC algorithm, and hence to obtain reproducible results. If RandSeed is set to a positive number the operator yields the same result on every call with the same parameters because the internally used random number generator is initialized with the RandSeed. If RandSeed = *0* the random number generator is initialized with the current time. In this case the results may not be reproducible. The value set for the HALCON system variable *'seed_rand'* (see set_system) does not affect the results of match_essential_matrix_ransac.

Parameters

- ▷ **Image1** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image 1.
- ▷ **Image2** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image 2.
- ▷ **Rows1** (input_control) . . . . . number-array ⇝ *real* / integer
  Row coordinates of characteristic points in image 1.
  Restriction: length(Rows1) >= 6 || length(Rows1) >= 3
- ▷ **Cols1** (input_control) . . . . . number-array ⇝ *real* / integer
  Column coordinates of characteristic points in image 1.
  Restriction: length(Cols1) == length(Rows1)
- ▷ **Rows2** (input_control) . . . . . number-array ⇝ *real* / integer
  Row coordinates of characteristic points in image 2.
  Restriction: length(Rows2) >= 6 || length(Rows2) >= 3
- ▷ **Cols2** (input_control) . . . . . number-array ⇝ *real* / integer
  Column coordinates of characteristic points in image 2.
  Restriction: length(Cols2) == length(Rows2)

▷ **CamMat1** (input_control) . . . . . hom_mat2d ⇝ *real* / integer
Camera matrix of the 1st camera.

▷ **CamMat2** (input_control) . . . . . hom_mat2d ⇝ *real* / integer
Camera matrix of the 2nd camera.

▷ **GrayMatchMethod** (input_control) . . . . . string ⇝ *string*
Gray value comparison metric.
Default: 'ssd'
List of values: GrayMatchMethod ∈ {'ssd', 'sad', 'ncc'}

▷ **MaskSize** (input_control) . . . . . integer ⇝ *integer*
Size of gray value masks.
Default: 10
Suggested values: MaskSize ∈ {3, 7, 15}
Value range: 1 ≤ MaskSize

▷ **RowMove** (input_control) . . . . . integer ⇝ *integer*
Average row coordinate shift of corresponding points.
Default: 0
Value range: 0 ≤ RowMove ≤ 200

▷ **ColMove** (input_control) . . . . . integer ⇝ *integer*
Average column coordinate shift of corresponding points.
Default: 0
Value range: 0 ≤ ColMove ≤ 200

▷ **RowTolerance** (input_control) . . . . . integer ⇝ *integer*
Half height of matching search window.
Default: 200
Value range: 1 ≤ RowTolerance

▷ **ColTolerance** (input_control) . . . . . integer ⇝ *integer*
Half width of matching search window.
Default: 200
Value range: 1 ≤ ColTolerance

▷ **Rotation** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
Estimate of the relative orientation of the right image with respect to the left image.
Default: 0.0
Suggested values: Rotation ∈ {0.0, 0.1, -0.1, 0.7854, 1.571, 3.142}

▷ **MatchThreshold** (input_control) . . . . . number ⇝ *integer* / real
Threshold for gray value matching.
Default: 10
Suggested values: MatchThreshold ∈ {10, 20, 50, 100, 0.9, 0.7}

▷ **EstimationMethod** (input_control) . . . . . string ⇝ *string*
Algorithm for the computation of the essential matrix and for special camera orientations.
Default: 'normalized_dlt'
List of values: EstimationMethod ∈ {'normalized_dlt', 'gold_standard', 'trans_normalized_dlt', 'trans_gold_standard'}

▷ **DistanceThreshold** (input_control) . . . . . number ⇝ *real* / integer
Maximal deviation of a point from its epipolar line.
Default: 1
Value range: 0.5 ≤ DistanceThreshold ≤ 5
Restriction: DistanceThreshold > 0

▷ **RandSeed** (input_control) . . . . . integer ⇝ *integer*
Seed for the random number generator.
Default: 0

▷ **EMatrix** (output_control) . . . . . hom_mat2d ⇝ *real*
Computed essential matrix.

▷ **CovEMat** (output_control) . . . . . real-array ⇝ *real*
9×9 covariance matrix of the essential matrix.

▷ **Error** (output_control) . . . . . real(-array) ⇝ *real*
Root-Mean-Square of the epipolar distance error.

▷ **Points1** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 1.

▷ **Points2** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 2.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_foerstner, points_harris

Possible Successors

vector_to_essential_matrix

See also

match_fundamental_matrix_ransac, match_rel_pose_ransac, stationary_camera_self_calibration

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

3D Metrology

match_fundamental_matrix_distortion_ransac (Image1, Image2 : : Rows1, Cols1, Rows2, Cols2, GrayMatchMethod, MaskSize, RowMove, ColMove, RowTolerance, ColTolerance, Rotation, MatchThreshold, EstimationMethod, DistanceThreshold, RandSeed : FMatrix, Kappa, Error, Points1, Points2)

Compute the fundamental matrix and the radial distortion coefficient for a pair of stereo images by automatically finding correspondences between image points.

Given a set of coordinates of characteristic points (Rows1, Cols1) and (Rows2, Cols2) in the stereo images Image1 and Image2, which must be of identical size, match_fundamental_matrix_distortion_ransac automatically finds the correspondences between the characteristic points and determines the geometry of the stereo setup. For unknown cameras the geometry of the stereo setup is represented by the fundamental matrix FMatrix and the radial distortion coefficient Kappa (κ). All corresponding points must fulfill the epipolar constraint:

$$\begin{pmatrix} c_2 \\ r_2 \\ 1 \end{pmatrix}^T \cdot \texttt{FMatrix} \cdot \begin{pmatrix} c_1 \\ r_1 \\ 1 \end{pmatrix} = 0 \ .$$

Here, (r_1, c_1) and (r_2, c_2) denote image points that are obtained by undistorting the input image points with the division model (see Calibration):

$$r = \frac{\tilde{r}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)} \qquad c = \frac{\tilde{c}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)}$$

Here, $(\tilde{r}_1, \tilde{c}_1) = (\texttt{Rows1} - 0.5(h-1), \texttt{Cols1} - 0.5(w-1))$
and $(\tilde{r}_2, \tilde{c}_2) = (\texttt{Rows2} - 0.5(h-1), \texttt{Cols2} - 0.5(w-1))$

denote the distorted image points, specified relative to the image center, and w and h denote the width and height of the input images. Thus, match_fundamental_matrix_distortion_ransac assumes that the principal point of the camera, i.e., the center of the radial distortions, lies at the center of the image.

The returned Kappa can be used to construct camera parameters that can be used to rectify images or points (see change_radial_distortion_cam_par, change_radial_distortion_image, and change_radial_distortion_points):

$$\begin{aligned} \texttt{CamPar} \quad = \quad & [\textit{'area\_scan\_telecentric\_division'}, 0.0, \texttt{Kappa}, 1.0, 1.0, \\ & 0.5(w-1), 0.5(h-1), w, h] \end{aligned}$$

Note the column/row ordering in the point coordinates above: since the fundamental matrix encodes the projective relation between two stereo images embedded in 3D space, the x/y notation must be compliant with the camera coordinate system. Therefore, (x,y) coordinates correspond to (column,row) pairs.

The matching process is based on characteristic points, which can be extracted with point operators like points_foerstner or points_harris. The matching itself is carried out in two steps: first, gray value correlations of mask windows around the input points in the first and the second image are determined and an initial matching between them is generated using the similarity of the windows in both images. Then, the RANSAC algorithm is applied to find the fundamental matrix and radial distortion coefficient that maximizes the number of correspondences under the epipolar constraint.

The size of the mask windows used for the matching is MaskSize×MaskSize. Three metrics for the correlation can be selected. If GrayMatchMethod has the value *'ssd'*, the sum of the squared gray value differences is used, *'sad'* means the sum of absolute differences, and *'ncc'* is the normalized cross correlation. For details please refer to binocular_disparity. The metric is minimized (*'ssd'*, *'sad'*) or maximized (*'ncc'*) over all possible point pairs. A matching thus found is only accepted if the value of the metric is below the value of MatchThreshold (*'ssd'*, *'sad'*) or above that value (*'ncc'*).

To increase the speed of the algorithm the search area for the match candidates can be limited to a rectangle by specifying its size and offset. Only points within a window of $2 \cdot$ RowTolerance $\times\, 2 \cdot$ ColTolerance points are considered. The offset of the center of the search window in the second image with respect to the position of the current point in the first image is given by RowMove and ColMove.

If the second camera is rotated around the optical axis with respect to the first camera, the parameter Rotation may contain an estimate for the rotation angle or an angle interval in radians. A good guess will increase the quality of the gray value matching. If the actual rotation differs too much from the specified estimate, the matching will typically fail. In this case, an angle interval should be specified and Rotation is a tuple with two elements. The larger the given interval is the slower is the operator is since the RANSAC algorithm is run over all (automatically determined) angle increments within the interval.

After the initial matching has been completed, a randomized search algorithm (RANSAC) is used to determine the fundamental matrix FMatrix and the radial distortion coefficient Kappa. It tries to find the parameters that are consistent with a maximum number of correspondences. For a point to be accepted, the distance in pixels to its corresponding epipolar line must not exceed the threshold DistanceThreshold.

The parameter EstimationMethod decides whether the relative orientation between the cameras is of a special type and which algorithm is to be applied for its computation. If EstimationMethod is either *'linear'* or *'gold_standard'*, the relative orientation is arbitrary. If the left and right cameras are identical and the relative orientation between them is a pure translation, EstimationMethod can be set to *'trans_linear'* or *'trans_gold_standard'*. The typical application for this special motion case is the scenario of a single fixed camera looking onto a moving conveyor belt. In order to get a unique solution for the correspondence problem, the minimum required number of corresponding points is nine in the general case and four in the special translational case.

The fundamental matrix is computed by a linear algorithm if EstimationMethod is set to *'linear'* or *'trans_linear'*. This algorithm is very fast. For the pure translation case (EstimationMethod = *'trans_linear'*), the linear method returns accurate results for small to moderate noise of the point coordinates and for most distortions (except for very small distortions). For a general relative orientation of the two cameras (EstimationMethod = *'linear'*), the linear method only returns accurate results for very small noise of the point coordinates and for sufficiently large distortions. For EstimationMethod = *'gold_standard'* or *'trans_gold_standard'*, a mathematically optimal but slower optimization is used, which minimizes the geometric reprojection error of reconstructed projective 3D points. For a general relative orientation of the two cameras, in general EstimationMethod = *'gold_standard'* should be selected.

The value `Error` indicates the overall quality of the estimation procedure and is the mean symmetric Euclidean distance in pixels between the points and their corresponding epipolar lines.

Point pairs consistent with the above constraints are considered to be corresponding points. `Points1` contains the indices of the matched input points from the first image and `Points2` contains the indices of the corresponding points in the second image.

The parameter `RandSeed` can be used to control the randomized nature of the RANSAC algorithm, and hence to obtain reproducible results. If `RandSeed` is set to a positive number, the operator returns the same result on every call with the same parameters because the internally used random number generator is initialized with `RandSeed`. If `RandSeed` = *0*, the random number generator is initialized with the current time. In this case the results may not be reproducible. The value set for the HALCON system variable *'seed_rand'* (see `set_system`) does not affect the results of `match_fundamental_matrix_distortion_ransac`.

Parameters

- ▷ **`Image1`** (input_object) . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image 1.
- ▷ **`Image2`** (input_object) . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image 2.
- ▷ **`Rows1`** (input_control) . . . point.y-array ⇝ *real* / integer
  Input points in image 1 (row coordinate).
  Restriction: length(Rows1) >= 9 || length(Rows1) >= 4
- ▷ **`Cols1`** (input_control) . . . point.x-array ⇝ *real* / integer
  Input points in image 1 (column coordinate).
  Restriction: length(Cols1) == length(Rows1)
- ▷ **`Rows2`** (input_control) . . . point.y-array ⇝ *real* / integer
  Input points in image 2 (row coordinate).
  Restriction: length(Rows2) >= 9 || length(Rows2) >= 4
- ▷ **`Cols2`** (input_control) . . . point.x-array ⇝ *real* / integer
  Input points in image 2 (column coordinate).
  Restriction: length(Cols2) == length(Rows2)
- ▷ **`GrayMatchMethod`** (input_control) . . . string ⇝ *string*
  Gray value match metric.
  Default: 'ncc'
  List of values: `GrayMatchMethod` ∈ {'ncc', 'ssd', 'sad'}
- ▷ **`MaskSize`** (input_control) . . . integer ⇝ *integer*
  Size of gray value masks.
  Default: 10
  Suggested values: `MaskSize` ∈ {3, 7, 15}
  Value range: 1 ≤ MaskSize
- ▷ **`RowMove`** (input_control) . . . integer ⇝ *integer*
  Average row coordinate offset of corresponding points.
  Default: 0
- ▷ **`ColMove`** (input_control) . . . integer ⇝ *integer*
  Average column coordinate offset of corresponding points.
  Default: 0
- ▷ **`RowTolerance`** (input_control) . . . integer ⇝ *integer*
  Half height of matching search window.
  Default: 200
  Restriction: RowTolerance >= 1
- ▷ **`ColTolerance`** (input_control) . . . integer ⇝ *integer*
  Half width of matching search window.
  Default: 200
  Restriction: ColTolerance >= 1
- ▷ **`Rotation`** (input_control) . . . angle.rad(-array) ⇝ *real* / integer
  Estimate of the relative rotation of the second image with respect to the first image.
  Default: 0.0
  Suggested values: `Rotation` ∈ {0.0, 0.1, -0.1, 0.7854, 1.571, 3.142}

▷ **MatchThreshold** (input_control) . . . . . number ⇝ *integer / real*
Threshold for gray value matching.
Default: 0.7
Suggested values: MatchThreshold ∈ {0.9, 0.7, 0.5, 10, 20, 50, 100}

▷ **EstimationMethod** (input_control) . . . . . string ⇝ *string*
Algorithm for the computation of the fundamental matrix and for special camera orientations.
Default: 'gold_standard'
List of values: EstimationMethod ∈ {'linear', 'gold_standard', 'trans_linear', 'trans_gold_standard'}

▷ **DistanceThreshold** (input_control) . . . . . number ⇝ *real / integer*
Maximal deviation of a point from its epipolar line.
Default: 1
Restriction: DistanceThreshold > 0

▷ **RandSeed** (input_control) . . . . . integer ⇝ *integer*
Seed for the random number generator.
Default: 0

▷ **FMatrix** (output_control) . . . . . hom_mat2d ⇝ *real*
Computed fundamental matrix.

▷ **Kappa** (output_control) . . . . . real ⇝ *real*
Computed radial distortion coefficient.

▷ **Error** (output_control) . . . . . real ⇝ *real*
Root-Mean-Square epipolar distance error.

▷ **Points1** (output_control) . . . . . integer-array ⇝ *integer*
Indices of matched input points in image 1.

▷ **Points2** (output_control) . . . . . integer-array ⇝ *integer*
Indices of matched input points in image 2.

Example

```
points_foerstner (Image1, 1, 2, 3, 200, 0.1, 'gauss', 'true', \
                  Rows1, Cols1, _, _, _, _, _, _, _, _)
points_foerstner (Image2, 1, 2, 3, 200, 0.1, 'gauss', 'true', \
                  Rows2, Cols2, _, _, _, _, _, _, _, _)
match_fundamental_matrix_distortion_ransac (Image1, Image2, \
                                            Rows1, Cols1, Rows2, \
                                            Cols2, 'ncc', 10, 0, 0, \
                                            100, 200, 0, 0.5, \
                                            'trans_gold_standard', \
                                            1, 42, FMatrix, Kappa, \
                                            Error, Points1, Points2)
get_image_size (Image1, Width, Height)
CamParDist := ['area_scan_division',0.0,Kappa,1.0,1.0,\
               0.5*(Width-1),0.5*Height-1,Width,Height]
change_radial_distortion_cam_par ('fixed', CamParDist, 0, CamPar)
change_radial_distortion_image (Image1, Image1, Image1Rect, \
                                CamParDist, CamPar)
change_radial_distortion_image (Image2, Image2, Image2Rect, \
                                CamParDist, CamPar)
gen_binocular_proj_rectification (Map1, Map2, FMatrix, [], Width, \
                                  Height, Width, Height, 1, \
                                  'bilinear_map', _, H1, H2)
map_image (Image1Rect, Map1, Image1Mapped)
map_image (Image2Rect, Map2, Image2Mapped)
binocular_disparity_mg (Image1Mapped, Image2Mapped, Disparity, \
                        Score, 1, 30, 8, 0, 'false', \
                        'default_parameters', 'fast_accurate')
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_foerstner, points_harris

Possible Successors

vector_to_fundamental_matrix_distortion, change_radial_distortion_cam_par, change_radial_distortion_image, change_radial_distortion_points, gen_binocular_proj_rectification

See also

match_fundamental_matrix_ransac, match_essential_matrix_ransac, match_rel_pose_ransac, proj_match_points_ransac, calibrate_cameras

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

3D Metrology

match_fundamental_matrix_ransac (Image1, Image2 : : Rows1, Cols1, Rows2, Cols2, GrayMatchMethod, MaskSize, RowMove, ColMove, RowTolerance, ColTolerance, Rotation, MatchThreshold, EstimationMethod, DistanceThreshold, RandSeed : FMatrix, CovFMat, Error, Points1, Points2)

Compute the fundamental matrix for a pair of stereo images by automatically finding correspondences between image points.

Given a set of coordinates of characteristic points (Rows1, Cols1) and (Rows2, Cols2) in the stereo images Image1 and Image2, match_fundamental_matrix_ransac automatically finds the correspondences between the characteristic points and determines the geometry of the stereo setup. For unknown cameras the geometry of the stereo setup is represented by the fundamental matrix FMatrix and all corresponding points have to fulfill the epipolar constraint, namely:

$$\begin{pmatrix} \text{Cols2} \\ \text{Rows2} \\ 1 \end{pmatrix}^T \cdot \text{FMatrix} \cdot \begin{pmatrix} \text{Cols1} \\ \text{Rows1} \\ 1 \end{pmatrix} = 0 \quad .$$

Note the column/row ordering in the point coordinates: because the fundamental matrix encodes the projective relation between two stereo images embedded in 3D space, the x/y notation has to be compliant with the camera coordinate system. So, (x,y) coordinates correspond to (column,row) pairs.

The matching process is based on characteristic points, which can be extracted with point operators like points_foerstner or points_harris. The matching itself is carried out in two steps: first, gray value correlations of mask windows around the input points in the first and the second image are determined and an initial matching between them is generated using the similarity of the windows in both images. Then, the RANSAC algorithm is applied to find the fundamental matrix that maximizes the number of correspondences under the epipolar constraint.

The size of the mask windows is MaskSize × MaskSize. Three metrics for the correlation can be selected. If GrayMatchMethod has the value *'ssd'*, the sum of the squared gray value differences is used, *'sad'* means the sum of absolute differences, and *'ncc'* is the normalized cross correlation. For details please refer to binocular_disparity. The metric is minimized (*'ssd'*, *'sad'*) or maximized (*'ncc'*) over all possible point pairs. A thus found matching is only accepted if the value of the metric is below the value of MatchThreshold (*'ssd'*, *'sad'*) or above that value (*'ncc'*).

To increase the speed of the algorithm the search area for the matching operations can be limited. Only points within a window of $2 \cdot$ RowTolerance $\times 2 \cdot$ ColTolerance points are considered. The offset of the center of the search window in the second image with respect to the position of the current point in the first image is given by RowMove and ColMove.

If the second camera is rotated around the optical axis with respect to the first camera the parameter Rotation may contain an estimate for the rotation angle or an angle interval in radians. A good guess will increase the quality of the gray value matching. If the actual rotation differs too much from the specified estimate the matching will typically fail. In this case, an angle interval should be specified and Rotation is a tuple with two elements. The larger the given interval the slower the operator is since the RANSAC algorithm is run over all angle increments within the interval.

After the initial matching is completed a randomized search algorithm (RANSAC) is used to determine the fundamental matrix FMatrix. It tries to find the matrix that is consistent with a maximum number of correspondences. For a point to be accepted, the distance to its corresponding epipolar line must not exceed the threshold DistanceThreshold.

The parameter EstimationMethod decides whether the relative orientation between the cameras is of a special type and which algorithm is to be applied for its computation. If EstimationMethod is either *'normalized_dlt'* or *'gold_standard'* the relative orientation is arbitrary. If left and right camera are identical and the relative orientation between them is a pure translation then choose EstimationMethod equal to *'trans_normalized_dlt'* or *'trans_gold_standard'*. The typical application for this special motion case is the scenario of a single fixed camera looking onto a moving conveyor belt. In order to get a unique solution in the correspondence problem the minimum required number of corresponding points is eight in the general case and three in the special, translational case.

The fundamental matrix is computed by a linear algorithm if *'normalized_dlt'* or *'trans_normalized_dlt'* is chosen. With *'gold_standard'* or *'trans_gold_standard'* the algorithm gives a statistically optimal result, and returns as well the covariance of the fundamental matrix CovFMat. Here, *'normalized_dlt'* and *'gold_standard'* stand for direct-linear-transformation and gold-standard-algorithm respectively.

The value Error indicates the overall quality of the estimation procedure and is the mean Euclidean distance in pixels between the points and their corresponding epipolar lines.

Point pairs consistent with the mentioned constraints are considered to be in correspondences. Points1 contains the indices of the matched input points from the first image and Points2 contains the indices of the corresponding points in the second image.

The parameter RandSeed can be used to control the randomized nature of the RANSAC algorithm, and hence to obtain reproducible results. If RandSeed is set to a positive number the operator yields the same result on every call with the same parameters because the internally used random number generator is initialized with the RandSeed. If RandSeed $=$ *0* the random number generator is initialized with the current time. In this case the results may not be reproducible. The value set for the HALCON system variable *'seed_rand'* (see set_system) does not affect the results of match_fundamental_matrix_ransac.

Parameters

- ▷ **Image1** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image 1.
- ▷ **Image2** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image 2.
- ▷ **Rows1** (input_control) . . . . . number-array ⇝ *real* / integer
  Row coordinates of characteristic points in image 1.
  Restriction: length(Rows1) >= 8 || length(Rows1) >= 3
- ▷ **Cols1** (input_control) . . . . . number-array ⇝ *real* / integer
  Column coordinates of characteristic points in image 1.
  Restriction: length(Cols1) == length(Rows1)
- ▷ **Rows2** (input_control) . . . . . number-array ⇝ *real* / integer
  Row coordinates of characteristic points in image 2.
  Restriction: length(Rows2) >= 8 || length(Rows2) >= 3
- ▷ **Cols2** (input_control) . . . . . number-array ⇝ *real* / integer
  Column coordinates of characteristic points in image 2.
  Restriction: length(Cols2) == length(Rows2)

▷ **GrayMatchMethod** (input_control) . . . . . string ⇝ *string*
Gray value comparison metric.
Default: 'ssd'
List of values: GrayMatchMethod ∈ {'ssd', 'sad', 'ncc'}

▷ **MaskSize** (input_control) . . . . . integer ⇝ *integer*
Size of gray value masks.
Default: 10
Suggested values: MaskSize ∈ {3, 7, 15}
Value range: 1 ≤ MaskSize

▷ **RowMove** (input_control) . . . . . integer ⇝ *integer*
Average row coordinate shift of corresponding points.
Default: 0
Value range: 0 ≤ RowMove ≤ 200

▷ **ColMove** (input_control) . . . . . integer ⇝ *integer*
Average column coordinate shift of corresponding points.
Default: 0
Value range: 0 ≤ ColMove ≤ 200

▷ **RowTolerance** (input_control) . . . . . integer ⇝ *integer*
Half height of matching search window.
Default: 200
Value range: 1 ≤ RowTolerance

▷ **ColTolerance** (input_control) . . . . . integer ⇝ *integer*
Half width of matching search window.
Default: 200
Value range: 1 ≤ ColTolerance

▷ **Rotation** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
Estimate of the relative orientation of the right image with respect to the left image.
Default: 0.0
Suggested values: Rotation ∈ {0.0, 0.1, -0.1, 0.7854, 1.571, 3.142}

▷ **MatchThreshold** (input_control) . . . . . number ⇝ *integer* / real
Threshold for gray value matching.
Default: 10
Suggested values: MatchThreshold ∈ {10, 20, 50, 100, 0.9, 0.7}

▷ **EstimationMethod** (input_control) . . . . . string ⇝ *string*
Algorithm for the computation of the fundamental matrix and for special camera orientations.
Default: 'normalized_dlt'
List of values: EstimationMethod ∈ {'normalized_dlt', 'gold_standard', 'trans_normalized_dlt', 'trans_gold_standard'}

▷ **DistanceThreshold** (input_control) . . . . . number ⇝ *real* / integer
Maximal deviation of a point from its epipolar line.
Default: 1
Value range: 0.5 ≤ DistanceThreshold ≤ 5
Restriction: DistanceThreshold > 0

▷ **RandSeed** (input_control) . . . . . integer ⇝ *integer*
Seed for the random number generator.
Default: 0

▷ **FMatrix** (output_control) . . . . . hom_mat2d ⇝ *real*
Computed fundamental matrix.

▷ **CovFMat** (output_control) . . . . . real-array ⇝ *real*
9×9 covariance matrix of the fundamental matrix.

▷ **Error** (output_control) . . . . . real ⇝ *real*
Root-Mean-Square of the epipolar distance error.

▷ **Points1** (output_control) . . . . . integer-array ⇝ *integer*
Indices of matched input points in image 1.

▷ **Points2** (output_control) . . . . . integer-array ⇝ *integer*
Indices of matched input points in image 2.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_foerstner, points_harris

Possible Successors

vector_to_fundamental_matrix, gen_binocular_proj_rectification

See also

match_essential_matrix_ransac, match_rel_pose_ransac, proj_match_points_ransac

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

3D Metrology

match_rel_pose_ransac (Image1, Image2 : : Rows1, Cols1, Rows2, Cols2, CamPar1, CamPar2, GrayMatchMethod, MaskSize, RowMove, ColMove, RowTolerance, ColTolerance, Rotation, MatchThreshold, EstimationMethod, DistanceThreshold, RandSeed : RelPose, CovRelPose, Error, Points1, Points2)

Compute the relative orientation between two cameras by automatically finding correspondences between image points.

Given a set of coordinates of characteristic points (Rows1,Cols1) and (Rows2,Cols2) in the stereo images Image1 and Image2 along with known internal camera parameters CamPar1 and CamPar2, match_rel_pose_ransac automatically determines the geometry of the stereo setup and finds the correspondences between the characteristic points. The geometry of the stereo setup is represented by the relative pose RelPose and all corresponding points have to fulfill the epipolar constraint. RelPose indicates the relative pose of camera 1 with respect to camera 2 (See create_pose for more information about poses and their representations.). This is in accordance with the explicit calibration of a stereo setup using the operator calibrate_cameras. Now, let R, t be the rotation and translation of the relative pose. Then, the essential matrix E is defined as $E = ([t]_\times R)^T$, where $[t]_\times$ denotes the 3×3 skew-symmetric matrix realizing the cross product with the vector t. The pose can be determined from the epipolar constraint:

$$\begin{pmatrix} X_2 \\ Y_2 \\ 1 \end{pmatrix}^T \cdot ([t]_\times R)^T \cdot \begin{pmatrix} X_1 \\ Y_1 \\ 1 \end{pmatrix} = 0 \quad \text{where} \quad [t]_\times = \begin{pmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{pmatrix} .$$

Note, that the essential matrix is a projective entity and thus is defined up to a scaling factor. From this follows that the translation vector of the relative pose can only be determined up to scale too. In fact, the computed translation vector will always be normalized to unit length. As a consequence, a subsequent three-dimensional reconstruction of the scene, using for instance vector_to_rel_pose, can be carried out only up to a single global scaling factor.

The operator match_rel_pose_ransac is designed to deal with a camera model, that includes lens distortions. This is in contrast to the operator match_essential_matrix_ransac, which encompasses only straight line preserving cameras. The camera parameters are passed in CamPar1 and CamPar2. The 3D direction vectors $(X_1, Y_1, 1)$ and $(X_2, Y_2, 1)$ are calculated from the point coordinates (Rows1,Cols1) and (Rows2,Cols2) by inverting the process of projection (see Calibration).

The matching process is based on characteristic points, which can be extracted with point operators like points_foerstner or points_harris. The matching itself is carried out in two steps: first, gray value correlations of mask windows around the input points in the first and the second image are determined and an initial matching between them is generated using the similarity of the windows in both images. Then, the RANSAC algorithm is applied to find the relative pose that maximizes the number of correspondences under the epipolar constraint.

The size of the mask windows is MaskSize $\times$ MaskSize. Three metrics for the correlation can be selected. If GrayMatchMethod has the value *'ssd'*, the sum of the squared gray value differences is used, *'sad'* means the sum of absolute differences, and *'ncc'* is the normalized cross correlation. For details please refer to binocular_disparity. The metric is minimized (*'ssd'*, *'sad'*) or maximized (*'ncc'*) over all possible point pairs. A thus found matching is only accepted if the value of the metric is below the value of MatchThreshold (*'ssd'*, *'sad'*) or above that value (*'ncc'*).

To increase the speed of the algorithm, the search area for the matching operations can be limited. Only points within a window of $2 \cdot$ RowTolerance $\times 2 \cdot$ ColTolerance points are considered. The offset of the center of the search window in the second image with respect to the position of the current point in the first image is given by RowMove and ColMove.

If the second camera is rotated around the optical axis with respect to the first camera the parameter Rotation may contain an estimate for the rotation angle or an angle interval in radians. A good guess will increase the quality of the gray value matching. If the actual rotation differs too much from the specified estimate the matching will typically fail. In this case, an angle interval should be specified, and Rotation is a tuple with two elements. The larger the given interval the slower the operator is since the RANSAC algorithm is run over all angle increments within the interval.

After the initial matching is completed a randomized search algorithm (RANSAC) is used to determine the relative pose RelPose. It tries to find the relative pose that is consistent with a maximum number of correspondences. For a point to be accepted, the distance to its corresponding epipolar line must not exceed the threshold DistanceThreshold.

The parameter EstimationMethod decides whether the relative orientation between the cameras is of a special type and which algorithm is to be applied for its computation. If EstimationMethod is either *'normalized_dlt'* or *'gold_standard'* the relative orientation is arbitrary. Choosing *'trans_normalized_dlt'* or *'trans_gold_standard'* means that the relative motion between the cameras is a pure translation. The typical application for this special motion case is the scenario of a single fixed camera looking onto a moving conveyor belt. In order to get a unique solution in the correspondence problem the minimum required number of corresponding points is six in the general case and three in the special, translational case.

The relative pose is computed by a linear algorithm if *'normalized_dlt'* or *'trans_normalized_dlt'* is chosen. With *'gold_standard'* or *'trans_gold_standard'* the algorithm gives a statistically optimal result, and returns as well the covariance of the relative pose CovRelPose. Here, *'normalized_dlt'* and *'gold_standard'* stand for direct-linear-transformation and gold-standard-algorithm respectively. Note, that in general the found correspondences differ depending on the deployed estimation method.

The value Error indicates the overall quality of the estimation procedure and is the mean Euclidean distance in pixels between the points and their corresponding epipolar lines.

Point pairs consistent with the mentioned constraints are considered to be in correspondences. Points1 contains the indices of the matched input points from the first image and Points2 contains the indices of the corresponding points in the second image.

For the operator match_rel_pose_ransac a special configuration of scene points and cameras exists: if all 3D points lie in a single plane and additionally are all closer to one of the two cameras then the solution in the essential matrix is not unique but twofold. As a consequence both solutions are computed and returned by the operator. This means that the output parameters RelPose, CovRelPose and Error are of double length and the values of the second solution are simply concatenated behind the values of the first one.

The parameter RandSeed can be used to control the randomized nature of the RANSAC algorithm, and hence to obtain reproducible results. If RandSeed is set to a positive number the operator yields the same result on every call with the same parameters because the internally used random number generator is initialized with the RandSeed. If RandSeed $=$ *0* the random number generator is initialized with the current time. In this case the results may not be reproducible. The value set for the HALCON system variable *'seed_rand'* (see set_system) does not affect the results of match_rel_pose_ransac.

Parameters

▷ **Image1** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image 1.

▷ **Image2** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image 2.

▷ **Rows1** (input_control) . . . . . number-array ⇝ *real* / integer
Row coordinates of characteristic points in image 1.
Restriction: length(Rows1) >= 6 || length(Rows1) >= 3

▷ **Cols1** (input_control) . . . . . number-array ⇝ *real* / integer
Column coordinates of characteristic points in image 1.
Restriction: length(Cols1) == length(Rows1)

▷ **Rows2** (input_control) . . . . . number-array ⇝ *real* / integer
Row coordinates of characteristic points in image 2.
Restriction: length(Rows2) >= 6 || length(Rows2) >= 3

▷ **Cols2** (input_control) . . . . . number-array ⇝ *real* / integer
Column coordinates of characteristic points in image 2.
Restriction: length(Cols2) == length(Rows2)

▷ **CamPar1** (input_control) . . . . . campar ⇝ *real* / integer / string
Parameters of the 1st camera.

▷ **CamPar2** (input_control) . . . . . campar ⇝ *real* / integer / string
Parameters of the 2nd camera.

▷ **GrayMatchMethod** (input_control) . . . . . string ⇝ *string*
Gray value comparison metric.
Default: 'ssd'
List of values: `GrayMatchMethod` ∈ {'ssd', 'sad', 'ncc'}

▷ **MaskSize** (input_control) . . . . . integer ⇝ *integer*
Size of gray value masks.
Default: 10
Suggested values: `MaskSize` ∈ {3, 7, 15}
Value range: 1 ≤ MaskSize

▷ **RowMove** (input_control) . . . . . integer ⇝ *integer*
Average row coordinate shift of corresponding points.
Default: 0
Value range: 0 ≤ `RowMove` ≤ 200

▷ **ColMove** (input_control) . . . . . integer ⇝ *integer*
Average column coordinate shift of corresponding points.
Default: 0
Value range: 0 ≤ `ColMove` ≤ 200

▷ **RowTolerance** (input_control) . . . . . integer ⇝ *integer*
Half height of matching search window.
Default: 200
Value range: 1 ≤ RowTolerance

▷ **ColTolerance** (input_control) . . . . . integer ⇝ *integer*
Half width of matching search window.
Default: 200
Value range: 1 ≤ ColTolerance

▷ **Rotation** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
Estimate of the relative orientation of the right image with respect to the left image.
Default: 0.0
Suggested values: `Rotation` ∈ {0.0, 0.1, -0.1, 0.7854, 1.571, 3.142}

▷ **MatchThreshold** (input_control) . . . . . number ⇝ *integer* / real
Threshold for gray value matching.
Default: 10
Suggested values: `MatchThreshold` ∈ {10, 20, 50, 100, 0.9, 0.7}

▷ **EstimationMethod** (input_control) ..... string ⇝ *string*
Algorithm for the computation of the relative pose and for special pose types.
Default: 'normalized_dlt'
List of values: EstimationMethod ∈ {'normalized_dlt', 'gold_standard', 'trans_normalized_dlt', 'trans_gold_standard'}

▷ **DistanceThreshold** (input_control) ..... number ⇝ *real / integer*
Maximal deviation of a point from its epipolar line.
Default: 1
Value range: 0.5 ≤ DistanceThreshold ≤ 5
Restriction: DistanceThreshold > 0

▷ **RandSeed** (input_control) ..... integer ⇝ *integer*
Seed for the random number generator.
Default: 0

▷ **RelPose** (output_control) ..... pose ⇝ *real / integer*
Computed relative orientation of the cameras (3D pose).

▷ **CovRelPose** (output_control) ..... real-array ⇝ *real*
6 × 6 covariance matrix of the relative orientation.

▷ **Error** (output_control) ..... real(-array) ⇝ *real*
Root-Mean-Square of the epipolar distance error.

▷ **Points1** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 1.

▷ **Points2** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 2.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_foerstner, points_harris

Possible Successors

vector_to_rel_pose, gen_binocular_rectification_map

See also

binocular_calibration, match_fundamental_matrix_ransac, match_essential_matrix_ransac, create_pose

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

3D Metrology

reconst3d_from_fundamental_matrix (: : Rows1, Cols1, Rows2, Cols2, CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2, FMatrix, CovFMat : X, Y, Z, W, CovXYZW)

Compute the projective 3d reconstruction of points based on the fundamental matrix.

A pair of stereo images is called weakly calibrated if the fundamental matrix, which defines the geometric relation between the two images, is known. Given such a fundamental matrix FMatrix and a set of corresponding points (Rows1,Cols1) and (Rows2,Cols2) the operator reconst3d_from_fundamental_matrix determines the three-dimensional space points projecting onto these image points. This 3D reconstruction is purely projective

and the projective coordinates are returned by the four-vector (X,Y,Z,W). This type of reconstruction is also known as projective triangulation. If additionally the covariances CovRR1, CovRC1, CovCC1 and CovRR2, CovRC2, CovCC2 of the image points are given the covariances of the reconstructed points CovXYZW are computed too. Let n be the number of points. Then the concatenated covariances are stored in a $16 \times n$ tuple. The computation of the covariances is more precise if the covariance of the fundamental matrix CovFMat is provided.

The operator reconst3d_from_fundamental_matrix is typically used after match_fundamental_matrix_ransac to perform 3d reconstruction. This will save computational cost compared with the deployment of vector_to_fundamental_matrix.

reconst3d_from_fundamental_matrix is the projective equivalent to the Euclidean reconstruction operator intersect_lines_of_sight.

Parameters

- ▷ **Rows1** (input_control) . . . number(-array) ⇝ *real* / integer
  Input points in image 1 (row coordinate).
- ▷ **Cols1** (input_control) . . . number(-array) ⇝ *real* / integer
  Input points in image 1 (column coordinate).
- ▷ **Rows2** (input_control) . . . number(-array) ⇝ *real* / integer
  Input points in image 2 (row coordinate).
- ▷ **Cols2** (input_control) . . . number(-array) ⇝ *real* / integer
  Input points in image 2 (column coordinate).
- ▷ **CovRR1** (input_control) . . . number(-array) ⇝ *real* / integer
  Row coordinate variance of the points in image 1.
  Default: []
- ▷ **CovRC1** (input_control) . . . number(-array) ⇝ *real* / integer
  Covariance of the points in image 1.
  Default: []
- ▷ **CovCC1** (input_control) . . . number(-array) ⇝ *real* / integer
  Column coordinate variance of the points in image 1.
  Default: []
- ▷ **CovRR2** (input_control) . . . number(-array) ⇝ *real* / integer
  Row coordinate variance of the points in image 2.
  Default: []
- ▷ **CovRC2** (input_control) . . . number(-array) ⇝ *real* / integer
  Covariance of the points in image 2.
  Default: []
- ▷ **CovCC2** (input_control) . . . number(-array) ⇝ *real* / integer
  Column coordinate variance of the points in image 2.
  Default: []
- ▷ **FMatrix** (input_control) . . . hom_mat2d ⇝ *real*
  Fundamental matrix.
- ▷ **CovFMat** (input_control) . . . real-array ⇝ *real*
  9×9 covariance matrix of the fundamental matrix.
  Default: []
- ▷ **X** (output_control) . . . real(-array) ⇝ *real*
  X coordinates of the reconstructed points in projective 3D space.
- ▷ **Y** (output_control) . . . real(-array) ⇝ *real*
  Y coordinates of the reconstructed points in projective 3D space.
- ▷ **Z** (output_control) . . . real(-array) ⇝ *real*
  Z coordinates of the reconstructed points in projective 3D space.
- ▷ **W** (output_control) . . . real(-array) ⇝ *real*
  W coordinates of the reconstructed points in projective 3D space.
- ▷ **CovXYZW** (output_control) . . . real(-array) ⇝ *real*
  Covariance matrices of the reconstructed points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

match_fundamental_matrix_ransac

Alternatives

vector_to_fundamental_matrix, intersect_lines_of_sight

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2000.

Module

3D Metrology

rel_pose_to_fundamental_matrix (: : RelPose, CovRelPose, CamPar1, CamPar2 : FMatrix, CovFMat)

Compute the fundamental matrix from the relative orientation of two cameras.

Cameras including lens distortions can be modeled by the following set of parameters: the focal length f, two scaling factors s_x, s_y, the coordinates of the principal point (c_x, c_y) and the distortion coefficient κ. For a more detailed description see the chapter Calibration. Only cameras with a distortion coefficient equal to zero project straight lines in the world onto straight lines in the image. This is also true for telecentric cameras and for cameras with tilt lenses. rel_pose_to_fundamental_matrix handles telecentric lenses and tilt lenses correctly. However, for reasons of simplicity, these lens types are ignored in the formulas below. If the distortion coefficient is equal to zero, image projection is a linear mapping and the camera, i.e., the set of internal parameters, can be described by the camera matrix $CamMat$:

$$CamMat = \begin{pmatrix} f/s_x & 0 & c_x \\ 0 & f/s_y & c_y \\ 0 & 0 & 1 \end{pmatrix} .$$

Going from a nonlinear model to a linear model is an approximation of the real underlying camera. For a variety of camera lenses, especially lenses with long focal length, the error induced by this approximation can be neglected. Following the formula $E = ([t]_\times R)^T$, the essential matrix E is derived from the translation t and the rotation R of the relative pose RelPose (see also operator vector_to_rel_pose). In the linearized framework the fundamental matrix can be calculated from the relative pose and the camera matrices according to the formula presented under essential_to_fundamental_matrix:

$$\texttt{FMatrix} = CamMat2^{-T} \cdot ([t]_\times R)^T \cdot CamMat1^{-1} .$$

The transformation from a relative pose to a fundamental matrix goes along with the propagation of the covariance matrices CovRelPose to CovFMat. If CovRelPose is empty CovFMat will be empty too.

The conversion operator rel_pose_to_fundamental_matrix is used especially for a subsequent visualization of the epipolar line structure via the fundamental matrix, which depicts the underlying stereo geometry.

Parameters

▷ **RelPose** (input_control) . . . . . pose ⇝ *real* / integer
Relative orientation of the cameras (3D pose).

▷ **CovRelPose** (input_control) . . . . . number-array ⇝ *real* / integer
6×6 covariance matrix of relative pose.
Default: []

▷ **CamPar1** (input_control) . . . . . campar ⇝ *real* / integer / string
Parameters of the 1. camera.

▷ **CamPar2** (input_control) . . . . . campar ⇝ *real* / integer / string
Parameters of the 2. camera.

▷ **FMatrix** (output_control) . . . . . hom_mat2d ⇝ *real*
Computed fundamental matrix.
▷ **CovFMat** (output_control) . . . . . real-array ⇝ *real*
9×9 covariance matrix of the fundamental matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

vector_to_rel_pose

Alternatives

essential_to_fundamental_matrix

See also

calibrate_cameras

Module

3D Metrology

vector_to_essential_matrix (: : Rows1, Cols1, Rows2, Cols2, CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2, CamMat1, CamMat2, Method : EMatrix, CovEMat, Error, X, Y, Z, CovXYZ)

Compute the essential matrix given image point correspondences and known camera matrices and reconstruct 3D points.

For a stereo configuration with known camera matrices the geometric relation between the two images is defined by the essential matrix. The operator vector_to_essential_matrix determines the essential matrix EMatrix from in general at least six given point correspondences, that fulfill the epipolar constraint:

$$\begin{pmatrix} X_2 \\ Y_2 \\ 1 \end{pmatrix}^T \cdot EMatrix \cdot \begin{pmatrix} X_1 \\ Y_1 \\ 1 \end{pmatrix} = 0$$

The operator vector_to_essential_matrix is designed to deal only with a linear camera model. This is in contrast to the operator vector_to_rel_pose, that encompasses lens distortions too. The internal camera parameters are passed by the arguments CamMat1 and CamMat2, which are 3×3 upper triangular matrices describing an affine transformation. The relation between the vector (X,Y,1), defining the direction from the camera to the viewed 3D point, and its (projective) 2D image coordinates (col,row,1) is:

$$\begin{pmatrix} col \\ row \\ 1 \end{pmatrix} = CamMat \cdot \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \quad \text{where} \quad CamMat = \begin{pmatrix} f/s_x & s & c_x \\ 0 & f/s_y & c_y \\ 0 & 0 & 1 \end{pmatrix} .$$

The focal length is denoted by f, s_x, s_y are scaling factors, s describes a skew factor and (c_x, c_y) indicates the principal point. Mainly, these are the elements known from the camera parameters as used for example in calibrate_cameras. Alternatively, the elements of the camera matrix can be described in a different way, see e.g. stationary_camera_self_calibration.

The point correspondences (Rows1,Cols1) and (Rows2,Cols2) are typically found by applying the operator match_essential_matrix_ransac. Multiplying the image coordinates by the inverse of the camera matrices results in the 3D direction vectors, which can then be inserted in the epipolar constraint.

The parameter Method decides whether the relative orientation between the cameras is of a special type and which algorithm is to be applied for its computation. If Method is either *'normalized_dlt'* or *'gold_standard'* the relative orientation is arbitrary. Choosing *'trans_normalized_dlt'* or *'trans_gold_standard'* means that the relative motion between the cameras is a pure translation. The typical application for this special motion case is the scenario

of a single fixed camera looking onto a moving conveyor belt. In this case the minimum required number of corresponding points is just two instead of six in the general case.

The essential matrix is computed by a linear algorithm if *'normalized_dlt'* or *'trans_normalized_dlt'* is chosen. With *'gold_standard'* or *'trans_gold_standard'* the algorithm gives a statistically optimal result. Here, *'normalized_dlt'* and *'gold_standard'* stand for direct-linear-transformation and gold-standard-algorithm respectively. All methods return the coordinates (X,Y,Z) of the reconstructed 3D points. The optimal methods also return the covariances of the 3D points in CovXYZ. Let n be the number of points then the 3×3 covariance matrices are concatenated and stored in a tuple of length $9n$. Additionally, the optimal methods return the covariance of the essential matrix CovEMat.

If an optimal gold-standard-algorithm is chosen the covariances of the image points (CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2) can be incorporated in the computation. They can be provided for example by the operator points_foerstner. If the point covariances are unknown, which is the default, empty tuples are input. In this case the optimization algorithm internally assumes uniform and equal covariances for all points.

The value Error indicates the overall quality of the optimization process and is the root-mean-square Euclidean distance in pixels between the points and their corresponding epipolar lines.

For the operator vector_to_essential_matrix a special configuration of scene points and cameras exists: if all 3D points lie in a single plane and additionally are all closer to one of the two cameras then the solution in the essential matrix is not unique but twofold. As a consequence both solutions are computed and returned by the operator. This means that all output parameters are of double length and the values of the second solution are simply concatenated behind the values of the first one.

Parameters

- ▷ **Rows1** (input_control) . . . number-array ⇝ *real* / integer
  Input points in image 1 (row coordinate).
  Restriction: length(Rows1) >= 6 || length(Rows1) >= 2
- ▷ **Cols1** (input_control) . . . number-array ⇝ *real* / integer
  Input points in image 1 (column coordinate).
  Restriction: length(Cols1) == length(Rows1)
- ▷ **Rows2** (input_control) . . . number-array ⇝ *real* / integer
  Input points in image 2 (row coordinate).
  Restriction: length(Rows2) == length(Rows1)
- ▷ **Cols2** (input_control) . . . number-array ⇝ *real* / integer
  Input points in image 2 (column coordinate).
  Restriction: length(Cols2) == length(Rows1)
- ▷ **CovRR1** (input_control) . . . number-array ⇝ *real* / integer
  Row coordinate variance of the points in image 1.
  Default: []
- ▷ **CovRC1** (input_control) . . . number-array ⇝ *real* / integer
  Covariance of the points in image 1.
  Default: []
- ▷ **CovCC1** (input_control) . . . number-array ⇝ *real* / integer
  Column coordinate variance of the points in image 1.
  Default: []
- ▷ **CovRR2** (input_control) . . . number-array ⇝ *real* / integer
  Row coordinate variance of the points in image 2.
  Default: []
- ▷ **CovRC2** (input_control) . . . number-array ⇝ *real* / integer
  Covariance of the points in image 2.
  Default: []
- ▷ **CovCC2** (input_control) . . . number-array ⇝ *real* / integer
  Column coordinate variance of the points in image 2.
  Default: []
- ▷ **CamMat1** (input_control) . . . hom_mat2d ⇝ *real* / integer
  Camera matrix of the 1st camera.
- ▷ **CamMat2** (input_control) . . . hom_mat2d ⇝ *real* / integer
  Camera matrix of the 2nd camera.

▷ **Method** (input_control) . . . . . string ⇝ *string*
Algorithm for the computation of the essential matrix and for special camera orientations.
Default: 'normalized_dlt'
List of values: Method ∈ {'normalized_dlt', 'gold_standard', 'trans_normalized_dlt', 'trans_gold_standard'}

▷ **EMatrix** (output_control) . . . . . hom_mat2d ⇝ *real*
Computed essential matrix.

▷ **CovEMat** (output_control) . . . . . real-array ⇝ *real*
9×9 covariance matrix of the essential matrix.

▷ **Error** (output_control) . . . . . real(-array) ⇝ *real*
Root-Mean-Square of the epipolar distance error.

▷ **X** (output_control) . . . . . real-array ⇝ *real*
X coordinates of the reconstructed 3D points.

▷ **Y** (output_control) . . . . . real-array ⇝ *real*
Y coordinates of the reconstructed 3D points.

▷ **Z** (output_control) . . . . . real-array ⇝ *real*
Z coordinates of the reconstructed 3D points.

▷ **CovXYZ** (output_control) . . . . . real-array ⇝ *real*
Covariance matrices of the reconstructed 3D points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

match_essential_matrix_ransac

Possible Successors

essential_to_fundamental_matrix

Alternatives

vector_to_rel_pose, vector_to_fundamental_matrix

See also

stationary_camera_self_calibration

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

J.Chris McGlone (editor): "Manual of Photogrammetry"; American Society for Photogrammetry and Remote Sensing ; 2004.

Module

3D Metrology

vector_to_fundamental_matrix (: : Rows1, Cols1, Rows2, Cols2, CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2, Method : FMatrix, CovFMat, Error, X, Y, Z, W, CovXYZW)

Compute the fundamental matrix given a set of image point correspondences and reconstruct 3D points.

For a stereo configuration with unknown camera parameters the geometric relation between the two images is defined by the fundamental matrix. The operator vector_to_fundamental_matrix determines the fundamental matrix FMatrix from given point correspondences (Rows1,Cols1), (Rows2,Cols2), that fulfill the epipolar constraint:

$$\begin{pmatrix} \texttt{Cols2} \\ \texttt{Rows2} \\ 1 \end{pmatrix}^T \cdot \texttt{FMatrix} \cdot \begin{pmatrix} \texttt{Cols1} \\ \texttt{Rows1} \\ 1 \end{pmatrix} = 0 \ .$$

Note the column/row ordering in the point coordinates: since the fundamental matrix encodes the projective relation between two stereo images embedded in 3D space, the x/y notation must be compliant with the camera coordinate system. Therefore, (x,y) coordinates correspond to (column,row) pairs.

For a general relative orientation of the two cameras the minimum number of required point correspondences is eight. Then, `Method` is chosen to be *'normalized_dlt'* or *'gold_standard'*. If left and right camera are identical and the relative orientation between them is a pure translation then choose `Method` equal to *'trans_normalized_dlt'* or *'trans_gold_standard'*. In this special case the minimum number of correspondences is only two. The typical application of the motion being a pure translation is that of a single fixed camera looking onto a moving conveyor belt.

The fundamental matrix is determined by minimizing a cost function. To minimize the respective error different algorithms are available, and the user can choose between the direct-linear-transformation ('normalized_dlt') and the gold-standard-algorithm ('gold_standard'). Like the motion case, the algorithm can be selected with the parameter `Method`. For `Method` = *'normalized_dlt'* or *'trans_normalized_dlt'*, a linear algorithm minimizes an algebraic error based on the above epipolar constraint. This algorithm offers a good compromise between speed and accuracy. For `Method` = *'gold_standard'* or *'trans_gold_standard'*, a mathematically optimal, but slower optimization is used, which minimizes the geometric backprojection error of reconstructed projective 3D points. In this case, in addition to the fundamental matrix its covariance matrix `CovFMat` is output, along with the projective coordinates (`X`,`Y`,`Z`,`W`) of the reconstructed points and their covariances `CovXYZW`. Let n be the number of points. Then the concatenated covariances are stored in a $16 \times n$ tuple.

If an optimal gold-standard-algorithm is chosen the covariances of the image points (`CovRR1`, `CovRC1`, `CovCC1`, `CovRR2`, `CovRC2`, `CovCC2`) can be incorporated in the computation. They can be provided for example by the operator `points_foerstner`. If the point covariances are unknown, which is the default, empty tuples are input. In this case the optimization algorithm internally assumes uniform and equal covariances for all points.

The value `Error` indicates the overall quality of the optimization procedure and is the mean Euclidean distance in pixels between the points and their corresponding epipolar lines.

If the correspondence between the points are not known, `match_fundamental_matrix_ransac` should be used instead.

Parameters

▷ **Rows1** (input_control) . . . number-array ⇝ *real* / integer
Input points in image 1 (row coordinate).
Restriction: length(Rows1) >= 8 || length(Rows1) >= 2

▷ **Cols1** (input_control) . . . number-array ⇝ *real* / integer
Input points in image 1 (column coordinate).
Restriction: length(Cols1) == length(Rows1)

▷ **Rows2** (input_control) . . . number-array ⇝ *real* / integer
Input points in image 2 (row coordinate).
Restriction: length(Rows2) == length(Rows1)

▷ **Cols2** (input_control) . . . number-array ⇝ *real* / integer
Input points in image 2 (column coordinate).
Restriction: length(Cols2) == length(Rows1)

▷ **CovRR1** (input_control) . . . number-array ⇝ *real* / integer
Row coordinate variance of the points in image 1.
Default: []

▷ **CovRC1** (input_control) . . . number-array ⇝ *real* / integer
Covariance of the points in image 1.
Default: []

▷ **CovCC1** (input_control) . . . number-array ⇝ *real* / integer
Column coordinate variance of the points in image 1.
Default: []

▷ **CovRR2** (input_control) . . . number-array ⇝ *real* / integer
Row coordinate variance of the points in image 2.
Default: []

▷ **CovRC2** (input_control) . . . number-array ⇝ *real* / integer
Covariance of the points in image 2.
Default: []

▷ **CovCC2** (input_control) . . . . . number-array ⇝ *real* / integer
Column coordinate variance of the points in image 2.
Default: []

▷ **Method** (input_control) . . . . . string ⇝ *string*
Estimation algorithm.
Default: 'normalized_dlt'
List of values: Method ∈ {'normalized_dlt', 'gold_standard', 'trans_normalized_dlt', 'trans_gold_standard'}

▷ **FMatrix** (output_control) . . . . . hom_mat2d ⇝ *real*
Computed fundamental matrix.

▷ **CovFMat** (output_control) . . . . . real-array ⇝ *real*
9×9 covariance matrix of the fundamental matrix.

▷ **Error** (output_control) . . . . . real ⇝ *real*
Root-Mean-Square of the epipolar distance error.

▷ **X** (output_control) . . . . . real-array ⇝ *real*
X coordinates of the reconstructed points in projective 3D space.

▷ **Y** (output_control) . . . . . real-array ⇝ *real*
Y coordinates of the reconstructed points in projective 3D space.

▷ **Z** (output_control) . . . . . real-array ⇝ *real*
Z coordinates of the reconstructed points in projective 3D space.

▷ **W** (output_control) . . . . . real-array ⇝ *real*
W coordinates of the reconstructed points in projective 3D space.

▷ **CovXYZW** (output_control) . . . . . real-array ⇝ *real*
Covariance matrices of the reconstructed 3D points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

match_fundamental_matrix_ransac

Possible Successors

gen_binocular_proj_rectification

Alternatives

vector_to_essential_matrix, vector_to_rel_pose

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2000.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

3D Metrology

vector_to_fundamental_matrix_distortion (: : Rows1, Cols1, Rows2, Cols2, CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2, ImageWidth, ImageHeight, Method : FMatrix, Kappa, Error, X, Y, Z, W)

Compute the fundamental matrix and the radial distortion coefficient given a set of image point correspondences and reconstruct 3D points.

For a stereo configuration with unknown camera parameters, the geometric relation between the two images is defined by the fundamental matrix. vector_to_fundamental_matrix_distortion determines the fundamental matrix FMatrix and the radial distortion coefficient Kappa (κ) from given point correspondences (Rows1,Cols1), (Rows2,Cols2) that fulfill the epipolar constraint:

$$\begin{pmatrix} c_2 \\ r_2 \\ 1 \end{pmatrix}^T \cdot \texttt{FMatrix} \cdot \begin{pmatrix} c_1 \\ r_1 \\ 1 \end{pmatrix} = 0 \ .$$

Here, (r_1, c_1) and (r_2, c_2) denote image points that are obtained by undistorting the input image points with the division model (see Calibration):

$$r = \frac{\tilde{r}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)} \qquad c = \frac{\tilde{c}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)}$$

Here, $(\tilde{r}_1, \tilde{c}_1) = (\texttt{Rows1} - 0.5(\texttt{ImageHeight} - 1), \texttt{Cols1} - 0.5(\texttt{ImageWidth} - 1))$
and $(\tilde{r}_2, \tilde{c}_2) = (\texttt{Rows2} - 0.5(\texttt{ImageHeight} - 1), \texttt{Cols2} - 0.5(\texttt{ImageWidth} - 1))$
denote the distorted image points, specified relative to the image center. Thus, vector_to_fundamental_matrix_distortion assumes that the principal point of the camera, i.e., the center of the radial distortions, lies at the center of the image.

The returned Kappa can be used to construct camera parameters that can be used to rectify images or points (see change_radial_distortion_cam_par, change_radial_distortion_image, and change_radial_distortion_points):

$$\begin{aligned} \texttt{CamPar} \ = \ & [\text{'}area\_scan\_telecentric\_division\text{'}, 0.0, \texttt{Kappa}, 1.0, 1.0, 0.5(\texttt{ImageWidth} - 1), \\ & 0.5(\texttt{ImageHeight} - 1), \texttt{ImageWidth}, \texttt{ImageHeight}] \end{aligned}$$

Note the column/row ordering in the point coordinates above: since the fundamental matrix encodes the projective relation between two stereo images embedded in 3D space, the x/y notation must be compliant with the camera coordinate system. Therefore, (x,y) coordinates correspond to (column,row) pairs.

For a general relative orientation of the two cameras, the minimum number of required point correspondences is nine. Then, Method must be set to *'linear'* or *'gold_standard'*. If the left and right cameras are identical and the relative orientation between them is a pure translation, Method must be set to *'trans_linear'* or *'trans_gold_standard'*. In this special case, the minimum number of correspondences is only four. The typical application of the motion being a pure translation is a single fixed camera looking onto a moving conveyor belt.

The fundamental matrix is determined by minimizing a cost function. To minimize the respective error, different algorithms are available, and the user can choose between the linear (*'linear'*) and the gold-standard algorithm (*'gold_standard'*). Like the motion type, the algorithm can be selected with the parameter Method. For Method = *'linear'* or *'trans_linear'*, a linear algorithm that minimizes an algebraic error based on the above epipolar constraint is used. This algorithm is very fast. For the pure translation case (Method = *'trans_linear'*), the linear method returns accurate results for small to moderate noise of the point coordinates and for most distortions (except for very small distortions). For a general relative orientation of the two cameras (Method = *'linear'*), the linear method only returns accurate results for very small noise of the point coordinates and for sufficiently large distortions. For Method = *'gold_standard'* or *'trans_gold_standard'*, a mathematically optimal but slower optimization is used, which minimizes the geometric reprojection error of reconstructed projective 3D points. In this case, in addition to the fundamental matrix and the distortion coefficient, the projective coordinates (X,Y,Z,W) of the reconstructed points are returned. For a general relative orientation of the two cameras, in general Method = *'gold_standard'* should be selected.

If an optimal gold-standard algorithm is chosen, the covariances of the image points (CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2) can be incorporated into the computation. They can be provided, for example, by the operator points_foerstner. If the point covariances are unknown, which is the default, empty tuples are passed. In this case, the optimization algorithm internally assumes uniform and equal covariances for all points.

The value Error indicates the overall quality of the optimization procedure and is the mean symmetric Euclidean distance in pixels between the points and their corresponding epipolar lines.

If the correspondence between the points is not known, match_fundamental_matrix_distortion_ransac should be used instead.

Parameters

▷ **Rows1** (input_control) . . . . . point.y-array ⇝ *real* / integer
Input points in image 1 (row coordinate).
Restriction: length(Rows1) >= 9 || length(Rows1) >= 4

▷ **Cols1** (input_control) . . . . . point.x-array ⇝ *real* / integer
Input points in image 1 (column coordinate).
Restriction: length(Cols1) == length(Rows1)

▷ **Rows2** (input_control) . . . . . point.y-array ⇝ *real* / integer
Input points in image 2 (row coordinate).
Restriction: length(Rows2) == length(Rows1)

▷ **Cols2** (input_control) . . . . . point.x-array ⇝ *real* / integer
Input points in image 2 (column coordinate).
Restriction: length(Cols2) == length(Rows1)

▷ **CovRR1** (input_control) . . . . . number-array ⇝ *real* / integer
Row coordinate variance of the points in image 1.
Default: []

▷ **CovRC1** (input_control) . . . . . number-array ⇝ *real* / integer
Covariance of the points in image 1.
Default: []

▷ **CovCC1** (input_control) . . . . . number-array ⇝ *real* / integer
Column coordinate variance of the points in image 1.
Default: []

▷ **CovRR2** (input_control) . . . . . number-array ⇝ *real* / integer
Row coordinate variance of the points in image 2.
Default: []

▷ **CovRC2** (input_control) . . . . . number-array ⇝ *real* / integer
Covariance of the points in image 2.
Default: []

▷ **CovCC2** (input_control) . . . . . number-array ⇝ *real* / integer
Column coordinate variance of the points in image 2.
Default: []

▷ **ImageWidth** (input_control) . . . . . integer ⇝ *integer*
Width of the images from which the points were extracted.
Restriction: ImageWidth > 0

▷ **ImageHeight** (input_control) . . . . . integer ⇝ *integer*
Height of the images from which the points were extracted.
Restriction: ImageHeight > 0

▷ **Method** (input_control) . . . . . string ⇝ *string*
Estimation algorithm.
Default: 'gold_standard'
List of values: Method ∈ {'linear', 'gold_standard', 'trans_linear', 'trans_gold_standard'}

▷ **FMatrix** (output_control) . . . . . hom_mat2d ⇝ *real*
Computed fundamental matrix.

▷ **Kappa** (output_control) . . . . . real ⇝ *real*
Computed radial distortion coefficient.

▷ **Error** (output_control) . . . . . real ⇝ *real*
Root-Mean-Square epipolar distance error.

▷ **X** (output_control) . . . . . real-array ⇝ *real*
X coordinates of the reconstructed points in projective 3D space.

▷ **Y** (output_control) . . . . . real-array ⇝ *real*
Y coordinates of the reconstructed points in projective 3D space.

▷ **Z** (output_control) . . . . . real-array ⇝ *real*
Z coordinates of the reconstructed points in projective 3D space.

▷ **W** (output_control) . . . . . real-array ⇝ *real*
W coordinates of the reconstructed points in projective 3D space.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

match_fundamental_matrix_distortion_ransac

Possible Successors

change_radial_distortion_cam_par, change_radial_distortion_image, change_radial_distortion_points, gen_binocular_proj_rectification

Alternatives

vector_to_fundamental_matrix, vector_to_essential_matrix, vector_to_rel_pose

See also

calibrate_cameras

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

3D Metrology

vector_to_rel_pose (: : Rows1, Cols1, Rows2, Cols2, CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2, CamPar1, CamPar2, Method : RelPose, CovRelPose, Error, X, Y, Z, CovXYZ)

Compute the relative orientation between two cameras given image point correspondences and known camera parameters and reconstruct 3D space points.

For a stereo configuration with known camera parameters the geometric relation between the two images is defined by the relative pose. The operator vector_to_rel_pose computes the relative pose from in general at least six point correspondences in the image pair. RelPose indicates the relative pose of camera 1 with respect to camera 2 (see create_pose for more information about poses and their representations.). This is in accordance with the explicit calibration of a stereo setup using the operator calibrate_cameras. Now, let R, t be the rotation and translation of the relative pose. Then, the essential matrix E is defined as $E = ([t]_\times R)^T$, where $[t]_\times$ denotes the 3×3 skew-symmetric matrix realizing the cross product with the vector t. The pose can be determined from the epipolar constraint:

$$\begin{pmatrix} X_2 \\ Y_2 \\ 1 \end{pmatrix}^T \cdot ([t]_\times R)^T \cdot \begin{pmatrix} X_1 \\ Y_1 \\ 1 \end{pmatrix} = 0 \quad \text{where} \quad [t]_\times = \begin{pmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{pmatrix} .$$

Note, that the essential matrix is a projective entity and thus is defined up to a scaling factor. From this follows that the translation vector of the relative pose can only be determined up to scale too. In fact, the computed translation vector will always be normalized to unit length. As a consequence, a three-dimensional reconstruction of the scene, here in terms of points given by their coordinates (X,Y,Z), can be carried out only up to a single global scaling factor. If the absolute 3D coordinates of the reconstruction are to be achieved the unknown scaling factor can be computed from a gauge, which has to be visible in both images. For example, a simple gauge can be given by any known distance between points in the scene.

The operator vector_to_rel_pose is designed to deal with a camera model that includes lens distortions. This is in contrast to the operator vector_to_essential_matrix, which encompasses only straight line

preserving cameras. The camera parameters are passed by the arguments CamPar1, CamPar2. The 3D direction vectors $(X_1, Y_1, 1)$ and $(X_2, Y_2, 1)$ are calculated from the point coordinates (Rows1,Cols1) and (Rows2,Cols2) by inverting the process of projection (see Calibration). The point correspondences are typically determined by applying the operator match_rel_pose_ransac.

The parameter Method decides whether the relative orientation between the cameras is of a special type and which algorithm is to be applied for its computation. If Method is either *'normalized_dlt'* or *'gold_standard'* the relative orientation is arbitrary. Choosing *'trans_normalized_dlt'* or *'trans_gold_standard'* means that the relative motion between the cameras is a pure translation. The typical application for this special motion case is the scenario of a single fixed camera looking onto a moving conveyor belt. In this case the minimum required number of corresponding points is just two instead of six in the general case.

The relative pose is computed by a linear algorithm if *'normalized_dlt'* or *'trans_normalized_dlt'* is chosen. With *'gold_standard'* or *'trans_gold_standard'* the algorithm gives a statistically optimal result. Here, *'normalized_dlt'* and *'gold_standard'* stand for direct-linear-transformation and gold-standard-algorithm respectively. All methods return the coordinates (X,Y,Z) of the reconstructed 3D points. The optimal methods also return the covariances of the 3D points in CovXYZ. Let n be the number of points then the 3×3 covariance matrices are concatenated and stored in a tuple of length $9n$. Additionally, the optimal methods return the 6×6 covariance matrix of the pose CovRelPose.

If an optimal gold-standard-algorithm is chosen the covariances of the image points (CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2) can be incorporated in the computation. They can be provided for example by the operator points_foerstner. If the point covariances are unknown, which is the default, empty tuples are input. In this case the optimization algorithm internally assumes uniform and equal covariances for all points.

The value Error indicates the overall quality of the optimization process and is the root-mean-square Euclidean distance in pixels between the points and their corresponding epipolar lines.

For the operator vector_to_rel_pose a special configuration of scene points and cameras exists: if all 3D points lie in a single plane and additionally are all closer to one of the two cameras then the solution in the relative pose is not unique but twofold. As a consequence both solutions are computed and returned by the operator. This means that all output parameters are of double length and the values of the second solution are simply concatenated behind the values of the first one.

Parameters

▷ **Rows1** (input_control) . . . . . number-array ⇝ *real* / integer
Input points in image 1 (row coordinate).
Restriction: length(Rows1) >= 6 || length(Rows1) >= 2

▷ **Cols1** (input_control) . . . . . number-array ⇝ *real* / integer
Input points in image 1 (column coordinate).
Restriction: length(Cols1) == length(Rows1)

▷ **Rows2** (input_control) . . . . . number-array ⇝ *real* / integer
Input points in image 2 (row coordinate).
Restriction: length(Rows2) == length(Rows1)

▷ **Cols2** (input_control) . . . . . number-array ⇝ *real* / integer
Input points in image 2 (column coordinate).
Restriction: length(Cols2) == length(Rows1)

▷ **CovRR1** (input_control) . . . . . number-array ⇝ *real* / integer
Row coordinate variance of the points in image 1.
Default: []

▷ **CovRC1** (input_control) . . . . . number-array ⇝ *real* / integer
Covariance of the points in image 1.
Default: []

▷ **CovCC1** (input_control) . . . . . number-array ⇝ *real* / integer
Column coordinate variance of the points in image 1.
Default: []

▷ **CovRR2** (input_control) . . . . . number-array ⇝ *real* / integer
Row coordinate variance of the points in image 2.
Default: []

▷ **CovRC2** (input_control) . . . . . number-array ⇝ *real* / integer
Covariance of the points in image 2.
Default: []

▷ **CovCC2** (input_control) . . . . . number-array ⇝ *real* / integer
Column coordinate variance of the points in image 2.
Default: []

▷ **CamPar1** (input_control) . . . . . campar ⇝ *real* / integer / string
Camera parameters of the 1st camera.

▷ **CamPar2** (input_control) . . . . . campar ⇝ *real* / integer / string
Camera parameters of the 2nd camera.

▷ **Method** (input_control) . . . . . string ⇝ *string*
Algorithm for the computation of the relative pose and for special pose types.
Default: 'normalized_dlt'
List of values: `Method` ∈ {'normalized_dlt', 'gold_standard', 'trans_normalized_dlt', 'trans_gold_standard'}

▷ **RelPose** (output_control) . . . . . pose ⇝ *real* / integer
Computed relative orientation of the cameras (3D pose).

▷ **CovRelPose** (output_control) . . . . . real-array ⇝ *real*
6×6 covariance matrix of the relative camera orientation.

▷ **Error** (output_control) . . . . . real(-array) ⇝ *real*
Root-Mean-Square of the epipolar distance error.

▷ **X** (output_control) . . . . . real-array ⇝ *real*
X coordinates of the reconstructed 3D points.

▷ **Y** (output_control) . . . . . real-array ⇝ *real*
Y coordinates of the reconstructed 3D points.

▷ **Z** (output_control) . . . . . real-array ⇝ *real*
Z coordinates of the reconstructed 3D points.

▷ **CovXYZ** (output_control) . . . . . real-array ⇝ *real*
Covariance matrices of the reconstructed 3D points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`match_rel_pose_ransac`

Possible Successors

`gen_binocular_rectification_map`, `rel_pose_to_fundamental_matrix`

Alternatives

`vector_to_essential_matrix`, `vector_to_fundamental_matrix`, `binocular_calibration`

See also

`camera_calibration`

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

J.Chris McGlone (editor): "Manual of Photogrammetry"; American Society for Photogrammetry and Remote Sensing ; 2004.

Module

3D Metrology

5.2 Depth From Focus

```
depth_from_focus ( MultiFocusImage : Depth, Confidence : Filter,
    Selection :  )
```

Extract depth using multiple focus levels.

The operator depth_from_focus extracts the depth using a focus sequence. The images of the focus sequence have to be passed as a multi channel image (MultiFocusImage). The depth for each pixel will be returned in Depth as the channel number. The parameter Confidence returns a confidence value for each depth estimation: The larger this value, the higher the confidence of the depth estimation is.

depth_from_focus selects the pixels with the best focus of all focus levels. The method used to extract these pixels is specified by the parameters Filter and Selection.

For the parameter Filter, you can choose between the values *'highpass'* and *'bandpass'*. To determine the focus within the image a high- or a bandpass filter can be applied. The larger the filter response, the more in focus is the image at this location. Compared to the highpass filter, the bandpass filter suppresses high frequencies. This is useful in particular in images containing strong noise.

Optionally, you can smooth the filtered image using the mean filter by passing two additional integer values for the mask size in the parameter Filter (e.g., ['*highpass*', *7*, *7*]). This blurs the in-focus region with neighboring pixels and thus allows to bridge small areas with no texture within the image. Note, however, that this smoothing does not suppress noise in the original image, since it is applied only after high- or bandpass filtering.

The parameter Selection determines how the optimum focus level is selected. If you pass the value *'next_maximum'*, the closest focus maximum in the neighborhood is used. In contrast, if you pass the value *'local'*, the focus level is determined based on the focus values of all focus levels of the pixel. With *'next_maximum'*, you typically achieve a slightly smoothed and more robust result.

This additional smoothing is useful if no telecentric lenses are used to take the input images. In this case, the position of an object will slightly shift within the sequence. By adding appropriate smoothing, this effect can be partially compensated.

Attention

If MultiFocusImage contains more than 255 channels (focus levels), Depth is clipped at 255, i.e. depth values higher than 255 are ignored.

If the filter mask for Filter is specified with even values, the routine uses the next larger odd values instead (this way the center of the filter mask is always explicitly determined).

If Selection is set to *'local'* and Filter is set to *'highpass'* or *'bandpass'*, depth_from_focus can be executed on OpenCL devices. If smoothing is enabled, the same restrictions and limitations as for mean_image apply.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **MultiFocusImage** (input_object) ..... multichannel-image(-array) ⇝ *object :* byte
  Multichannel gray image consisting of multiple focus levels.
- ▷ **Depth** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Depth image.
- ▷ **Confidence** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Confidence of depth estimation.
- ▷ **Filter** (input_control) ..... string(-array) ⇝ *string* / integer
  Filter used to find sharp pixels.
  Default: 'highpass'
  Suggested values: Filter ∈ {'highpass', 'bandpass', 3, 5, 7, 9}
- ▷ **Selection** (input_control) ..... string(-array) ⇝ *string*
  Method used to find sharp pixels.
  Default: 'next_maximum'
  List of values: Selection ∈ {'next_maximum', 'local'}

Example

```
compose3(Focus0,Focus1,Focus2,&MultiFocus);
depth_from_focus(MultiFocus,&Depth,&Confidence,'highpass','next_maximum');
mean_image(Depth,&Smooth,15,15);
select_grayvalues_from_channels(MultiChannel,Smooth,SharpImage);
threshold(Confidence,HighConfidence,10,255);
reduce_domain(SharpImage,HighConfidence,ConfidentSharp);
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

compose2, compose3, compose4, add_channels, read_image, read_sequence

Possible Successors

select_grayvalues_from_channels, mean_image, binomial_filter, gauss_filter, threshold

See also

count_channels

Module

3D Metrology

select_grayvalues_from_channels (MultichannelImage, IndexImage : Selected : :)

Selection of gray values of a multi-channel image using an index image.

The operator select_grayvalues_from_channels selects gray values from the different channels of MultichannelImage. The channel number for each pixel is determined from the corresponding pixel value in IndexImage. Note, IndexImage is allowed to have an arbitrary number of channels for reasons of backward compatibility, but only the first channel is considered.

Parameters

▷ **MultichannelImage** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Multi-channel gray value image.

▷ **IndexImage** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
Image, where pixel values are interpreted as channel index.
Number of elements: IndexImage == MultichannelImage || IndexImage == 1

▷ **Selected** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte
Resulting image.

Example

```
compose3(Focus0,Focus1,Focus2,&MultiFocus);
depth_from_focus(MultiFocus,&Depth,&Confidence,'highpass','next_maximum');
mean_image(Depth,&Smooth,15,15);
select_grayvalues_from_channels(MultiChannel,Smooth,SharpImage);
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

depth_from_focus, mean_image

Possible Successors

disp_image

See also

count_channels

Module

Foundation

5.3 Multi-View Stereo

This chapter contains operators for multi-view 3D reconstruction.

Concept of Multi-view 3D Reconstruction

With multi-view 3D reconstruction, it is possible to generate 3D objects using 2D images from multiple cameras. It is possible to reconstruct the complete 3D surface of an object, or single 3D points.

In the following, the steps that are required to reconstruct surfaces and points are described briefly. Note that a well-calibrated camera setup is the main requirement for a precise 3D reconstruction; see Calibration for more details. Additionally, in the HDevelop example `reconstruct_surface_mixed_camera_types.hdev`, a typical calibration workflow (from the calibration data model via the camera setup model to the stereo model) is performed.

Generate stereo model: First, create the stereo model using

- create_stereo_model.

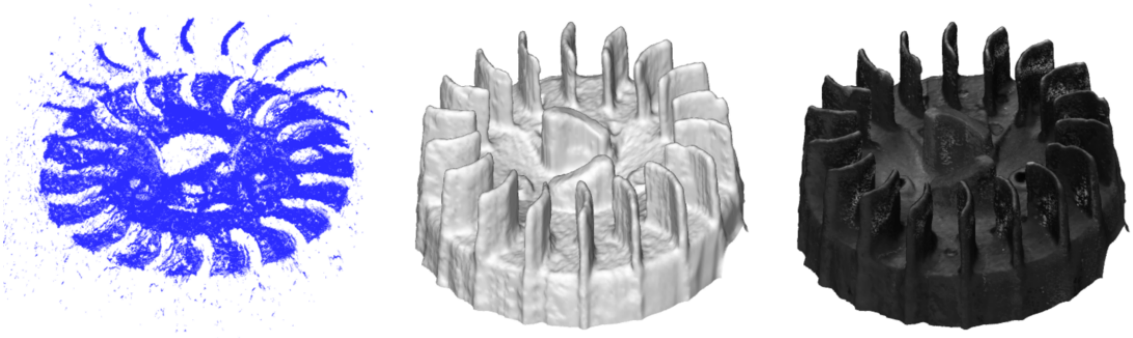
If you want to reconstruct 3D points, choose the `Method` *'points_3d'*.

3D point reconstruction with *'points_3d'*.

For the reconstruction of surfaces, the methods *'surface_pairwise'* and *'surface_fusion'* are available. For detailed information on these two methods, have a look at the reference manual entry of reconstruct_surface_stereo.

(1) (2) (3)

These three 2D images are used for the surface reconstruction as seen in the images below.



(1) (2) (3)

(1) Surface reconstruction with *'surface_pairwise'*. (2) Surface reconstruction with *'surface_fusion'*. (3) Surface reconstruction with *'surface_fusion'*, where the color information is extracted from the used 2D images. Have a look at the HDevelop example `reconstruct_surface_mixed_camera_types.hdev` to see the 3D reconstruction process.

Set the image pairs (only for surface reconstruction): For the reconstruction of 3D surfaces, multiple binocular stereo reconstructions are performed, and then combined. For the binocular reconstruction, image pairs have to be specified. For example, for the three images shown above, the image pairs might be `[0,1]` and `[1,2]`. The image pairs have to be specified using

- `set_stereo_model_image_pairs`,

and query the image pairs with

- `get_stereo_model_image_pairs`.

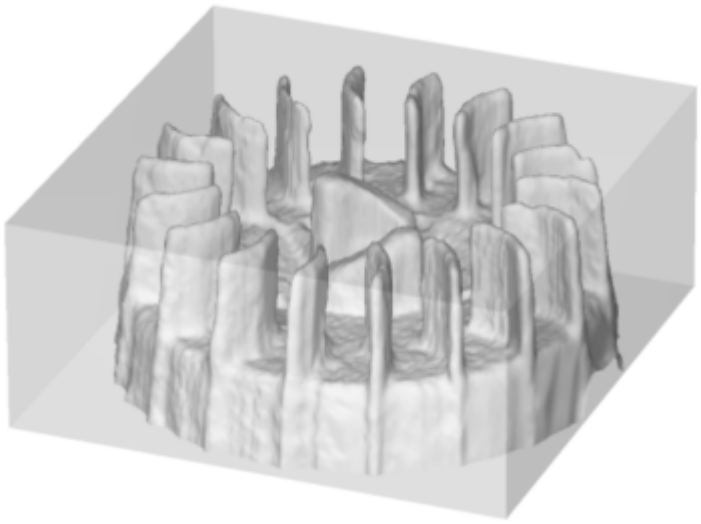
For more information, see `reconstruct_surface_stereo` as well as the above-mentioned operators.

Modify the stereo model parameters: With

- `set_stereo_model_param`,

you can optimize the settings of the 3D reconstruction for your setup.

When reconstructing surfaces, it is highly recommended to limit the 3D reconstruction using a bounding box which is as tight as possible around the object that is to be reconstructed.



The bounding box, which is set with `set_stereo_model_param`, restricts the area where the object is reconstructed, and thus can be used to reduce the runtime greatly.

When using the *'surface_fusion'* `Method` in `create_stereo_model`, it is recommended to first optimize the parameters of the *'surface_pairwise'* `Method`, since it is used as a basis. For more details on the parameters, see the examples `reconstruct_surface_stereo_pairwise_workflow.hdev` and `reconstruct_surface_stereo_fusion_workflow.hdev`.

You can query the set parameters with

- `get_stereo_model_param`.

Perform the 3D reconstruction: Then, to perform the actual reconstruction, use

- `reconstruct_points_stereo` or
- `reconstruct_surface_stereo`.

Get intermediate results (only for surface reconstruction): Note that to query these intermediate results, you must enable the *'persistence'* mode for the stereo model with `set_stereo_model_param` before performing the reconstruction.

With

- `get_stereo_model_object`,

you can access and inspect intermediate results of a surface reconstruction performed with `reconstruct_surface_stereo`. These images can be used for troubleshooting the reconstruction process.

With

- `get_stereo_model_object_model_3d`,

you can get the 3D object model that was reconstructed with `reconstruct_surface_stereo` as an intermediate result using the `Method` *'surface_fusion'*.

`clear_stereo_model` (: : StereoModelID :)

Free the memory of a stereo model.

The operator `clear_stereo_model` frees the memory of the stereo model `StereoModelID` that was created by `create_stereo_model`. After calling `clear_stereo_model`, the model can no longer be used. The handle `StereoModelID` becomes invalid.

Parameters

▷ **`StereoModelID`** (input_control) ..... stereo_model ⇝ *handle*
Handle of the stereo model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- StereoModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

3D Metrology

create_stereo_model (: : CameraSetupModelID, Method, GenParamName, GenParamValue : StereoModelID)

Create a HALCON stereo model.

The operator create_stereo_model creates a HALCON stereo model and returns a handle to it in StereoModelID. The model provides functionality for reconstructing either 3D points or surfaces from a calibrated multi-view stereo camera setup specified in CameraSetupModelID (refer to Calibration / Multi-View for further details on calibration of multiple cameras).

For Method='*points_3d*', a stereo model is created that, after being configured, can be passed to reconstruct_points_stereo. The latter reconstructs 3D points by intersecting lines of sight from point correspondences, extracted from multiple calibrated images (see reconstruct_points_stereo for more details).

For Method='*surface_pairwise*' or Method='*surface_fusion*', a stereo model is created that, after being configured, can be passed to reconstruct_surface_stereo. The latter obtains disparity images from preselected image pairs in a calibrated multi-view stereo setup and fuses the collected 3D information in a single surface reconstruction (see reconstruct_surface_stereo for more details).

The parameters GenParamName and GenParamValue can be used to set general model parameters. Alternatively, these parameters can be modified with the operator set_stereo_model_param before the corresponding reconstruction operator is called (see set_stereo_model_param for more details on the available model parameters).

Parameters

▷ **CameraSetupModelID** (input_control) . . . . . camera_setup_model ⇝ *handle*
Handle to the camera setup model.

▷ **Method** (input_control) . . . . . string ⇝ *string*
Reconstruction method.
Default: 'surface_pairwise'
List of values: Method ∈ {'surface_pairwise', 'surface_fusion', 'points_3d'}

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the model parameter to be set.
Default: []
List of values: GenParamName ∈ {'bounding_box', 'persistence', 'sub_sampling_step', 'rectif_interpolation', 'rectif_method', 'disparity_method', 'binocular_method', 'binocular_num_levels', 'binocular_mask_width', 'binocular_mask_height', 'binocular_texture_thresh', 'binocular_score_thresh', 'binocular_filter', 'binocular_sub_disparity', 'binocular_mg_gray_constancy', 'binocular_mg_gradient_constancy', 'binocular_mg_smoothness', 'binocular_mg_initial_guess', 'binocular_mg_default_parameters', 'binocular_mg_solver', 'binocular_mg_cycle_type', 'binocular_mg_pre_relax', 'binocular_mg_post_relax', 'binocular_mg_initial_level', 'binocular_mg_iterations', 'binocular_mg_pyramid_factor', 'binocular_ms_surface_smoothing', 'binocular_ms_edge_smoothing', 'binocular_ms_consistency_check', 'binocular_ms_similarity_measure', 'binocular_ms_sub_disparity', 'point_meshing', 'resolution', 'surface_tolerance', 'min_thickness', 'smoothing', 'color', 'color_invisible', 'min_disparity', 'max_disparity'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Value of the model parameter to be set.
Default: []
Suggested values: GenParamValue ∈ {-1, -2, -5, 0, 0.3, 0.5, 0.9, 1, 2, 3, 'census_dense', 'census_sparse', 'binocular', 'ncc', 'none', 'sad', 'ssd', 'bilinear', 'viewing_direction', 'geometric', 'false', 'very_accurate', 'accurate', 'fast_accurate', 'fast', 'v', 'w', 'none', 'gauss_seidel', 'multigrid', 'true', 'poisson', 'isosurface', 'interpolation', 'left_right_check', 'full_multigrid', 'binocular_mg', 'binocular_ms', 'smallest_distance', 'mean_by_distance', 'line_of_sight', 'mean_by_line_of_sight', 'median'}

▷ **StereoModelID** (output_control) . . . . . stereo_model ⇝ *handle*
Handle of the stereo model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_stereo_model_param, set_stereo_model_image_pairs, reconstruct_surface_stereo, reconstruct_points_stereo

Module

3D Metrology

get_stereo_model_image_pairs (: : StereoModelID : From, To)

Return the list of image pairs set in a stereo model.

The operator get_stereo_model_image_pairs returns the list of image pairs for the stereo model StereoModelID. The camera indices of the *from* and *to* cameras in the pairs are returned in the parameters From and To, respectively (the terms "from" and "to" signal that during reconstruction the disparity "from" one image "to" the other image of the pair is computed). The indices identify cameras from the camera setup model assigned to the stereo model (see create_stereo_model).

The list of image pairs can be set with the operator set_stereo_model_image_pairs.

Parameters

▷ **StereoModelID** (input_control) ..... stereo_model ⇝ *handle*
Handle of the stereo model.

▷ **From** (output_control) ..... integer-array ⇝ *integer*
Camera indices for the *from* cameras in the image pairs.

▷ **To** (output_control) ..... integer-array ⇝ *integer*
Camera indices for the *to* cameras in the image pairs.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_stereo_model_image_pairs

Possible Successors

reconstruct_surface_stereo

See also

set_stereo_model_image_pairs

Module

3D Metrology

get_stereo_model_object (: Object : StereoModelID, PairIndex, ObjectName :)

Get intermediate iconic results of a stereo reconstruction.

With the operator get_stereo_model_object you can access and inspect intermediate iconic results of a surface reconstruction performed with reconstruct_surface_stereo for the stereo model StereoModelID. In particular, this is useful for troubleshooting the reconstruction process. Note that to collect the iconic results, you must enable the *'persistence'* mode for the stereo model (see set_stereo_model_param) *before* performing the reconstruction. Iconic results are then associated with each image pair that was processed during the reconstruction (see get_stereo_model_image_pairs).

You select the image pair of interest by specifying the corresponding camera indices `[From, To]` in `PairIndex`. By setting one of the following values in `ObjectName`, the corresponding iconic objects are then returned in `Object`:

'from_image_rect', *'to_image_rect'*: Rectified image corresponding to the *from* and *to* camera, respectively. Both images can be used to inspect the quality of the internal binocular stereo image rectification.

'disparity_image': Disparity image for this pair. The quality of the disparity image has a direct impact on the final surface reconstruction.

'score_image': Score image assigned to the disparity image for this pair.

A mismatch between the rectified images, i.e., features appearing in different rows in the two images, or errors in the disparity or the score image have direct impact on the quality of the final surface reconstruction. Therefore, we recommend to correct any detected imperfections by adjusting the stereo model parameters (see `set_stereo_model_param`), in particular those which control the internal usage of `gen_binocular_rectification_map` and `binocular_disparity` (see `set_stereo_model_image_pairs` and `reconstruct_surface_stereo` for further details).

Parameters

▷ **`Object`** (output_object) ..... object(-array) ⇝ *object*
Iconic result.

▷ **`StereoModelID`** (input_control) ..... stereo_model ⇝ *handle*
Handle of the stereo model.

▷ **`PairIndex`** (input_control) ..... number(-array) ⇝ *integer* / string / real
Camera indices of the pair (`[From, To]`).
Suggested values: `PairIndex` ∈ {0, 1, 2}

▷ **`ObjectName`** (input_control) ..... string ⇝ *string*
Name of the iconic result to be returned.
Suggested values: `ObjectName` ∈ {'from_image_rect', 'to_image_rect', 'disparity_image', 'score_image'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`reconstruct_surface_stereo`

Module

3D Metrology

`get_stereo_model_object_model_3d` (: : `StereoModelID`, `GenParamName` : `ObjectModel3D`)

Get intermediate 3D object model of a stereo reconstruction

With the operator `get_stereo_model_object_model_3d` it is possible to get a 3D object model `ObjectModel3D` that was reconstructed with `reconstruct_surface_stereo` as an intermediate result using the method *'surface_fusion'*. The returned object model is equal to the result of `reconstruct_surface_stereo` using method *'surface_pairwise'*.

For this, a call to `get_stereo_model_object_model_3d` has to be performed using the value *'m3d_pairwise'* for the parameter `GenParamName`. It should be noted that the model can only be queried if the *'persistence'* mode for the stereo model (see `set_stereo_model_param`) is enabled *before* performing the reconstruction. Furthermore the object model can only be queried if the stereo model has been created using the method *'surface_fusion'*. Otherwise, an error is returned. If no object model has been created, the operator returns -1.

Parameters

▷ **StereoModelID** (input_control) . . . . . stereo_model ⇝ *handle*
Handle of the stereo model.

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Names of the model parameters.
List of values: GenParamName ∈ {'m3d_pairwise'}

▷ **ObjectModel3D** (output_control) . . . . . object_model_3d ⇝ *handle*
Values of the model parameters.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

reconstruct_surface_stereo, set_stereo_model_param

See also

set_stereo_model_param

Module

3D Metrology

get_stereo_model_param (: : StereoModelID, GenParamName : GenParamValue)

Get stereo model parameters.

The operator get_stereo_model_param can be used to inspect diverse parameters of the stereo model StereoModelID by specifying their names in GenParamName and getting their values in GenParamValue. Two types of parameters can be inspected with this operator - general and specific for surface reconstruction. Note that no specific parameters are provided for 3D point stereo reconstruction.

All parameters that can be set with set_stereo_model_param can also be queried with get_stereo_model_param - for a description see the former operator. In contrast, the following parameters are set by other operators and cannot be modified afterwards.

General parameters

'type': Type of the stereo model (currently either *'surface_pairwise'*, *'surface_fusion'* or *'points_3d'*).

'camera_setup_model': Handle to a *copy* of the camera setup model set in the stereo model. Changing properties of the copy does not affect the camera setup model stored in the stereo model.

'from_cam_param_rect N', *'to_cam_param_rect N'*: Camera parameters of the rectified *from*- and *to*-cameras of camera pair N. See set_stereo_model_image_pairs for more information about camera pairs.

'from_cam_pose_rect N', *'to_cam_pose_rect N'*: Point transformation from the rectified *from*- and *to*-cameras of camera pair N to the respective unrectified camera. See set_stereo_model_image_pairs for more information about camera pairs.

'rel_pose_rect N': Point transformation from the rectified *to*-camera to the rectified *from*-camera. See set_stereo_model_image_pairs for more information about camera pairs.

The parameters *'type'* and *'camera_setup_model'* are set when creating the stereo model with create_stereo_model. For *'from_cam_param_rect N'*, *'to_cam_param_rect N'*, *'from_cam_pose_rect N'*, *'to_cam_pose_rect N'*, and *'rel_pose_rect N'*, note that these parameters are only available *after* setting the image pairs (see set_stereo_model_image_pairs).

A note on tuple-valued model parameters

Most of the stereo model parameters are single-valued. Thus, you can provide a list (i.e., tuple) of parameter names and get a list (tuple) of values that has the same length as the output tuple. In contrast, when querying a tuple-valued parameter, a tuple of values is returned. When querying such a parameter together with other parameters, the value-to-parameter-name correspondence is not obvious anymore. Thus, tuple-valued parameters like *'bounding_box'*, *'min_disparity'* or *'max_disparity'* should always be queried in a separate call to `get_stereo_model_param`.

Parameters

▷ **StereoModelID** (input_control) . . . . . stereo_model ⇝ *handle*
Handle of the stereo model.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the parameters to be set.
List of values: `GenParamName` ∈ {'type', 'camera_setup_model', 'bounding_box', 'persistence', 'sub_sampling_step', 'rectif_interpolation', 'rectif_sub_sampling', 'rectif_method', 'disparity_method', 'binocular_method', 'binocular_num_levels', 'binocular_mask_width', 'binocular_mask_height', 'binocular_texture_thresh', 'binocular_score_thresh', 'binocular_filter', 'binocular_sub_disparity', 'binocular_mg_gray_constancy', 'binocular_mg_gradient_constancy', 'binocular_mg_smoothness', 'binocular_mg_initial_guess', 'binocular_mg_solver', 'binocular_mg_cycle_type', 'binocular_mg_pre_relax', 'binocular_mg_post_relax', 'binocular_mg_initial_level', 'binocular_mg_iterations', 'binocular_mg_pyramid_factor', 'binocular_ms_surface_smoothing', 'binocular_ms_edge_smoothing', 'binocular_ms_consistency_check', 'binocular_ms_similarity_measure', 'binocular_ms_sub_disparity', 'min_disparity', 'max_disparity', 'point_meshing', 'poisson_depth', 'poisson_solver_divide', 'poisson_samples_per_node', 'resolution', 'surface_tolerance', 'min_thickness', 'smoothing', 'color', 'color_invisible', 'from_cam_param_rect', 'to_cam_param_rect', 'from_cam_pose_rect', 'to_cam_pose_rect', 'rel_pose_rect'}

▷ **GenParamValue** (output_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of the parameters to be set.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_stereo_model`, `set_stereo_model_param`

Possible Successors

`reconstruct_surface_stereo`, `reconstruct_points_stereo`

See also

`set_stereo_model_param`

Module

3D Metrology

reconstruct_points_stereo (: : StereoModelID, Row, Column, CovIP, CameraIdx, PointIdx : X, Y, Z, CovWP, PointIdxOut)

Reconstruct 3D points from calibrated multi-view stereo images.

The operator `reconstruct_points_stereo` reconstructs 3D points from point correspondences found in the images of a calibrated multi-view stereo setup. The calibration information for the images is provided in the camera setup model that is associated with the stereo model `StereoModelID` during its creation (see `create_stereo_model`). Note that the stereo model type must be *'points_3d'*, otherwise the operator will return an error.

The point correspondences must be passed in the parameters `Row`, `Column`, `CameraIdx`, and `PointIdx` in form of tuples of the same length. Each set `(Row[I],Column[I],CameraIdx[I],PointIdx[I])` represents the image coordinates (`Row`, `Column`) of the 3D point (`PointIdx`) in the image of a certain camera (`CameraIdx`).

The reconstructed 3D point coordinates are returned in the tuples `X`, `Y`, and `Z`, relative to the coordinate system of the camera setup model (see `create_camera_setup_model`). The tuple `PointIdxOut` contains the corresponding point indices.

The reconstruction algorithm works as follows: First, it identifies point correspondences for a given 3D point by collecting all sets with the same `PointIdx`. Then, it uses the `Row`, `Column`, and `CameraIdx` information from the collected sets to project lines of sight from each camera through the corresponding image point `[Row,Column]`. If there are at least 2 lines of sight for the point `PointIdx`, they are intersected and the result is stored as the set `(X[J],Y[J],Z[J],PointIdxOut[J])`. The intersection is performed with a least-squares algorithm, without taking into account potentially invalid lines of sight (e.g., if an image point was falsely specified as corresponding to a certain 3D point).

To compute the covariance matrices for the reconstructed 3D points, statistical information about the extracted image coordinates, i.e., the covariance matrices of the image points (see , e.g., `points_foerstner`) are needed as input and must be passed in the parameter `CovIP`. Otherwise, if no covariance matrices for the 3D points are needed or no covariance matrices for the image points are available, an empty tuple can be passed in `CovIP`. Then no covariance matrix for the reconstructed 3D points is computed.

The covariance matrix of an image point is:

$$\texttt{CovIP} = \begin{bmatrix} (sigma\_r)^2 & sigma\_rc \\ sigma\_rc & (sigma\_c)^2 \end{bmatrix}$$

The covariance matrices are symmetric `2x2` matrices, whose entries in the main diagonal represent the variances of the image point in row-direction and column-direction, respectively. For each image point, a covariance matrix must be passed in `CovIP` in form of a tuple with 4 elements:

$$[(sigma\_r)^2, sigma\_rc, sigma\_rc, (sigma\_c)^2].$$

Thus, `|CovIP|=4*|Row|` and `CovIP[I*4:I*4+3]` is the covariance matrix for the `I`-th image point.

The computed covariance matrix for a successfully reconstructed 3D point is represented by a symmetric `3x3` matrix:

$$\texttt{CovWP} = \begin{bmatrix} (sigma\_x)^2 & sigma\_xy & sigma\_xz \\ sigma\_yx & (sigma\_y)^2 & sigma\_yz \\ sigma\_zx & sigma\_zy & (sigma\_z)^2 \end{bmatrix}$$

The diagonal entries represent the variances of the reconstructed 3D point in x-, y-, and z-direction. The computed matrices are returned in the parameter `CovWP` in form of tuples each with 9 elements:

$$[(sigma\_x)^2, sigma\_xy, sigma\_xz, sigma\_yx, (sigma\_y)^2, sigma\_yz, sigma\_zx, sigma\_zy, (sigma\_z)^2].$$

Thus, `|CovWP|=9*|X|` and `CovWP[J*9:J*9+8]` is the covariance matrix for the `J`-th 3D point. Note that if the camera setup associated with the stereo model contains the covariance matrices for the camera parameters, these covariance matrices are considered in the computation of `CovWP` too.

If the stereo model has a valid bounding box set (see `set_stereo_model_param`), the resulting points are clipped to this bounding box, i.e., points outside it are not returned. If the bounding box associated with the stereo model is invalid, it is ignored and all points that could be reconstructed are returned.

Parameters

- ▷ **StereoModelID** (input_control) ..... stereo_model ⇝ *handle*
  Handle of the stereo model.
- ▷ **Row** (input_control) ..... number(-array) ⇝ *real* / integer
  Row coordinates of the detected points.
- ▷ **Column** (input_control) ..... number(-array) ⇝ *real* / integer
  Column coordinates of the detected points.
- ▷ **CovIP** (input_control) ..... number-array ⇝ *real* / integer
  Covariance matrices of the detected points.
  Default: []

▷ **CameraIdx** (input_control) ..... number ⇝ *integer*
Indices of the observing cameras.
Suggested values: CameraIdx ∈ {0, 1, 2}

▷ **PointIdx** (input_control) ..... number ⇝ *integer*
Indices of the observed world points.
Suggested values: PointIdx ∈ {0, 1, 2}

▷ **X** (output_control) ..... real(-array) ⇝ *real*
X coordinates of the reconstructed 3D points.

▷ **Y** (output_control) ..... number(-array) ⇝ *real*
Y coordinates of the reconstructed 3D points.

▷ **Z** (output_control) ..... number(-array) ⇝ *real*
Z coordinates of the reconstructed 3D points.

▷ **CovWP** (output_control) ..... number(-array) ⇝ *real*
Covariance matrices of the reconstructed 3D points.

▷ **PointIdxOut** (output_control) ..... number(-array) ⇝ *integer*
Indices of the reconstructed 3D points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

reconstruct_surface_stereo, intersect_lines_of_sight

Module

3D Metrology

reconstruct_surface_stereo (
Images : : StereoModelID : ObjectModel3D)

Reconstruct surface from calibrated multi-view stereo images.

The operator reconstruct_surface_stereo reconstructs a surface from multiple Images, acquired with a calibrated multi-view setup associated with a stereo model StereoModelID. The reconstructed surface is stored in the handle ObjectModel3D.

Preparation and requirements

A summary of the preparation of a stereo model for surface reconstruction:

1. Obtain calibrated camera setup model (use calibrate_cameras or create_camera_setup_model) and configure it.
2. Create a stereo model with create_stereo_model by selecting Method=*'surface_pairwise'* or *'surface_fusion'* (see 'Reconstruction algorithm').
3. Configure the rectification parameters with set_stereo_model_param and afterwards set the image pairs with set_stereo_model_image_pairs.
4. Configure the bounding box for the system with set_stereo_model_param (GenParamName=*'bounding_box'*).
5. Configure parameters of pairwise reconstruction with set_stereo_model_param.
6. For models with Method=*'surface_fusion'* configure parameters of the fusion algorithm with set_stereo_model_param.
7. Acquire images with the calibrated cameras setup and collect them in an image array Images.
8. Perform surface reconstruction with reconstruct_surface_stereo.
9. Query and analyze intermediate results with get_stereo_model_object and get_stereo_model_object_model_3d.

10. Readjust the parameters of the stereo model to improve the results with respect to quality and runtime with set_stereo_model_param.

A **camera setup model** is associated with the stereo model StereoModelID upon its creation with create_stereo_model. The camera setup must contain *calibrated information* about the cameras, with which the images in the image array Images were acquired: the I-th image from the array corresponds to the camera with index I-1 from the camera setup; the number of images in the array must be the same as the number of cameras in the camera setup. The Images must represent a static scene or they must be taken simultaneously, otherwise, the reconstruction of the surface might be impossible.

A well-calibrated camera setup is the main requirement for a precise surface reconstruction. Therefore, special attention should be paid to obtaining a precise calibration of the cameras in the multi-view stereo setup used. HALCON provides calibration of a multi-view setup with the operator calibrate_cameras. The resulting calibrated camera setup can be accessed with a successive call to get_calib_data. Alternatively, for camera setups with known parameters a calibrated camera setup can be created with create_camera_setup_model.

The proper selection of **image pairs** (see set_stereo_model_image_pairs) has an important role for the general quality of the surface reconstruction. On the one hand, camera pairs with a small base line (small distance between the camera centers) are better suited for the binocular stereo disparity algorithms. On the other hand, in order to derive more accurate depth information of the scene, pairs with a long base line should be preferred. Camera pairs should provide different points of view, such that if one pair does not *see* a certain area of the surface, it is covered by another pair. Please note that the number of pairs linearly affects the runtime of the pairwise reconstruction. Therefore, use "as many as needed and just as few as possible" image pairs in order to handle the trade-off between completeness of the surface reconstruction and reconstruction runtime.

A **bounding box** is associated with the stereo model StereoModelID. For the surface stereo reconstruction, it is required that the bounding box is valid (see set_stereo_model_param for further details). The reconstruction algorithm needs the bounding box for three reasons:

- First, if MinDisparity and MaxDisparity were not set manually using the operators create_stereo_model or set_stereo_model_param, it uses the projection of the bounding box into both images of each image pair in order to estimate the values for MinDisparity and MaxDisparity, which in turn are used in the internal call to binocular_disparity and binocular_disparity_ms. In the case of using binocular_disparity_mg as disparity method, suitable values for the parameters InitialGuess and *'initial_level'* are derived from the above-mentioned parameters. However, the automatic estimation for this method is only used if called with default values for the two parameters. Otherwise, the values as set by the user with set_stereo_model_param are used.
- Secondly, the default parameters for the fusion of pairwise reconstructions are calculated based on the bounding box. They are reset in case the bounding box is changed. The bounding box should be tight around the volume of interest. Else, the runtime will increase unnecessarily and drastically.
- Thirdly, the surface fragments lying outside the bounding box are clipped and are not returned in ObjectModel3D. A too large bounding box results in a large difference between MinDisparity and MaxDisparity and this usually slows down the execution of binocular_disparity, binocular_disparity_ms or binocular_disparity_mg and therefore reconstruct_surface_stereo. A too small bounding box might result in clipping valid surface areas.

Note that the method *'surface_fusion'* will try to produce a closed surface. If the object is only observed and reconstructed from one side, the far end of the bounding box usually determines where the object is cut off.

Setting parameters of pairwise reconstruction before setting parameters of fusion is essential since the pairwise reconstruction of the object is input for the fusion algorithm. For a description of parameters, see set_stereo_model_param. The choice of *'disparity_method'* has a major influence. The objects in the scene should expose certain surface properties in order to make the scene suitable for the dense surface reconstruction. First, the surface reflectance should exhibit Lambertian properties as closely as possible (i.e., light falling on the surface is scattered such that its apparent brightness is the same regardless of the angle of view). Secondly, the surface should exhibit enough texture, but no repeating patterns.

get_stereo_model_object can be used to view **intermediate results**, in particular rectified, disparity and score images. get_stereo_model_object_model_3d can be used to view the result of pairwise reconstruction for models with Method='surface_fusion'. See the paragraph "Troubleshooting for the configuration of a stereo model" on how to use the obtained results.

Reconstruction algorithm

The operator reconstruct_surface_stereo performs multiple binocular stereo reconstructions and subsequently combines the results. The image pairs of this pairwise reconstruction are specified in StereoModelID as pairs of cameras of an associated calibrated multi-view setup.

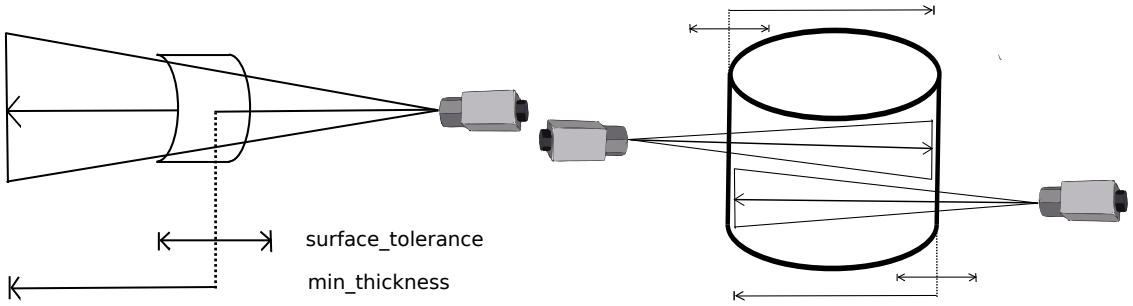
For each image pair, the images are rectified before internally one of the operators binocular_disparity, binocular_disparity_mg or binocular_disparity_ms is called. The disparity information is then converted to points in the coordinate system of the *from*-camera by an internal call of disparity_image_to_xyz. In the next step, the points are transformed into the common coordinate system that is specified in the camera setup model associated with StereoModelID and stored in a common point cloud together with the points extracted from other image pairs.

'surface_pairwise' If the stereo model is of type *'surface_pairwise'* (compare create_stereo_model), the point cloud obtained as described above is directly returned in ObjectModel3D. For each point, the normal vector is calculated by fitting a plane through the neighboring 3D points. In contrast to surface_normals_object_model_3d, the neighboring points are not determined in 3D but simply in 2D by using the neighboring points in the X, Y, and Z images. The normal vector of each 3D point is then set to the normal vector of the respective plane. Additionally, the score of the calculated disparity is attached to every reconstructed 3D point and stored as an extended attribute. Furthermore, transformed coordinate images can be sub-sampled. If only one image pair is processed and no point meshing is enabled, reconstruct_surface_stereo stores a 'xyz_mapping' attribute in ObjectModel3D, which reveals the mapping of the reconstructed 3D points to coordinates of the first image of the pair. This attribute is required by operators like segment_object_model_3d or object_model_3d_to_xyz (with Type=*'from_xyz_map'*). In contrast to the single pair case, if two or more image pairs are processed, reconstruct_surface_stereo does not store the 'xyz_mapping' attribute since single reconstructed points would originate from different image pairs. The presence of the attribute in the output object model can be verified by calling get_object_model_3d_params with GenParamName=*'has_xyz_mapping'*.

The so-obtained point cloud can be additionally meshed in a post-processing step. The object model returned in ObjectModel3D then contains the description of the mesh. The used meshing algorithm depends on the type of the stereo model. For a stereo model of type *'surface_pairwise'*, only a Poisson solver is supported which can be activated by setting the parameter *'point_meshing'* to *'poisson'*. It creates a water-tight mesh, therefore surface regions with missing data are covered by an interpolated mesh.

'surface_fusion' If the stereo model is of type *'surface_fusion'*, the point cloud obtained as described above is processed further. The goal is to obtain a preferably smooth surface while keeping form fidelity. To this end, the bounding box is sampled and each sample point is assigned a distance to a so-called isosurface (consisting of points with distance 0). The final distance values (and thus the isosurface) are obtained by minimizing an error function based on the points resulting from pairwise reconstruction. This leads to a fusion of the reconstructed point clouds of all camera pairs (see the second paper in References below).

The calculation of the isosurface can be influenced by set_stereo_model_param with the parameters *'resolution'*, *'surface_tolerance'*, *'min_thickness'* and *'smoothing'*. The distance between sample points in the bounding box (in each coordinate direction) can be set by the parameter *'resolution'*. The parameter *'smoothing'* regulates the 'jumpiness' of the distance function by weighting the two terms in the error function: Fidelity to the initial point clouds obtained by pairwise reconstruction on the one hand, total variation of the distance function on the other hand. Note that the actual value of *'smoothing'* for a given data set to be visually pleasing has to be found by trial and error. Too small values lead to integrating many outliers into the surface even if the object surface then exhibits many jumps. Too large values lead to loss of fidelity towards the point cloud of pairwise reconstruction. Fidelity to the initial surfaces obtained by pairwise reconstruction is not maintained in the entire bounding box, but only in cones of sight of cameras to the initial surface. A sample point in such a cone is considered surely outside of the object (in front of the surface) or surely inside the object (behind the surface) with respect to the given camera if its distance to the initial surface exceeds a given value which can be set by the parameter *'surface_tolerance'*. The length of considered cones behind the initial surface can roughly be set by the parameter *'min_thickness'* (see set_stereo_model_param for more details). *'min_thickness'* always has to be larger than or equal to *'surface_tolerance'*.



The parameters *'surface_tolerance'* and *'min_thickness'* regulate the fidelity to the initial surface obtained by pairwise reconstruction. Points in a cone of sight of a camera are considered surely outside of the object (in front of the surface) or surely inside the object (behind the surface) with respect to the given camera if their distance to the initial surface exceeds *'surface_tolerance'*. Points behind the surface (viewed from the given camera) are only considered to lie inside the object if their distance to the initial surface does not exceed *'min_thickness'*.

Each 3D point of the object model returned in `ObjectModel3D` is extracted from the isosurface where the distance function equals zero. Its normal vector is calculated from the gradient of the distance function. While the method *'surface_fusion'* requires the setting of more parameters than simple pairwise reconstruction, post-processing of the obtained point cloud representing the object surface will probably get a lot simpler. In particular, suppression of outliers, smoothing, equidistant sub-sampling and hole filling can be handled nicely and often in high quality by this method. The same can be said about the possible internal meshing of the output surface, see the next paragraph. Note that the algorithm will try to produce a closed surface. If the object is only observed and reconstructed from one side, the far end of the bounding box usually determines where the object is cut off. The method *'surface_fusion'* may take considerably longer than simple pairwise reconstruction, depending mainly on the parameter *'resolution'*.

Additionally, the so-obtained point cloud can be meshed in a post-processing step. The object model returned in `ObjectModel3D` then contains the description of the mesh. For a stereo model of type *'surface_fusion'*, the algorithm 'marching tetrahedra' is used which can be activated by setting the parameter *'point_meshing'* to *'isosurface'*. The wanted meshed surface is extracted as the isosurface where the distance function equals zero. Note that there are more points in `ObjectModel3D` if meshing of the isosurface is enabled even if the used *'resolution'* is the same.

Coloring the 3D object model

It is possible to provide color information for 3D object models that have been reconstructed with `reconstruct_surface_stereo` from the input images. The computation of the color depends on the chosen method set with `set_stereo_model_param` (see explanation in the list there). Each 3D point is assigned a color value consisting of a red, green and blue channel which are stored as attributes named *'red'*, *'green'* and *'blue'* in the output 3D object model `ObjectModel3D`. These attributes can for example be used in the procedure `visualize_object_model_3d` with `GenParamName` = *'red_channel_attrib'*, *'green_channel_attrib'* and *'blue_channel_attrib'*. They can also be queried with `get_object_model_3d_params` or be processed with `select_points_object_model_3d` or other operators that use extended attributes. If the reconstruction has been performed using gray value images, the color value for the three channels is identical. If multi-channel images are used, the reconstruction is performed using the first channel only. The remaining channels are solely used for the calculation of the color values.

If stereo models of type *'surface_fusion'* are used, the reconstruction will contain points without a direct correspondence to points in the images. These points are not seen by any of the cameras of the stereo system and are therefore "invisible". A color value for these points is derived by assigning the value of the nearest visible neighbor. Normally, the nearest neighbor search is not very time-consuming and can remain active. However, it may happen that the value for the parameter *'resolution'* is considerably finer than the available image resolution. In this case, many invisible 3D points are reconstructed making the nearest neighbor search very time consuming. In order to avoid an increased runtime, it is recommended to either adapt the value of *'resolution'* or to switch off the calculation for invisible points. This can be done by calling `set_stereo_model_param` with `GenParamName`=*'color_invisible'* and `GenParamValue`= *'false'*. In this case, invisible points are assigned 255 as gray value.

Troubleshooting for the configuration of a stereo model

The proper configuration of a stereo model is not always easy. Please follow the workflow above. If the reconstruction results are not satisfactory, please consult the following hints and ideas:

Run in persistence mode If you enable the *'persistence'* mode of stereo model (call set_stereo_model_param with GenParamName=*'persistence'*) a successive call to reconstruct_surface_stereo will store intermediate iconic results, which provide additional information. They can be accessed by get_stereo_model_object_model_3d and get_stereo_model_object.

Check the quality of the calibration

- If the camera setup was obtained by calibrate_cameras, it stores some quality information about the camera calibration in form of standard deviations of the camera internal parameters. This information is then carried in the camera setup model associated with the stereo model. It can be queried by first calling get_stereo_model_param with GenParamName=*'camera_setup_model'* and then inspecting the camera parameter standard deviations by calling get_camera_setup_param with GenParamName=*'params_deviations'*. Unusually big standard deviation values might indicate a bad camera calibration.
- After setting the stereo model *'persistence'* mode, we recommend inspecting the rectified images for each image pair. The rectified images are returned by get_stereo_model_object with a camera index pair [From, To] specifying the pair of interest in the parameter PairIndex and the values *'from_image_rect'* and *'to_image_rect'* in ObjectName, respectively. If the images are properly rectified, all corresponding image features must appear in the same row in both rectified images. A discrepancy of several rows is a serious indication for a bad camera calibration.

Inspect the used bounding box Make sure that the bounding box is tight around the volume of interest. If the parameters *'min_disparity'* and *'max_disparity'* are not set manually by using create_stereo_model or set_stereo_model_param, the algorithm uses the projection of the bounding box into both images of each image pair in order to estimate the values for MinDisparity and MaxDisparity, which in turn are used in the internal call to binocular_disparity and binocular_disparity_ms. These values can be queried using get_stereo_model_param and if needed, can be adapted using set_stereo_model_param. If the disparity values are set manually, the bounding box is only used to restrict the reconstructed 3D points. In the case of using binocular_disparity_mg as disparity method, suitable values for the parameters InitialGuess and *'initial_level'* are derived from the bounding box. However, these values can also be reset using set_stereo_model_param. Use the procedures gen_bounding_box_object_model_3d to create a 3D object model of your stereo model, and inspect it in conjunction with the reconstructed 3D object model to verify the bounding box visually.

Improve the quality of the disparity images After setting the stereo model *'persistence'* mode (see above), inspect the disparity and the score images for each image pair. They are returned by get_stereo_model_object with a camera index pair [From, To] specifying the pair of interest in the parameter PairIndex and the values *'disparity_image'* and *'score_image'* in ObjectName, respectively. If both images exhibit significant imperfection (e.g., the disparity image does not really resemble the shape of the object seen in the image), try to adjust the parameters used for the internal call to binocular_disparity (the parameters with a *'binocular_'* prefix) by modifying set_stereo_model_param until some improvement is achieved.

Alternatively, a different method to calculate the disparities can be used. Besides the above-mentioned internal call of binocular_disparity, HALCON also provides the two other methods binocular_disparity_mg and binocular_disparity_ms. These methods feature e.g., the calculation of disparities in textureless regions at an expanse of the reconstruction time if compared with cross-correlation methods. However, for these methods, it can be necessary to adapt the parameters to the underlying dataset as well. Dependent on the chosen method, the user can either set the parameters with a *'binocular_mg_'* or a *'binocular_ms_'* prefix until some improvement is achieved.

A detailed description of the provided methods and their parameters can be found in binocular_disparity, binocular_disparity_mg or binocular_disparity_ms, respectively.

Fusion parameters If the result of pairwise reconstruction as inspected by get_stereo_model_object_model_3d can not be improved anymore, begin to adapt the fusion parameters. For a description of parameters see also set_stereo_model_param. Note that the pairwise reconstruction is sometimes not discernible when the fusion algorithm can still tweak it into something sensible. In any case, pairwise reconstruction should yield enough points as input for the fusion algorithm.

Runtime

In order to improve the runtime, consider the following hints:

Extent of the bounding box The bounding box should be tight around the volume of interest. Else, the runtime will increase unnecessarily and - for the method *'surface_fusion'* - drastically.

Reduce the domain of the input images Reducing the domain of the input images (e.g., with `reduce_domain`) to the relevant part of the image may heavily speed up the algorithm, especially for large images.

Sub-sampling in the rectification step The stereo model parameter *'rectif_sub_sampling'* (see `set_stereo_model_param`) controls the sub-sampling in the rectification step. Setting this factor to a value > 1.0 will reduce the resolution of the rectified images compared to the original images. This factor has a direct impact on the succeeding performance of the chosen disparity method, but it causes loss of image detail. The parameter *'rectif_interpolation'* could have also some impact, but typically not a significant one.

Disparity parameters There is a trade-off between completeness of the pairwise surface reconstruction on the one hand and reconstruction runtime on the other. The stereo model offers three different methods to calculate the disparity images. Dependent on the chosen method, the stereo model provides a particular set of parameters that enables a precise adaption of the method to the used dataset. If the method `binocular_disparity` is selected, only parameters with a *'binocular_'* prefix can be set. For the method `binocular_disparity_mg`, all settable parameters have to exhibit the prefix *'binocular_mg_'*, whereas for the method `binocular_disparity_ms` only parameters with *'binocular_ms_'* are applicable.

Parameters using the method `binocular_disparity`:

- `NumLevels`
- `MaskWidth`
- `MaskHeight`
- `Filter`
- `SubDisparity`

Each of these parameters of `binocular_disparity` has a corresponding stereo model parameter written in snake case and with the prefix *'binocular_'*, and has, some more or others less, impact on the performance. Adapting them properly could improve the performance. performance.

Parameters using the method `binocular_disparity_mg`:

- `GrayConstancy`
- `GradientConstancy`
- `Smoothness`
- `InitialGuess`
- *'mg_solver'*
- *'mg_cycle_type'*
- *'mg_pre_relax'*
- *'mg_post_relax'*
- *'initial_level'*
- *'iterations'*
- *'pyramid_factor'*

Each of these parameters of `binocular_disparity_mg` has a corresponding stereo model parameter written in snake case and with the prefix *'binocular_mg_'*, and has, some more or others less, impact on the performance and the result. Adapting them properly could improve the performance.

Parameters using the method `binocular_disparity_ms`:

- `SurfaceSmoothing`
- `EdgeSmoothing`
- *'consistency_check'*
- *'similarity_measure'*

- *'sub_disparity'*

Each of these parameters of binocular_disparity_ms has a corresponding stereo model parameter written in snake case and with the prefix *'binocular_ms_'*, and has, some more or others less, impact on the performance and the result. Adapting them properly could improve the performance.

Reconstruct only points with high disparity score Besides adapting the sub-sampling it is also possible to exclude points of the 3D reconstruction because of their computed disparity score. In order to do this, the user should first query the score images for the disparity values by calling get_stereo_model_object using GenParamName = *'score_image'*. Dependent on the distribution of these values, the user can decide whether disparities with a score beneath a certain threshold should be excluded from the reconstruction. This can be achieved with set_stereo_model_param using either GenParamName = *'binocular_score_thresh'*. The advantage of excluding points of the reconstruction is a slight speed-up since it is not necessary to process the entire dataset. As an alternative to the above-mentioned procedure, it is also possible to exclude points after executing reconstruct_surface_stereo by filtering reconstructed 3D points. The advantage of this is that at the expense of a slightly increased runtime, a second call to reconstruct_surface_stereo is not necessary.

Sub-sampling of X,Y,Z data For the method *'surface_pairwise'*, you can use a larger sub-sampling step for the X,Y,Z data in the last step of the reconstruction algorithm by modifying GenParamName=*'sub_sampling_step'* with set_stereo_model_param. The reconstructed data will be much sparser, thus speeding up the post-processing.

Fusion parameters For the method *'surface_fusion'*, enlarging the parameter *'resolution'* will speed up the execution considerably.

Parameters

▷ **Images** (input_object) ..... singlechannelimage-array ⇝ *object :* byte
An image array acquired by the camera setup associated with the stereo model.

▷ **StereoModelID** (input_control) ..... stereo_model ⇝ *handle*
Handle of the stereo model.

▷ **ObjectModel3D** (output_control) ..... object_model_3d ⇝ *handle*
Handle to the resulting surface.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

create_stereo_model, get_calib_data, set_stereo_model_image_pairs

Possible Successors

get_stereo_model_object_model_3d

Alternatives

reconstruct_points_stereo

References

M. Kazhdan, M. Bolitho, and H. Hoppe: "Poisson Surface Reconstruction." Symposium on Geometry Processing (June 2006).,

C. Zach, T. Pock, and H. Bischof: "A globally optimal algorithm for robust TV-L1 range image integration." Proceedings of IEEE International Conference on Computer Vision (ICCV 2007).

Module

3D Metrology

set_stereo_model_image_pairs (: : StereoModelID, From, To :)

Specify image pairs to be used for surface stereo reconstruction.

The operator set_stereo_model_image_pairs stores a list of image pairs for a stereo model StereoModelID of type *'surface_pairwise'* or *'surface_fusion'*. Calling the operator for a model of another type will raise an error. In the mode *'surface_pairwise'* or *'surface_fusion'*, surfaces are reconstructed by computing disparity images for image pairs. You specify these image pairs by passing tuples of camera indices in the parameters From and To. Then, e.g., the disparity image from the camera with index From[0] to the camera with index To[0] is computed and so on.

The camera indices must be valid for the camera setup model assigned to the stereo model (see create_stereo_model), otherwise an error is returned. If an image pairs list already exists in the stereo model, it is substituted by the current one.

Besides storing the list of image pairs, the operator set_stereo_model_image_pairs prepares a pair of rectification image maps for each image pair, which are used repeatedly in successive calls to reconstruct_surface_stereo to rectify the images to a normalized binocular stereo pair; refer to gen_binocular_rectification_map for further details. Three of the gen_binocular_rectification_map parameters are exported as stereo model parameters and can be modified by set_stereo_model_param or just inspected by get_stereo_model_param:

'rectif_interpolation': Interpolation mode corresponding to the parameter MapType of gen_binocular_rectification_map.

'rectif_sub_sampling': sub-sampling factor corresponding to the parameter SubSampling of gen_binocular_rectification_map.

'rectif_method': Rectification method corresponding to the parameter Method of gen_binocular_rectification_map.

Note that after modifying these parameters, set_stereo_model_image_pairs must be executed again for the changes to take effect.

The current list of image pairs in the model can be inspected by get_stereo_model_image_pairs.

Parameters

▷ **StereoModelID** (input_control) ..... stereo_model ⇝ *handle*
Handle of the stereo model.

▷ **From** (input_control) ..... integer-array ⇝ *integer*
Camera indices for the *from* cameras in the image pairs.
Number of elements: From > 0

▷ **To** (input_control) ..... integer-array ⇝ *integer*
Camera indices for the *to* cameras in the image pairs.
Number of elements: To == From

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- StereoModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_stereo_model

Possible Successors

reconstruct_surface_stereo

See also

set_stereo_model_param, get_stereo_model_image_pairs

Module

3D Metrology

set_stereo_model_param (: : StereoModelID, GenParamName, GenParamValue :)

Set stereo model parameters.

The operator set_stereo_model_param can be used to set diverse parameters for the stereo model StereoModelID. Several types of parameters can be set with this operator depending on type of the stereo model which was specified in create_stereo_model. Note that no specific parameters are provided for *'points_3d'*.

General parameters:

By setting GenParamName to one of the following values, general stereo model parameters can be set to the value passed in GenParamValue:

'bounding_box': A tuple [x1,y1,z1,x2,y2,z2] specifying two opposite corner points P1=[x1,y1,z1] and P2=[x2,y2,z2] of a bounding box for the reconstructions. The bounding box defines a box in the space in the coordinate frame of the camera setup model used for the reconstruction (specified by CameraSetupModelID in create_stereo_model). The reconstruction algorithms then clip any resulting reconstruction to this bounding box.

Furthermore, if the parameters *'min_disparity'* and *'max_disparity'* are not set manually by using create_stereo_model or set_stereo_model_param, the operator reconstruct_surface_stereo requires a valid bounding box for the estimation of the minimal and maximal disparity parameters for the pairwise disparity estimation (see set_stereo_model_image_pairs for more details).

Note that the values of parameters for the fusion of surfaces are reset to default values each time the bounding box is reset.

You can use the procedure estimate_bounding_box_3d_reconstruction to get initial values for the bounding box of your 3D reconstruction. This bounding box is based on the pose of a reference calibration plate and the cones of sight of the cameras. Later, the bounding box should be set as tight as possible around the object that is to be to be reconstructed.

Additionally, the procedures gen_bounding_box_object_model_3d and gen_camera_setup_object_model_3d can be used to visualize your camera setup.

For a valid bounding box, P1 must be the point on the front lower left corner and P2 on the back upper right corner of the bounding box, i.e., x1<x2, y1<y2 and z1<z2. While the surface reconstruction (see reconstruct_surface_stereo) will terminate in the case of an invalid bounding box, the 3D point reconstruction algorithm (see reconstruct_points_stereo) simply ignores it, and it reconstructs all points (it can), without clipping them. Thus, you can turn off the result clipping for the 3D points reconstruction by passing the tuple [0,0,0,0,0,-1].

Note that because *'bounding_box'* is a tuple-valued parameter, it cannot be set in a single call of set_stereo_model_param together with other model parameters (see the paragraph "A note on tuple-valued model parameters" below).

Tuple format: [x1,y1,z1,x2,y2,z2]

'persistence': Enables (GenParamValue=*1*) or disables (GenParamValue=*0*) the *'persistence'* mode of the stereo model. When in persistence mode, the model stores intermediate results of the reconstruction (only for reconstruct_surface_stereo), which can be inspected later by get_stereo_model_object and get_stereo_model_object_model_3d.

Note that the model might need significant memory space in this mode. This can worsen the performance of the reconstruction algorithms and even lead to running out of memory, in particular for setups with many cameras and/or large images. Therefore, we recommend to enable this mode only for inspection and debugging a reconstruction with small data sets.

List of values: *0* , *1*.

Default: *0*.

Parameters for the surface reconstruction using 'surface_pairwise' or 'surface_fusion':

By setting GenParamName to one of the following values, additional parameters specific for surface reconstruction can be set with GenParamValue for a stereo model of type *'surface_pairwise'* or *'surface_fusion'*:

'color': By setting this parameter to one of the following values, the coloring of the reconstructed 3D object model is either enabled or disabled (*'none'*). See reconstruct_surface_stereo on how to access the resulting color information.

'median' The color value of a 3D point is the median of the color values of all cameras where the 3D point is visible.

'smallest_distance' The color value of a 3D point corresponds to the color value of the camera that exhibits the smallest distance to this 3D point.

'mean_weighted_distances' All cameras that contribute to the reconstruction of a 3D point are weighted according to their distance to the 3D point. Cameras with a smaller distance receive a higher weight, whereas cameras with a larger distance get a lower weight. The color value of a 3D point is then computed by averaging the weighted color values of the cameras.

'line_of_sight' The color value of a 3D point corresponds to the color value of the camera that exhibits the smallest angle between the point normal and the line of sight.

'mean_weighted_lines_of_sight' All cameras that contribute to the reconstruction of a 3D point are weighted according to their angle between the point normal and the line of sight. Cameras with a smaller angle receive a higher weight. The color value of a 3D point is then computed by averaging the weighted color values of the cameras.

List of values: *'none'*, *'smallest_distance'*, *'mean_weighted_distances'*, *'line_of_sight'*, *'mean_weighted_lines_of_sight'*, *'median'*.

Default: *'none'*.

'color_invisible': If stereo models of type *'surface_fusion'* are used, the reconstruction will contain points without a direct correspondence to points in the images. These points are not seen by any of the cameras of the stereo system and are therefore "invisible". A color value for these points has to be calculated using the color of points in the vicinity. Coloring these "invisible" points can be switched off by setting this parameter to *'false'*. In this case invisible points are assigned 255 as gray value. Normally, coloring of "invisible" points is not very time-consuming and can remain active. However, it may happen that the value for the parameter *'resolution'* is considerably finer than the available image resolution. In this case, many invisible 3D points are reconstructed making the nearest neighbor search very time consuming. In order to avoid an increased runtime, it is recommended to either adapt the value of *'resolution'* or to switch off the calculation for invisible points. Please note that for stereo models of type *'surface_pairwise'*, this parameter will not have any effect.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'rectif_interpolation': Interpolation mode for the rectification maps (see set_stereo_model_image_pairs). Note that after changing this parameter, you must call set_stereo_model_image_pairs again for the changes to take effect.

List of values: *'none'*, *'bilinear'*.

Default: *'bilinear'*.

'rectif_sub_sampling': Sub-sampling factor for the rectification maps (see set_stereo_model_image_pairs). Note that after changing this parameter, you must call set_stereo_model_image_pairs again for the changes to take effect.

Suggested values: *0.5*, *0.66*, *1.0*, *1.5*, *2.0*, *3.0*, *4.0*.

Default: *1.0*.

'rectif_method': Rectification method for the rectification maps (see set_stereo_model_image_pairs). Note that after changing this parameter, you must call set_stereo_model_image_pairs again for the changes to take effect.

List of values: *'viewing_direction'*, *'geometric'*.

Default: *'viewing_direction'*.

'disparity_method': Method used to create disparity images from the image pairs (see reconstruct_surface_stereo). Currently, the three methods *'binocular'*, *'binocular_mg'* and *'binocular_ms'* are supported. Dependent on the chosen method, the HALCON operator

binocular_disparity, binocular_disparity_mg or binocular_disparity_ms is called internally.

List of values: *'binocular'*, *'binocular_mg'*, *'binocular_ms'*.

Default: *'binocular'*.

'min_disparity', *'max_disparity'*: Minimum and maximum disparity values that are used in the operator reconstruct_surface_stereo. The number of minimum and maximum disparity values must correspond to the number of image pairs. If *'min_disparity'* and *'max_disparity'* are not set by the operator set_stereo_model_param, the disparity values are estimated internally by using the underlying bounding box.

Note that because *'min_disparity'* and *'max_disparity'* are tuple-valued parameters, they cannot be set in a single call of set_stereo_model_param together with other model parameters (see the paragraph "A note on tuple-valued model parameters" below).

'binocular_score_thresh': For the methods *'binocular_mg'* and *'binocular_ms'* the disparities that have a score above the passed threshold are excluded from further processing steps and do not end up in the reconstructed 3D point cloud. For the method *'binocular'* the disparities below the passed threshold are excluded.

For stereo models with the method *'binocular'*: **List of values:** positive and negative integer or float value.
Default: *0.5*.

For stereo models with the method *'binocular_mg'* **or** *'binocular_ms'*: **List of values:** integer or float value greater or equal to 0.0.
Default: *-1*.

Depending on the selected disparity method, a set of different parameters is available for the user. These parameters allow a fine tuning to the used data set. More information about the parameters can be found in the respective operator reference of binocular_disparity, binocular_disparity_mg or binocular_disparity_ms.

Set of parameters for stereo models with method = *'binocular'*

'binocular_method': Sets the desired matching method.
List of values: *'ncc'*, *'sad'*, *'ssd'*.
Default: *'ncc'*.

'binocular_num_levels': Number of used image pyramids. **List of values:** integer value greater or equal to 1.
Default: *1*.

'binocular_mask_width': Width of the correlation window.
List of values: Odd integer value greater or equal to 3.
Default: *11*.

'binocular_mask_height': Height of the correlation window.
List of values: Odd integer value greater or equal to 3.
Default: *11*.

'binocular_texture_thresh': Variance threshold of textured image regions.
List of values: integer or float value greater or equal to 0.0.
Default: *0.0*.

'binocular_filter': Downstream filters.
List of values: *'none'*, *'left_right_check'*.
Default: *'none'*.

'binocular_sub_disparity': Subpixel interpolation of disparities.
List of values: *'none'*, *'interpolation'*.
Default: *'none'*.

Set of parameters for stereo models with method = *'binocular_mg'*

'binocular_mg_gray_constancy': Weight of the gray value constancy in the data term.
List of values: integer or float value greater or equal to *0.0*.
Default: *1.0*.

'binocular_mg_gradient_constancy': Weight of the gradient constancy in the data term.
List of values: integer or float value greater or equal to *0.0*.
Default: *30.0*.

'binocular_mg_smoothness': Weight of the smoothness term in relation to the data term.
List of values: integer or float value greater *0.0*.
Default: *5.0*.

'binocular_mg_initial_guess': Initial guess of the disparity.
List of values: integer or float value.
Default: *0.0*.

The subsequent parameters control the behavior of the used multigrid method.

'binocular_mg_default_parameters': Sets predefined values for the following parameters of the used multigrid method: *'binocular_mg_solver'*, *'binocular_mg_cycle_type'*, *'binocular_mg_pre_relax'*, *'binocular_mg_post_relax'*, *'binocular_mg_initial_level'*, *'binocular_mg_iterations'*, *'binocular_mg_pyramid_factor'*. The exact values of these parameters can be found in the operator reference of `binocular_disparity_mg`.
List of values: *'very_accurate'*, *'accurate'*, *'fast_accurate'*, *'fast'*.
Default: *'fast_accurate'*.

'binocular_mg_solver': Solver for the linear system.
List of values: *'multigrid'*, *'full_multigrid'*, *'gauss_seidel'*.
Default: *'full_multigrid'*.

'binocular_mg_cycle_type': Selects the type of recursion for the multigrid solvers.
List of values: *'v'*,*'w'*, *'none'*.
Default: *'v'*.

'binocular_mg_pre_relax': Sets the number of iterations of the pre-relaxation step in multigrid solvers, or the number of iterations for the Gauss-Seidel solver, depending on which is selected.
List of values: integer or float value greater *0.0*.
Default: *1*.

'binocular_mg_post_relax': Sets the number of iterations of the post-relaxation step.
List of values: integer or float value.
Default: *1*.

'binocular_mg_initial_level': Sets the coarsest level of the image pyramid where the coarse-to-fine process starts.
List of values: integer value.
Default: *-2*.

'binocular_mg_iterations': Sets the number of iterations of the fixed point iteration per pyramid level.
List of values: integer or float value greater or equal to *0*.
Default: *1*

'binocular_mg_pyramid_factor': Determines the factor by which the images are scaled when creating the image pyramid for the coarse-to-fine process.
List of values: integer or float value between *0.1* and *0.9*.
Default: *0.6*.

Set of parameters for stereo models with method = *'binocular_ms'*

'binocular_ms_surface_smoothing': Smoothing of surfaces.
List of values: integer value greater or equal to *0*.
Default: *50*.

'binocular_ms_edge_smoothing': Smoothing of edges.
List of values: integer value greater or equal to *0*.
Default: *50*.

'binocular_ms_consistency_check': This parameter increases the robustness of the returned matches since the result relies on a concurrent direct and reverse match.
List of values: *'true'*, *'false'*.
Default: *'true'*.

'binocular_ms_similarity_measure': Sets the method of the similarity measure.
List of values: *'census_dense'*, *'census_sparse'*.
Default: *'census_dense'*.

'binocular_ms_sub_disparity': Enables or disables the sub-pixel refinement of disparities.
List of values: *'true'*, *'false'*.
Default: *'true'*.

'point_meshing': Enables the post-processing step for meshing the reconstructed surface points. For a stereo model of type *'surface_pairwise'*, a Poisson solver is supported. For a stereo model of type *'surface_fusion'*, a meshing of the isosurface is supported (see `reconstruct_surface_stereo` for more details).

List of values: *'none'*, *'poisson'*, *'isosurface'*.

Default: *'none'*.

If the Poisson-based meshing is enabled, the following parameters can be set:

- *'poisson_depth'*: Depth of the solver octree. More detail (i.e., a higher resolution) of the resulting mesh is achieved with deeper trees. However, this requires more time and memory.
  Suggested values:
  6, *8*, *10*.
  Default: *8*.
  Restriction: *3 <= 'poisson_depth' <= 12*
- *'poisson_solver_divide'*: Depth of block Gauss-Seidel solver used for solving the Poisson equation. At the price of a small time overhead, this parameter reduces the memory consumption of the underlying meshing algorithm. Proposed values are depths by 0 to 2 smaller compared to the main octree depth.
  Suggested values: *6*, *8*, *10*.
  Default: *8*.
  Restriction: *3 <= 'poisson_solver_divide' <= 'poisson_depth'*
- *'poisson_samples_per_node'*: Minimum number of points that should fall in a single octree leaf. This parameter is used to handle noisy data, e.g., noise-free data can be distributed over many leaves, whereas more noisy data should be stored in a single leaf to compensate for the noise. As a side effect, bigger values of this parameter distribute the data in fewer leaves, which results in a smaller octree, which means a speedup but possibly less detail of the reconstruction.
  Suggested values: *1*, *5*, *10*, *30*, *40*.
  Default: *30*.

Parameters only for 'surface_pairwise':

By setting `GenParamName` to one of the following values, additional parameters specific for surface reconstruction can be set with `GenParamValue` for a stereo model of type *'surface_pairwise'*:

'sub_sampling_step': sub-sampling step for the X, Y and Z image data resulting from the pairwise disparity estimation, before this data is used in its turn for the surface reconstruction (see `reconstruct_surface_stereo`).

Suggested values: *1*, *2*, *3*.

Default: *2*.

Parameters only for 'surface_fusion':

By setting `GenParamName` to one of the following values, additional parameters specific for surface reconstruction can be set with `GenParamValue` for a stereo model of type *'surface_fusion'*:

'resolution': Distance of neighboring sample points in each coordinate direction in discretization of bounding box. *'resolution'* is set in [m]. See `reconstruct_surface_stereo` for more details.

Too small values will unnecessarily increase the runtime. Too large values will lead to a reconstruction with too few details. Per default, it is set to a coarse resolution depending on the bounding box. The parameter will be reset if the bounding box is reset.

'smoothing' may need to be adapted when *'resolution'* is changed.

'surface_tolerance' should always be a bit larger than *'resolution'* in order to avoid effects of discretization.

Suggested values: *0.001*, *0.01*

'surface_tolerance': Specifies how much noise around the input point cloud should be combined to a surface. Points in a cone of sight of a camera are considered surely outside of the object (in front of the surface) or surely inside the object (behind the surface) with respect to the given camera if their distance to the initial surface exceeds *'surface_tolerance'*. *'surface_tolerance'* is set in [m]. See `reconstruct_surface_stereo` for more details and a figure.

Too small values lead to an uneven surface. Too large values smudge distinct surfaces into one. Per default, it is set to three times *'resolution'*. The parameter will be reset if the bounding box is reset.

'surface_tolerance' should always be a bit larger than *'resolution'* in order to avoid effects of discretization. *'min_thickness'* always has to be larger than or equal to *'surface_tolerance'*. If *'min_thickness'* is set too

small, *'surface_tolerance'* is automatically set to the same value as *'min_thickness'*. If *'surface_tolerance'* is set too big, an error is raised.

Suggested values: *0.003*, *0.03*

Restriction: *'surface_tolerance'* < *'min_thickness'*

'min_thickness': Length of considered cone of sight of a camera behind the initial surface obtained by pairwise reconstruction. Points behind the surface (viewed from the given camera) are only considered to lie inside the object if their distance to the initial surface does not exceed *'min_thickness'*. *'min_thickness'* is set in [m]. See `reconstruct_surface_stereo` for more details and a figure.

If lines of sight are expected to intersect the closed object only once (cameras all observe the object head-on from one side), this parameter should remain at the very large default setting.

If lines of sight are expected to intersect the object more often (cameras observe the object from different sides), only the interior of the object of interest should be marked as lying behind the surface. Thus, a first guess for the parameter could be less than the thickness of your object.

The method *'surface_fusion'* will try to produce a closed surface. If you observe several distinct objects from only one side, you may want to reduce the parameter *'min_thickness'* to restrict the depth of reconstructed objects and thus keep them from being smudged into one surface. The backside of the objects is not observed and thus its reconstruction will probably be incorrect.

Too small values can result in holes in the reconstructed point cloud or double walls. Too large values can result in a distorted point cloud or blow up the surface towards the outside of the object (if the surface is blown up beyond the bounding box, no points will be reconstructed). Per default set to the diameter of the bounding box. The parameter will be reset if the bounding box is reset.

'min_thickness' always has to be larger than or equal to *'surface_tolerance'*. If *'min_thickness'* is set too small, *'surface_tolerance'* is automatically set to the same value as *'min_thickness'*. If *'surface_tolerance'* is set too big, an error is raised.

Suggested values: *0.005*, *0.05*.

'smoothing': The parameter *'smoothing'* determines how important a small total variation of the distance function is compared to data fidelity. Thus, *'smoothing'* regulates the 'jumpiness' of the resulting surface (see `reconstruct_surface_stereo` for more details).

Note that the actual value of *'smoothing'* for a given data set to be visually pleasing has to be found by try and error. Too small values lead to integrating many outliers into the surface even if the surface then exhibits many jumps. Too large values lead to lost fidelity towards the point clouds of pairwise reconstruction (how the algorithm views distances to the input point clouds depends heavily on *'surface_tolerance'* and *'min_thickness'*).

The parameter will be reset if the bounding box is reset. *'smoothing'* may need to be adapted when *'resolution'* is changed.

Suggested values: *15.0*, *1.0*, *0.1*.

Default: *1.0*.

All parameters except *'binocular_mg_default_parameters'* can be read back by `get_stereo_model_param`.

A note on tuple-valued model parameters

Most of the stereo model parameters are single-valued. Thus, you can provide a list (i.e., tuple) of parameter names and a list (tuple) of values that has the same length as the input tuple. In contrast, when setting a tuple-valued parameter, you must pass a tuple of values. When setting such a parameter together with other parameters, the value-to-parameter-name correspondence is not obvious anymore. Thus, tuple-valued parameters like *'bounding_box'*, *'min_disparity'* or *'max_disparity'* should always be set in a separate call to `set_stereo_model_param`.

Parameters

▷ **`StereoModelID`** (input_control) ..... stereo_model ⇝ *handle*
Handle of the stereo model.

▷ **`GenParamName`** (input_control) ..... attribute.name(-array) ⇝ *string*
Names of the parameters to be set.
List of values: `GenParamName` ∈ {'bounding_box', 'persistence', 'sub_sampling_step', 'rectif_interpolation', 'rectif_sub_sampling', 'rectif_method', 'disparity_method', 'binocular_method', 'binocular_num_levels', 'binocular_mask_width', 'binocular_mask_height', 'binocular_texture_thresh', 'binocular_score_thresh', 'binocular_filter', 'binocular_sub_disparity', 'binocular_mg_gray_constancy', 'binocular_mg_gradient_constancy', 'binocular_mg_smoothness', 'binocular_mg_initial_guess',

'binocular_mg_default_parameters', 'binocular_mg_solver', 'binocular_mg_cycle_type', 'binocular_mg_pre_relax', 'binocular_mg_post_relax', 'binocular_mg_initial_level', 'binocular_mg_iterations', 'binocular_mg_pyramid_factor', 'binocular_ms_surface_smoothing', 'binocular_ms_edge_smoothing', 'binocular_ms_consistency_check', 'binocular_ms_similarity_measure', 'binocular_ms_sub_disparity', 'point_meshing', 'poisson_depth', 'poisson_solver_divide', 'poisson_samples_per_node', 'resolution', 'surface_tolerance', 'min_thickness', 'smoothing', 'color', 'color_invisible', 'min_disparity', 'max_disparity'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of the parameters to be set.
Suggested values: GenParamValue ∈ {1, -2, -5, 0, 0.3, 0.5, 0.9, 1, 2, 3, 'census_dense', 'census_sparse', 'binocular', 'ncc', 'none', 'sad', 'ssd', 'bilinear', 'false', 'viewing_direction', 'geometric', 'very_accurate', 'accurate', 'fast_accurate', 'fast', 'v', 'w', 'none', 'gauss_seidel', 'multigrid', 'true', 'poisson', 'isosurface', 'interpolation', 'left_right_check', 'full_multigrid', 'binocular_mg', 'binocular_ms', 'smallest_distance', 'mean_weighted_distances', 'line_of_sight', 'mean_weighted_lines_of_sight', 'median'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- StereoModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_stereo_model

Possible Successors

reconstruct_surface_stereo, reconstruct_points_stereo

See also

get_stereo_model_param, set_stereo_model_image_pairs

Module

3D Metrology

5.4 Photometric Stereo

estimate_al_am (Image : : : Albedo, Ambient)

Estimate the albedo of a surface and the amount of ambient light.

estimate_al_am estimates the Albedo of a surface, i.e. the percentage of light reflected by the surface, and the amount of ambient light Ambient by using the maximum and minimum gray values of the image.

Attention

It is assumed that the image contains at least one point for which the reflection function assumes its minimum, e.g., points in shadows. Furthermore, it is assumed that the image contains at least one point for which the reflection function assumes its maximum. If this is not the case, wrong values will be estimated.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Image for which albedo and ambient are to be estimated.

▷ **Albedo** (output_control) . . . . . real(-array) ⇝ *real*
Amount of light reflected by the surface.

▷ **Ambient** (output_control) . . . . . real(-array) ⇝ *real*
Amount of ambient light.

Result

`estimate_al_am` always returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

`sfs_mod_lr`, `sfs_orig_lr`, `sfs_pentland`, `photometric_stereo`, `shade_height_field`

Module

3D Metrology

estimate_sl_al_lr (Image : : : Slant, Albedo)

Estimate the slant of a light source and the albedo of a surface.

`estimate_sl_al_lr` estimates the `Slant` of a light source, i.e., the angle between the light source and the positive z-axis, and the albedo of the surface in the input image `Image`, i.e. the percentage of light reflected by the surface, using the algorithm of Lee and Rosenfeld.

Attention

The `Albedo` is assumed constant for the entire surface depicted in the image.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
Image for which slant and albedo are to be estimated.

▷ **Slant** (output_control) ..... angle.deg(-array) ⇝ *real*
Angle between the light sources and the positive z-axis (in degrees).

▷ **Albedo** (output_control) ..... real(-array) ⇝ *real*
Amount of light reflected by the surface.

Result

`estimate_sl_al_lr` always returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

`sfs_mod_lr`, `sfs_orig_lr`, `sfs_pentland`, `photometric_stereo`, `shade_height_field`

Module

3D Metrology

estimate_sl_al_zc (Image : : : Slant, Albedo)

Estimate the slant of a light source and the albedo of a surface.

`estimate_sl_al_zc` estimates the `Slant` of a light source, i.e. the angle between the light source and the positive z-axis, and the albedo of the surface in the input image `Image`, i.e. the percentage of light reflected by the surface, using the algorithm of Zheng and Chellappa.

Attention

The `Albedo` is assumed constant for the entire surface depicted in the image.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Image for which slant and albedo are to be estimated.

▷ **Slant** (output_control) . . . . . angle.deg(-array) ⇝ *real*
Angle of the light sources and the positive z-axis (in degrees).

▷ **Albedo** (output_control) . . . . . real(-array) ⇝ *real*
Amount of light reflected by the surface.

Result

`estimate_sl_al_zc` always returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

`sfs_mod_lr`, `sfs_orig_lr`, `sfs_pentland`, `photometric_stereo`, `shade_height_field`

Module

3D Metrology

estimate_tilt_lr (Image : : : Tilt)

Estimate the tilt of a light source.

`estimate_tilt_lr` estimates the tilt of a light source, i.e. the angle between the light source and the x-axis after projection into the xy-plane, from the image `Image` using the algorithm of Lee and Rosenfeld.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Image for which the tilt is to be estimated.

▷ **Tilt** (output_control) . . . . . angle.deg(-array) ⇝ *real*
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).

Result

`estimate_tilt_lr` always returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

`sfs_mod_lr`, `sfs_orig_lr`, `sfs_pentland`, `photometric_stereo`, `shade_height_field`

Module

3D Metrology

estimate_tilt_zc (Image : : : Tilt)

Estimate the tilt of a light source.

`estimate_tilt_zc` estimates the tilt of a light source, i.e. the angle between the light source and the x-axis after projection into the xy-plane, from the image `Image` using the algorithm of Zheng and Chellappa.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Image for which the tilt is to be estimated.

▷ **Tilt** (output_control) . . . . . angle.deg(-array) ⇝ *real*
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).

Result

`estimate_tilt_zc` always returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

`sfs_mod_lr`, `sfs_orig_lr`, `sfs_pentland`, `photometric_stereo`, `shade_height_field`

Module

3D Metrology

photometric_stereo (Images : HeightField, Gradient, Albedo : Slants, Tilts, ResultType, ReconstructionMethod, GenParamName, GenParamValue :)

Reconstruct a surface according to the photometric stereo technique.

`photometric_stereo` can be used to separate the three-dimensional shape of an object from its two-dimensional texture, e.g., its print image. The operator requires at least three images of the same object taken with different and known directions of illumination. Note, that the point of view of the camera must be the same for all images.

The three-dimensional shape of the object is primarily computed as the local gradients of the three-dimensional surface. Those gradients can be further integrated to obtain a height field, i.e., an image in which the pixel values correspond to a relative height. The two-dimensional texture is called albedo and corresponds to the local light absorption and reflection characteristics of the surface exclusive of any shading effect.

Typical applications of photometric stereo

Typical applications of photometric stereo are to detect small inconsistencies in a surface that represent, e.g., defects, or to exclude the influence of the direction of light from images that are used, e.g., for the print inspection of non flat characters. Note that photometric stereo is not suitable for the reconstruction of absolute heights, i.e., it is no alternative to typical 3D reconstruction algorithms like depth from focus or sheet of light.

Limitations of photometric stereo

`photometric_stereo` is based on the algorithm of Woodham and therefore assumes on the one hand that the camera performs an orthoscopic projection. That is, you must use a telecentric lens or a lens with a long focal distance. On the other hand, it assumes that each of the light sources delivers a parallel and uniform beam of light. That is, you must use telecentric illumination sources with uniform intensity or, as an alternative, distant point light sources. Additionally, the object must have Lambertian reflectance characteristics, i.e., it must reflect incoming light in a diffuse way. Objects or regions of an object that have specular reflectance characteristics (i.e., mirroring or glossy surfaces) cannot be processed correctly and thus lead to erroneous results.

The acquisition setup

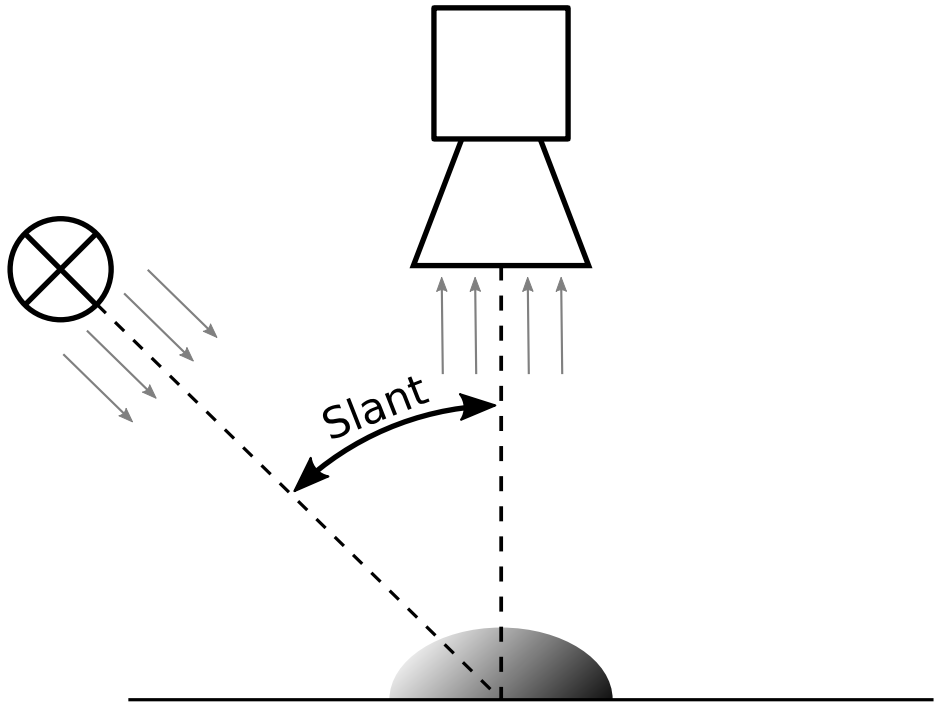
The camera with a telecentric lens must be placed orthogonally, i.e., perpendicular, to the scene that should be reconstructed. The orientation of the camera with respect to the scene must not change during the acquisition of the images. In contrast, the orientation of the illumination with respect to the camera must change for at least three gray value images.

Specifying the directions of illumination

For each image, the directions of illumination must be specified as angles within the parameters `Slants` and `Tilts`, which describe the direction of the illumination in relation to the scene. To understand the meaning of the

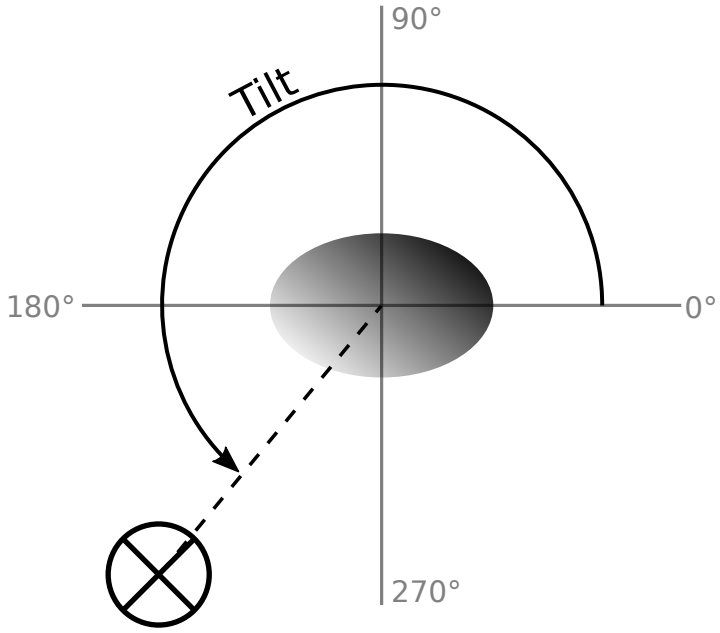
parameters Slants and Tilts, remember that the illumination source is assumed to produce parallel light rays, the camera has a telecentric lens, and the camera is placed orthogonal to the scene to reconstruct:

Slants The Slants angle is the angle between the optical axis of the camera and the direction of the illumination.



Side view

Tilts The Tilts angle is measured within the object plane or any plane that is parallel to it, e.g., the image plane. In particular, it describes the angle between the direction that points from the center of the image to the right and the direction of light that is projected into the plane. That is, when looking at the image (or the corresponding scene), a tilt angle of 0 means that the light comes from the right, a tilt angle of 90 means that the light is coming from the top, a tilt angle of 180 means that the light is coming from the left, etc.



Top view

As stated before, photometric stereo requires at least three images with different directions of illumination. However, the three-dimensional geometry of objects typically leads to shadow casting. In the shadow regions, the number of effectively available directions of illumination is reduced, which leads to ambiguities. To nevertheless get a robust result, redundancy is needed. Therefore, typically more than three light sources with different directions should be used. But note that an increasing number of illumination directions also leads to a higher number

of images to be processed and therefore to a higher processing time. In most applications, a number of four to six light sources is reasonable. As a rule of thumb, the slant angles should be chosen between 30° and 60°. The tilt angles typically should be equally distributed around the object to be measured. Please note that the directions of illumination must be selected such that they do not lie in the same plane (i.e., the illumination directions must be independent), otherwise the computing fails and an exception is thrown.

Input images and domains of definition

The input images must be provided in an image array (`Images`). Each image must have been taken with a different direction of illumination as stated above. If the images are primarily stored in a multi-channel image, they can be easily converted to an image array using `image_to_channels`. As an alternative, the image array can be created using `concat_obj`.

`photometric_stereo` relies on the evaluation of the "photometric information", i.e., the gray values stored in the images. Therefore, this information should be unbiased and accurate. We recommend to ensure that the camera that is used to acquire the images has a linear characteristic. You can use the operator `radiometric_self_calibration` to determine the characteristic of your camera and the operator `lut_trans` to correct the gray value information in case of a non linear characteristic. Additionally, if accurate measurements are required, we recommend to utilize the full dynamic range of the camera since this leads to more accurate gray value information. For the same reason, using images with a bit depth higher than 8 (e.g., uint2 images instead of byte images) leads to a better accuracy.

The domain of definition of the input images determines which algorithm is used internally to process the `Images`. Three algorithms are available:

- If all images have a full domain, the fastest algorithm is used. This mode is recommended for most applications.
- If the input images share the same reduced domain of definition, only the pixels within the domain are processed. This mode can be used to exclude areas of the object from all images. Typically, areas are excluded that are known to show non-Lambertian reflectance characteristics or that are of no interest, e.g., holes in the surface.
- If images with distinct domains of definition are provided, only the gray values that are contained in the domains are used in the respective images. Then, only those pixels are processed that have independent slant and tilt angles in at least three images. This mode is suitable, e.g., to exclude specific regions of individual images from the processing. These can be, e.g., areas of the object for which is known that they show non-Lambertian reflectance characteristics or regions for which is known that they contain biased photometric information, e.g., shadows. To exclude such regions leads to more accurate results. Please note that this last mode requires significantly more processing time than the modes that use the full domain or the same domain for all images.

Output images

The operator can return the images for the reconstructed `Gradient`, `Albedo`, and the `HeightField` of the surface:

- The `Gradient` image is a vector field that contains the partial derivative of the surface. Note that `Gradient` can be used as input to `reconstruct_height_field_from_gradient`. For visualization purposes, instead of the surface gradients normalized surface normals can be returned. Then, `ResultType` must be set to *'normalized_surface_normal'* (legacy: *'normalized_gradient'*) instead of *'gradient'*. Here, the row and column components represent the row and column components of the normalized surface normal. If `ResultType` is set to *'all'*, the default mode, i.e., *'gradient'* and not *'normalized_surface_normal'* is used.
- The `Albedo` image describes the ratio of reflected radiation to incident radiation and has a value between one (white surface) and zero (black surface). Thus, the albedo is a characteristic of the surface. For example, for a printed surface it corresponds to the print image exclusive of any influences of the incident light (shading).
- The `HeightField` image is an image in which the pixel values correspond to a relative height.

By default, all of these iconic objects are returned, i.e., the parameter `ResultType` is set to *'all'*. In case that only some of these results are needed, the parameter `ResultType` can be set to a tuple specifying only the required results among the values *'gradient'*, *'albedo'*, and *'height_field'*. Note that in certain applications like surface inspection tasks only the `Gradient` or `Albedo` images are required. Here, one can significantly

increase the processing speed by not reconstructing the surface, i.e., by passing only *'gradient'* and *'albedo'* but not *'height_field'* to ResultType.

Note that internally photometric_stereo first determines the gradient values and, if required, integrates these values in order to obtain the height field. This integration is performed by the same algorithms that are provided by the operator reconstruct_height_field_from_gradient and that can be controlled by the parameters ReconstructionMethod, GenParamName, and GenParamValue. Please, refer to the operator reconstruct_height_field_from_gradient for more information on these parameters. If ResultType is set such that *'height_field'* is not one of the results, the parameters ReconstructionMethod, GenParamName, and GenParamValue are ignored.

Attention

Note that photometric_stereo assumes square pixels. Additionally, it assumes that the heights are computed on a lattice with step width 1 in object space. If this is not the case, i.e., if the pixel size of the camera projected into the object space differs from 1, the returned height values must be multiplied by the actual step width (value of the pixel size projected into the object space). The size of the pixel in object space is computed by dividing the size of the pixel in the camera by the magnification of the (telecentric) lens.

Parameters

▷ **Images** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2
Array with at least three input images with different directions of illumination.

▷ **HeightField** (output_object) . . . . . image ⇝ *object :* real
Reconstructed height field.

▷ **Gradient** (output_object) . . . . . image ⇝ *object :* vector_field
The gradient field of the surface.

▷ **Albedo** (output_object) . . . . . image ⇝ *object :* real
The albedo of the surface.

▷ **Slants** (input_control) . . . . . angle.deg-array ⇝ *real* / integer
Angle between the camera and the direction of illumination (in degrees).
Default: 45.0
Suggested values: Slants ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Slants ≤ 180.0 (lin)
Minimum increment: 0.01
Recommended increment: 10.0

▷ **Tilts** (input_control) . . . . . angle.deg-array ⇝ *real* / integer
Angle of the direction of illumination within the object plane (in degrees).
Default: 45.0
Suggested values: Tilts ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Tilts ≤ 360.0 (lin)
Minimum increment: 0.01
Recommended increment: 10.0

▷ **ResultType** (input_control) . . . . . string-array ⇝ *string*
Types of the requested results.
Default: 'all'
List of values: ResultType ∈ {[], 'all', 'height_field', 'gradient', 'normalized_surface_normal', 'albedo'}

▷ **ReconstructionMethod** (input_control) . . . . . string ⇝ *string*
Type of the reconstruction method.
Default: 'poisson'
List of values: ReconstructionMethod ∈ {'fft_cyclic', 'rft_cyclic', 'poisson'}

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of the generic parameters.
Default: []
List of values: GenParamName ∈ {'optimize_speed', 'caching'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the generic parameters.
Default: []
List of values: GenParamValue ∈ {'standard', 'patient', 'exhaustive', 'use_cache', 'no_cache', 'free_cache'}

Result

If the parameters are valid, `photometric_stereo` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`optimize_fft_speed`

Module

3D Metrology

reconstruct_height_field_from_gradient (
Gradient : HeightField : ReconstructionMethod, GenParamName,
GenParamValue :)

Reconstruct a surface from surface gradients.

`reconstruct_height_field_from_gradient` reconstructs a surface from the surface gradients that are given in `Gradient`. The surface is returned as a height field, i.e., an image in which the gray value of each image point corresponds to a relative height.

The reconstruction is done by integrating the gradients by different algorithms that can be selected in the parameter `ReconstructionMethod`. Because gradient fields are typically non-integrable due to noise, the various algorithms return a solution in a least-squares sense. The algorithms differ in the way how they model the boundary condition. Currently three algorithms are supported: *'fft_cyclic'*, *'rft_cyclic'* and *'poisson'*.

Reconstruction with Fast Fourier transforms

The variants *'fft_cyclic'* and *'rft_cyclic'* assume that the image function is cyclic at the boundaries. Note that due to the assumed cyclic image function artifacts may occur at the image boundaries. Thus, in most cases, we recommend to use the *'poisson'* algorithm instead.

The difference between *'fft_cyclic'* and *'rft_cyclic'* is that the rft version has faster processing times and requires less memory than the fft version. While theoretically fft and rft should return the same result, the fft version is numerically slightly more accurate. As `reconstruct_height_field_from_gradient` internally uses a fast Fourier transform, the run time of the operator can be influenced by a previous call to `optimize_fft_speed` or `optimize_rft_speed`, respectively.

Reconstruction according to Poisson

The *'poisson'* algorithm assumes that the image has constant gradients at the image border. In most cases, it is the recommended reconstruction method for `reconstruct_height_field_from_gradient`. Its run time can only be optimized by setting `GenParamName` to *'optimize_speed'* and `GenParamValue` to *'standard'*, *'patient'*, or *'exhaustive'*. These parameters are described in more detail with the description of `optimize_fft_speed`.

Note that by default, the *'poisson'* algorithm uses a cache that depends on the image size and that speeds up the reconstruction significantly, provided that all images have the same size. The cache is allocated at the first time when the *'poisson'* algorithm is called. Therefore the first call always takes longer than subsequent calls. The additionally needed memory corresponds to the memory needed for the specific size of one image. Please note that when calling the operator with different image sizes, the cache needs to be reallocated, which leads to a longer processing time. In this case it may be preferable to not use the cache. To switch off the caching, you must set the parameter `GenParamName` to *'caching'* and the parameter `GenParamValue` to *'no_cache'*. The cache can explicitly be deallocated by setting `GenParamName` to *'caching'* and `GenParamValue` to *'free_cache'*. However, in the majority of cases, we recommend to use the cache, i.e., to use the default setting for the parameter *'caching'*.

Saving and loading optimization parameters

The optimization parameters for all algorithms can be saved and loaded by `write_fft_optimization_data` and `read_fft_optimization_data`.

Non obvious applications

Please note that the operator `reconstruct_height_field_from_gradient` has various non-obvious applications, especially in the field called gradient domain manipulation technique. In many applications, the gradient values that are passed as input to the operator do not have the semantics of surface gradients (i.e., the first derivatives of the height values), but are rather the first derivatives of other kinds of parameters, typically gray values (then, the gradients have the semantics of gray value edges). When processing these gradient images by various means, e.g., by adding or subtracting images, or by a filtering, the original gradient values are altered and the subsequent call to `reconstruct_height_field_from_gradient` delivers a modified image, in which, e.g., unwanted edges are removed or the contrast has been changed locally. Typical applications are noise removal, seamless fusion of images, or high dynamic range compression.

Attention

`reconstruct_height_field_from_gradient` takes into account the values of all pixels in `Gradient`, not only the values within its domain. If `Gradient` does not have a full domain, one could cut out the relevant square part of the gradient field and generate a smaller image with full domain.

Parameters

- ▷ **`Gradient`** (input_object) ..... singlechannelimage ⇝ *object :* vector_field
  The gradient field of the image.
- ▷ **`HeightField`** (output_object) ..... image ⇝ *object :* real
  Reconstructed height field.
- ▷ **`ReconstructionMethod`** (input_control) ..... string ⇝ *string*
  Type of the reconstruction method.
  Default: 'poisson'
  List of values: `ReconstructionMethod` ∈ {'fft_cyclic', 'rft_cyclic', 'poisson'}
- ▷ **`GenParamName`** (input_control) ..... string-array ⇝ *string*
  Names of the generic parameters.
  Default: []
  List of values: `GenParamName` ∈ {'optimize_speed', 'caching'}
- ▷ **`GenParamValue`** (input_control) ..... integer-array ⇝ *integer* / real / string
  Values of the generic parameters.
  Default: []
  List of values: `GenParamValue` ∈ {'standard', 'patient', 'exhaustive', 'use_cache', 'no_cache', 'free_cache'}

Result

If the parameters are valid `reconstruct_height_field_from_gradient` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

References

M. Kazhdan, M. Bolitho, and H. Hoppe: "Poisson Surface Reconstruction." Symposium on Geometry Processing (June 2006).

Module

3D Metrology

`sfs_mod_lr` `( Image : Height : Slant, Tilt, Albedo, Ambient : )`

Reconstruct a surface from a gray value image.

sfs_mod_lr reconstructs a surface (i.e. the relative height of each image point) using the modified algorithm of Lee and Rosenfeld. The surface is reconstructed from the input image Image, and the light source given by the parameters Slant, Tilt, Albedo and Ambient, and is assumed to lie infinitely far away in the direction given by Slant and Tilt. The parameter Albedo determines the albedo of the surface, i.e. the percentage of light reflected in all directions. Ambient determines the amount of ambient light falling onto the surface. It can be set to values greater than zero if, for example, the white balance of the camera was badly adjusted at the moment the image was taken.

Attention

sfs_mod_lr assumes that the heights are to be extracted on a lattice with step width 1. If this is not the case, the calculated heights must be multiplied with the step width after the call to sfs_mod_lr. A Cartesian coordinate system with the origin in the lower left corner of the image is used internally. sfs_mod_lr can only handle byte-images.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Shaded input image.

▷ **Height** (output_object) . . . . . image(-array) ⇝ *object :* real
Reconstructed height field.

▷ **Slant** (input_control) . . . . . angle.deg ⇝ *real* / integer
Angle between the light source and the positive z-axis (in degrees).
Default: 45.0
Suggested values: Slant ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Slant ≤ 180.0 (lin)
Minimum increment: 0.01
Recommended increment: 10.0

▷ **Tilt** (input_control) . . . . . angle.deg ⇝ *real* / integer
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).
Default: 45.0
Suggested values: Tilt ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Tilt ≤ 360.0 (lin)
Minimum increment: 0.01
Recommended increment: 10.0

▷ **Albedo** (input_control) . . . . . number ⇝ *real* / integer
Amount of light reflected by the surface.
Default: 1.0
Suggested values: Albedo ∈ {0.1, 0.5, 1.0, 5.0}
Value range: 0.0 ≤ Albedo ≤ 5.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Albedo >= 0.0

▷ **Ambient** (input_control) . . . . . number ⇝ *real* / integer
Amount of ambient light.
Default: 0.0
Suggested values: Ambient ∈ {0.1, 0.5, 1.0}
Value range: 0.0 ≤ Ambient ≤ 1.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Ambient >= 0.0

Result

If all parameters are correct sfs_mod_lr returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

estimate_al_am, estimate_sl_al_lr, estimate_sl_al_zc, estimate_tilt_lr, estimate_tilt_zc, optimize_fft_speed

Possible Successors

shade_height_field

Module

3D Metrology

sfs_orig_lr (Image : Height : Slant, Tilt, Albedo, Ambient :)

Reconstruct a surface from a gray value image.

sfs_orig_lr reconstructs a surface (i.e. the relative height of each image point) using the original algorithm of Lee and Rosenfeld. The surface is reconstructed from the input image Image. The light source is to be given by the parameters Slant, Tilt, Albedo and Ambient, and is assumed to lie infinitely far away in the direction given by Slant and Tilt. The parameter Albedo determines the albedo of the surface, i.e. the percentage of light reflected in all directions. Ambient determines the amount of ambient light falling onto the surface. It can be set to values greater than zero if, for example, the white balance of the camera was badly adjusted at the moment the image was taken.

Attention

sfs_orig_lr assumes that the heights are to be extracted on a lattice with step width 1. If this is not the case, the calculated heights must be multiplied with the step width after the call to sfs_orig_lr. A Cartesian coordinate system with the origin in the lower left corner of the image is used internally. sfs_orig_lr can only handle byte-images.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Shaded input image.

▷ **Height** (output_object) . . . . . image(-array) ⇝ *object :* real
Reconstructed height field.

▷ **Slant** (input_control) . . . . . angle.deg ⇝ *real* / integer
Angle between the light source and the positive z-axis (in degrees).
Default: 45.0
Suggested values: Slant ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Slant ≤ 90.0
Minimum increment: 0.01
Recommended increment: 10.0

▷ **Tilt** (input_control) . . . . . angle.deg ⇝ *real* / integer
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).
Default: 45.0
Suggested values: Tilt ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Tilt ≤ 360.0
Minimum increment: 0.01
Recommended increment: 10.0

▷ **Albedo** (input_control) . . . . . number ⇝ *real* / integer
Amount of light reflected by the surface.
Default: 1.0
Suggested values: Albedo ∈ {0.1, 0.5, 1.0, 5.0}
Value range: 0.0 ≤ Albedo ≤ 5.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Albedo >= 0.0

▷ **Ambient** (input_control) . . . . . number ⇝ *real* / integer
Amount of ambient light.
Default: 0.0
Suggested values: Ambient ∈ {0.1, 0.5, 1.0}
Value range: 0.0 ≤ Ambient ≤ 1.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Ambient >= 0.0

Result

If all parameters are correct sfs_orig_lr returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

estimate_al_am, estimate_sl_al_lr, estimate_sl_al_zc, estimate_tilt_lr, estimate_tilt_zc, optimize_fft_speed

Possible Successors

shade_height_field

Module

3D Metrology

sfs_pentland (Image : Height : Slant, Tilt, Albedo, Ambient :)

Reconstruct a surface from a gray value image.

sfs_pentland reconstructs a surface (i.e. the relative height of each image point) using the algorithm of Pentland. The surface is reconstructed from the input image Image. The light source must be given by the parameters Slant, Tilt, Albedo and Ambient, and is assumed to lie infinitely far away in the direction given by Slant and Tilt. The parameter Albedo determines the albedo of the surface, i.e. the percentage of light reflected in all directions. Ambient determines the amount of ambient light falling onto the surface. It can be set to values greater than zero if, for example, the white balance of the camera was badly adjusted at the moment the image was taken.

Attention

sfs_pentland assumes that the heights are to be extracted on a lattice with step width 1. If this is not the case, the calculated heights must be multiplied with the step width after the call to sfs_pentland. A Cartesian coordinate system with the origin in the lower left corner of the image is used internally. sfs_pentland can only handle byte-images.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Shaded input image.

▷ **Height** (output_object) . . . . . image(-array) ⇝ *object :* real
Reconstructed height field.

▷ **Slant** (input_control) . . . . . angle.deg ⇝ *real* / integer
Angle between the light source and the positive z-axis (in degrees).
Default: 45.0
Suggested values: Slant ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Slant ≤ 180.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Tilt** (input_control) . . . . . angle.deg ⇝ *real* / integer
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).
Default: 45.0
Suggested values: Tilt ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Tilt ≤ 360.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Albedo** (input_control) . . . . . number ⇝ *real* / integer
Amount of light reflected by the surface.
Default: 1.0
Suggested values: Albedo ∈ {0.1, 0.5, 1.0, 5.0}
Value range: 0.0 ≤ Albedo ≤ 5.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Albedo >= 0.0

▷ **Ambient** (input_control) . . . . . number ⇝ *real* / integer
Amount of ambient light.
Default: 0.0
Suggested values: Ambient ∈ {0.1, 0.5, 1.0}
Value range: 0.0 ≤ Ambient ≤ 1.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Ambient >= 0.0

Result

If all parameters are correct sfs_pentland returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

estimate_al_am, estimate_sl_al_lr, estimate_sl_al_zc, estimate_tilt_lr, estimate_tilt_zc, optimize_fft_speed

Possible Successors

shade_height_field

Module

3D Metrology

shade_height_field (ImageHeight : ImageShade : Slant, Tilt, Albedo, Ambient, Shadows :)

Shade a height field.

shade_height_field computes a shaded image from the height field ImageHeight as if the image were illuminated by an infinitely far away light source. It is assumed that the surface described by the height field has Lambertian reflection properties determined by Albedo and Ambient. The parameter Shadows determines whether shadows are to be calculated.

Attention

shade_height_field assumes that the heights are given on a lattice with step width 1. If this is not the case, the heights must be divided by the step width before the call to shade_height_field. Otherwise, the derivatives used internally to compute the orientation of the surface will be estimated to steep or too flat. Example: The height field is given on 100*100 points on the square [0,1]*[0,1]. Then the heights must be divided by 1/100 first. A Cartesian coordinate system with the origin in the lower left corner of the image is used internally.

Parameters

▷ **ImageHeight** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / int4 / real
Height field to be shaded.

▷ **ImageShade** (output_object) . . . . . image(-array) ⇝ *object :* byte
Shaded image.

▷ **Slant** (input_control) . . . . . angle.deg ⇝ *real* / integer
Angle between the light source and the positive z-axis (in degrees).
Default: 0.0
Suggested values: Slant ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Slant ≤ 180.0 (lin)
Minimum increment: 0.01
Recommended increment: 10.0

▷ **Tilt** (input_control) . . . . . angle.deg ⇝ *real* / integer
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).
Default: 0.0
Suggested values: Tilt ∈ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Tilt ≤ 360.0 (lin)
Minimum increment: 0.01
Recommended increment: 10.0

▷ **Albedo** (input_control) . . . . . number ⇝ *real* / integer
Amount of light reflected by the surface.
Default: 1.0
Suggested values: Albedo ∈ {0.1, 0.5, 1.0, 5.0}
Value range: 0.0 ≤ Albedo ≤ 5.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Albedo >= 0.0

▷ **Ambient** (input_control) . . . . . number ⇝ *real* / integer
Amount of ambient light.
Default: 0.0
Suggested values: Ambient ∈ {0.1, 0.5, 1.0}
Value range: 0.0 ≤ Ambient ≤ 1.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Ambient >= 0.0

▷ **Shadows** (input_control) . . . . . string ⇝ *string*
Should shadows be calculated?
Default: 'false'
Suggested values: Shadows ∈ {'true', 'false'}

Result

If all parameters are correct shade_height_field returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

sfs_mod_lr, sfs_orig_lr, sfs_pentland, photometric_stereo

Module

Foundation

uncalibrated_photometric_stereo (Images : NormalField, Gradient, Albedo : ResultType :)

Reconstruct a surface from several, differently illuminated images.

uncalibrated_photometric_stereo can be used to extract high-frequency surface details from a given object with no prior knowledge about the illumination, geometry and reflectance of the object. The geometry of interest can be for example dents, folds or scratches. The operator can usually not be used for measuring the overall shape of an object. The operator returns the normals NormalField of the surface as a 3-channel image with each image encoding a component of the normal. This is used as a visualization of the result as a color coded image. Further, it returns the Gradient and the Albedo of the surface. Which result should be calculated can be controlled with ResultType. This operator is related to photometric_stereo, but does not require known (i.e. previously calibrated) light directions. Note that photometric_stereo is faster and more accurate, but needs the light direction information. For sensible results an orthographic projection of the camera is assumed for both the calibrated and uncalibrated case. This is typically reached by using a telecentric lens or at least a lens with a long focal distance.

The operator requires at least three images of the same object, taken with a static, non-moving camera and different lighting directions for each image. For best results, the object should exhibit Lambertian reflection properties, no inter-reflection or shadow castings.

Parameters

- ▷ **Images** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2
  The input images with different illumination.
- ▷ **NormalField** (output_object) ..... image(-array) ⇝ *object :* real
  The normal field of the surface.
- ▷ **Gradient** (output_object) ..... image ⇝ *object :* vector_field
  The gradient field of the surface .
- ▷ **Albedo** (output_object) ..... image ⇝ *object :* real
  The albedo of the surface.
- ▷ **ResultType** (input_control) ..... string-array ⇝ *string*
  The result type.
  Default: 'all'
  List of values: ResultType ∈ {[], 'all', 'normal_field', 'gradient', 'normalized_gradient', 'albedo'}

Example

```
* read severally illuminated images
FName := 'photometric_stereo/pharma_braille_0' + [1:4] + '.png'
read_image(Images, FName)
* extract surface normals, gradients and albedo from images
uncalibrated_photometric_stereo(Images, NormalField, Gradient, Albedo, 'all')
derivate_vector_field (Gradient, Result, 0.1, 'mean_curvature')
reconstruct_height_field_from_gradient (Gradient, HeightField, 'poisson', \
                                        [], [])
```

Result

The operator uncalibrated_photometric_stereo returns the NormalField for the given images as well as the appropriate gradients for each pixel and the Albedo of the object.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

photometric_stereo

See also

photometric_stereo

References

H. Hayakawa: “Photometric stereo under a light source with arbitrary motion”. Journal Optical Society America, Vol. 11, No. 11/November 1994.

Module

3D Metrology

5.5 Sheet of Light

apply_sheet_of_light_calibration (
Disparity : : SheetOfLightModelID :)

Apply the calibration transformations to the input disparity image.

The operator apply_sheet_of_light_calibration reads the disparity image Disparity, stores it to the sheet-of-light model specified by SheetOfLightModelID and applies the calibration transformation to this image, in order to compute the calibrated coordinates of the reconstructed 3D surface points. The resulting calibrated coordinates can be retrieved from the model by using the operator get_sheet_of_light_result. The corresponding 3D object model can be retrieved with get_sheet_of_light_result_object_model_3d. Note that prior to the next call of apply_sheet_of_light_calibration for a disparity image of smaller height, reset_sheet_of_light_model should be called.

The disparity image Disparity may have been acquired previously by using the operator measure_profile_sheet_of_light or by an image acquisition device, which directly provides disparity values and works according to the sheet-of-light technique.

In order to compute the calibrated coordinates, the parameters listed below must have been set for the sheet-of-light model with the help of the operator set_sheet_of_light_param:

'calibration': extent of the calibration transformation which shall be applied to the disparity image. *'calibration'* must be set to *'xz'*, *'xyz'* or *'offset_scale'*. Refer to set_sheet_of_light_param for details on this parameter.

'camera_parameter': the internal parameters of the camera used for the measurement. This pose is required when the calibration extent has been set to *'xyz'* or *'xz'*.

'camera_pose': the pose of the world coordinate system relative to the camera coordinate system. This pose is required when the calibration extent has been set to *'xyz'* or *'xz'*.

'lightplane_pose': the pose of the light-plane coordinate system relative to the world coordinate system. The light-plane coordinate system must be chosen so that its plane z=0 coincides with the light plane described by the light line projector. This pose is required when the calibration extent has been set to *'xyz'* or *'xz'*.

'movement_pose': a pose representing the movement of the object between two successive profile images with respect to the measurement system built by the camera and the laser. This pose is required when the calibration extent has been set to *'xyz'*. It is ignored when the calibration extent has been set to *'xz'*.

'scale': with this parameter you can scale the 3D coordinates X, Y and Z that result when applying the calibration transformations to the disparity image. *'scale'* must be specified as the ratio *desired unit/original unit*. The original unit is determined by the coordinates of the calibration object. If the original unit is meters (which is the case if you use the standard calibration plate), you can set the desired unit directly by selecting 'm', 'cm', 'mm' or 'um' for the parameter Scale. By default, *'scale'* is set to 1.0.

Parameters

▷ **Disparity** (input_object) ..... singlechannelimage ⇝ *object : real*
Height or range image to be calibrated.

▷ **SheetOfLightModelID** (input_control) ..... sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

Example

```
* ...
* Read an already acquired disparity map from file
read_image (Disparity, 'sheet_of_light/connection_rod_disparity.tif')
*
* Create a model and set the required parameters
gen_rectangle1 (ProfileRegion, 120, 75, 195, 710)
create_sheet_of_light_model (ProfileRegion, ['min_gray','num_profiles', \
                             'ambiguity_solving'], [70,290,'first'], \
                              SheetOfLightModelID)
set_sheet_of_light_param (SheetOfLightModelID, 'calibration', 'xyz')
set_sheet_of_light_param (SheetOfLightModelID, 'scale', 'mm')
set_sheet_of_light_param (SheetOfLightModelID, 'camera_parameter', \
                          CameraParameter)
set_sheet_of_light_param (SheetOfLightModelID, 'camera_pose', CameraPose)
set_sheet_of_light_param (SheetOfLightModelID, 'lightplane_pose', \
                          LightPlanePose)
set_sheet_of_light_param (SheetOfLightModelID, 'movement_pose', \
                          MovementPose)
*
* Apply the calibration transforms and
* get the resulting calibrated coordinates
apply_sheet_of_light_calibration (Disparity, SheetOfLightModelID)
get_sheet_of_light_result (X, SheetOfLightModelID, 'x')
get_sheet_of_light_result (Y, SheetOfLightModelID, 'y')
get_sheet_of_light_result (Z, SheetOfLightModelID, 'z')
*
```

Result

The operator `apply_sheet_of_light_calibration` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SheetOfLightModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Successors

`get_sheet_of_light_result`, `get_sheet_of_light_result_object_model_3d`

Module

3D Metrology

calibrate_sheet_of_light (: : SheetOfLightModelID : Error)

Calibrate a sheet-of-light setup with a 3D calibration object.

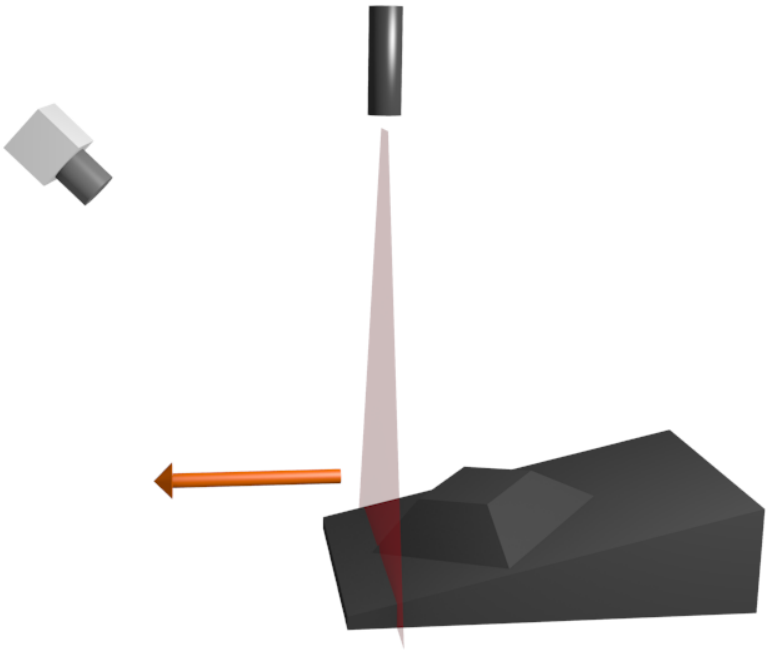
`calibrate_sheet_of_light` calibrates the sheet-of-light setup `SheetOfLightModelID` from one disparity image of a 3D calibration object and returns the back projection error of the optimization in `Error`.

Overview

The calibration of a sheet-of-light setup with `calibrate_sheet_of_light` is simpler than the calibration of a sheet-of-light setup with standard HALCON calibration plates, which is shown in the HDevelop example

calibrate_sheet_of_light_calplate.hdev. It is only necessary to obtain one uncalibrated reconstruction, i.e., a disparity image, of a special 3D calibration object to calibrate the sheet-of-light model.

In the following, the steps that are necessary for the calibration are described.



Calibration of a sheet-of-light setup

Supply of a 3D calibration object

A special 3D calibration object must be provided. This calibration object must correspond to the CAD model created with create_sheet_of_light_calib_object. The 3D calibration object has an inclined plane on which a truncated pyramid is located. It has a thinner side, which is hereinafter referred to as front side. The thicker side is referred to as back side of the calibration object.

The dimensions of the calibration object should be chosen such that the calibration object covers the complete measuring volume. Be aware, that only parts on the 3D calibration object above HeightMin (see create_sheet_of_light_calib_object) are taken into account.

The CAD model, which is written as a DXF file, also serves as description file of the calibration object.

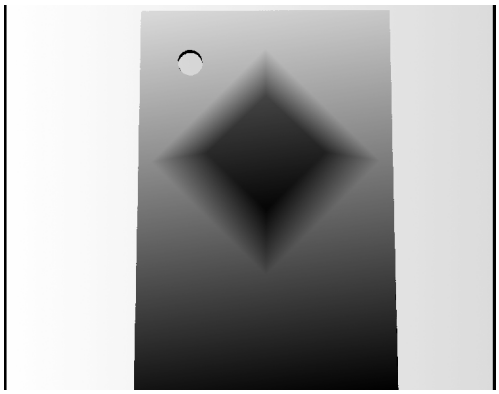
Preparation of the sheet-of-light model

To prepare a sheet-of-light model for the calibration, the following steps must be performed.

- Create a sheet-of-light model with create_sheet_of_light_model and adapt the default parameters to your specific measurement task.
- Set the initial parameters of the camera with set_sheet_of_light_param. So far, only pinhole cameras with the division model are supported, i.e., only cameras of type *'area_scan_division'*.
- Set the description file of the calibration object (created with create_sheet_of_light_calib_object) with set_sheet_of_light_param.

Uncalibrated reconstruction of the 3D calibration object

The 3D calibration object must be reconstructed with the (uncalibrated) sheet-of-light model prepared above, i.e., a disparity image of the 3D calibration object must be created.



Disparity image of a calibration object

For this, the calibration object must be oriented such that either its front side or its back side intersect the light plane first (i.e., the movement vector should be parallel to the Y axis of the calibration object, see create_sheet_of_light_calib_object). As far as possible, the domain of the disparity image of the calibration object should be restricted to the calibration object. Besides, the domain of the disparity image should have no holes on the truncated pyramid. All four sides of the truncated pyramid must be clearly visible.

Calibration of the sheet-of-light setup

The calibration is then performed with calibrate_sheet_of_light. The returned Error is the RMS of the distance of the reconstructed points to the calibration object in meters.

For sheet-of-light models calibrated with calibrate_sheet_of_light, in rare cases the parameters might yield an unrealistic setup. However, the quality of measurements performed with the calibrated parameters is not affected.

Parameters

▷ **SheetOfLightModelID** (input_control) ..... sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

▷ **Error** (output_control) ..... number ⇝ *real*
Average back projection error of the optimization.

Example

```
* Calibrate a sheet-of-light model with a 3D calibration object
gen_rectangle1 (Rectangle, 300, 0, 800, 1023)
CameraParam := ['area_scan_division', 0.016, 0, 4.65e-6, 4.65e-6, \
                640.0, 512.0, 1280, 1024]
create_sheet_of_light_model (Rectangle, 'min_gray', 50, SheetOfLightModelID)
set_sheet_of_light_param (SheetOfLightModelID, 'camera_parameter', \
                          CameraParam)
set_sheet_of_light_param (SheetOfLightModelID, 'calibration_object', \
                         'calib_object.dxf')
* Uncalibrated reconstruction of the calibration object
for ProfileIndex := 1 to 1000 by 1
    grab_image_async (Image, AcqHandle, -1)
    measure_profile_sheet_of_light (Image, SheetOfLightModelID, [])
endfor
* Calibration of the sheet-of-light-model
calibrate_sheet_of_light (SheetOfLightModelID, Error)
* Now get a calibrated reconstruction of the calibration object
get_sheet_of_light_result_object_model_3d (SheetOfLightModelID, \
                                           ObjectModel3D)
```

Result

The operator calibrate_sheet_of_light returns the value 2 (H_MSG_TRUE) if the calibration was successful. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- SheetOfLightModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_sheet_of_light_model`, `set_sheet_of_light_param`, `set_profile_sheet_of_light`, `measure_profile_sheet_of_light`

Possible Successors

`set_profile_sheet_of_light`, `apply_sheet_of_light_calibration`

Module

3D Metrology

clear_sheet_of_light_model (: : SheetOfLightModelID :)

Delete a sheet-of-light model and free the allocated memory.

The operator `clear_sheet_of_light_model` deletes a sheet-of-light model that was created by `create_sheet_of_light_model`. All memory used by the model is freed. The handle of the model is passed in `SheetOfLightModelID`. After the operator call it is invalid.

Parameters

▷ **SheetOfLightModelID** (input_control) ..... sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

Result

The operator `clear_sheet_of_light_model` returns the value `2` (`H_MSG_TRUE`) if a valid handle is passed and the referred sheet-of-light model can be freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SheetOfLightModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

`create_sheet_of_light_model`

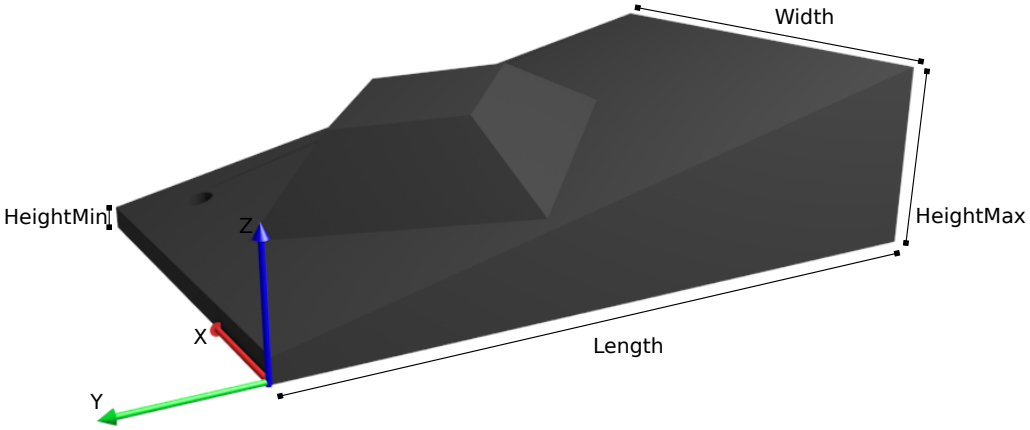
Module

3D Metrology

create_sheet_of_light_calib_object (: : Width, Length, HeightMin, HeightMax, FileName :)

Create a calibration object for sheet-of-light calibration.

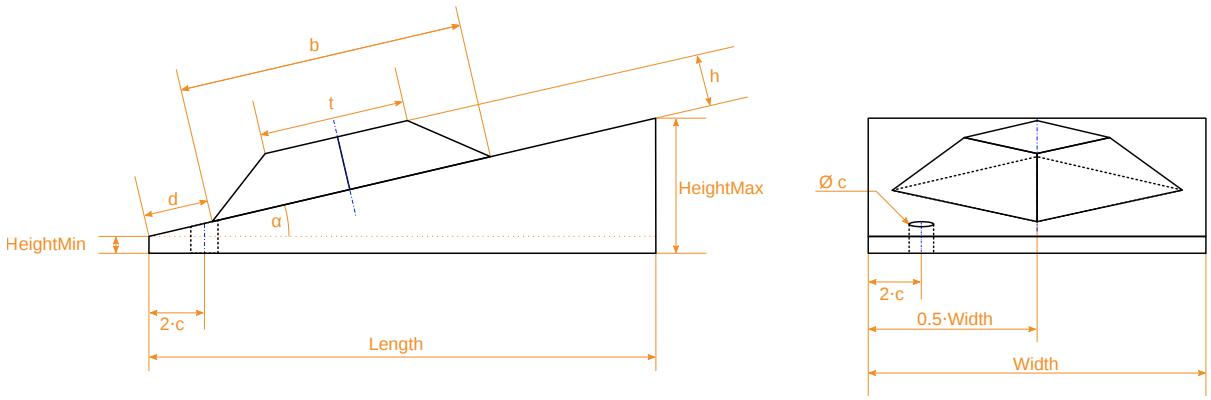
`create_sheet_of_light_calib_object` creates a CAD model of a calibration object for sheet-of-light calibration with `calibrate_sheet_of_light` and stores it in `FileName`.



A calibration object for sheet-of-light calibration

The calibration object consists of a ramp with a truncated pyramid rotated by 45 degrees. The calibration object contains an orientation mark in the form of a circular hole. The dimensions of the calibration target in `Width`, `Length`, `HeightMin`, and `HeightMax` must be given in meters. `Length` must be at least 10% larger than `Width`. The Z coordinate of the highest point on the truncated pyramid is at most `HeightMax`. The calibration object might not be found by `calibrate_sheet_of_light` if the height difference between the truncated pyramid and the ramp is too small. In this case, adjust `HeightMin` and `HeightMax` accordingly or increase the sampling rate when acquiring the calibration data.

The dimensions of the calibration object should be chosen such that it is possible to cover the measuring volume of the sheet-of-light setup. In addition, when selecting the `Length` of the calibration object, the speed of the sheet-of-light setup should be considered such that the calibration object is sampled with enough profile measurements.



Technical drawing of the calibration object, where `c` is the diameter of the orientation mark, `d` is the distance of the pyramid from the front of the calibration object, `h` is the height of the truncated pyramid, `b` is the length of the diagonal of the pyramid at the bottom, `t` is the corresponding length at the top, and α is the angle of the ramp as seen in the drawing. You can calculate these dimensions with the procedure `get_sheet_of_light_calib_object_dimensions`.

Set the parameter *'calibration_object'* to `FileName` with `set_sheet_of_light_param` to use the generated calibration object in a subsequent call to `calibrate_sheet_of_light`.

Note that MVTec does not offer 3D calibration objects. Instead, use `create_sheet_of_light_calib_object` to generate a customized CAD model of a calibration object. This CAD model can then be used to produce the calibration object. Milled aluminum is an established material for this. However, depending on the required precision, its thermal stability may be a problem. Note that the surface should be bright. Its color may have to be adjusted depending on the color of the laser to provide a

sufficient contrast to the color of the laser. Additionally, the surface must not be translucent nor reflective. To achieve this, you can anodize or lacquer it. Please note that when lacquering it, the accuracy might be decreased due to the applied paintwork. However, a surface that is too rough leads to a decreasing precision as well. It is advisable to have the produced calibration object remeasured to determine whether the required accuracy can be achieved. The accuracy of the calibration object should be ten times higher than the required accuracy of measurement. After having the object measured, the results can be manually inserted into the DXF file that can then be used for the calibration with `calibrate_sheet_of_light`.

Parameters

▷ **`Width`** (input_control) . . . . . number ⇝ *real*
Width of the object.
Default: 0.1

▷ **`Length`** (input_control) . . . . . number ⇝ *real*
Length of the object.
Default: 0.15

▷ **`HeightMin`** (input_control) . . . . . number ⇝ *real*
Minimum height of the ramp.
Default: 0.005

▷ **`HeightMax`** (input_control) . . . . . number ⇝ *real*
Maximum height of the ramp.
Default: 0.04

▷ **`FileName`** (input_control) . . . . . filename.write ⇝ *string*
Filename of the model of the calibration object.
Default: 'calib_object.dxf'
File extension: `.dxf`

Result

The operator `create_sheet_of_light_calib_object` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

3D Metrology

`create_sheet_of_light_model` `( ProfileRegion : : GenParamName, GenParamValue : SheetOfLightModelID )`

Create a model to perform 3D-measurements using the sheet-of-light technique.

The operator `create_sheet_of_light_model` creates a model to perform 3D-Measurements using the sheet-of-light technique.

The sheet-of-light technique performs a three-dimensional reconstruction of the surface of an opaque and diffuse reflecting solid by using an area scan camera and a light line projector (typically a laser line projector). The camera and the line projector must be mounted so that their main axis form an angle of triangulation. The value of the angle of triangulation is typically chosen between 30° and 60°. The projected light line defines a plane in space. This plane intersects the surface of the solid under measurement and builds a profile of the surface visible for the camera. By moving the solid in front of the measurement system (i.e., the combination of the camera and the line projector), it is possible to record the whole surface of the solid. As an alternative, the measurement system can also be moved over the surface under measurement. Please note that the profiles must be oriented roughly horizontal in the profile images, because they are processed column by column.

If geometrical information about the measurement setup is available, it is possible to compute true three-dimensional coordinates of the reconstructed surface. For an overview of the required geometrical (i.e., calibration)

information, refer to the operator set_sheet_of_light_param. If such information is not available, the result of the measurement is a disparity image, where each pixel holds a record of the subpixel precise position of the detected profile.

The operator returns a handle to the sheet-of-light model in SheetOfLightModelID, which is used for all further operations on the sheet-of-light model, like modifying parameters of the model, measuring profiles, applying calibration transformations or accessing the results of measurements.

Mandatory input iconic parameters

In order to perform measurements, you will have to set the following input iconic parameter:

ProfileRegion: defines the region of the profile images, which will be processed by the operator measure_profile_sheet_of_light. This region should be rectangular and can be generated e.g., by using the operator gen_rectangle1. If the region passed to ProfileRegion is not rectangular, its smallest enclosing rectangle (bounding box) will be used. Note that ProfileRegion is only taken into account by the operator measure_profile_sheet_of_light and is ignored when disparity images are processed.

Default settings of the sheet-of-light model parameters

The default settings of the sheet-of-light model were chosen to perform non-calibrated measurements in a basic configuration. The following list provides an overview of the parameter values used by default (refer to set_sheet_of_light_param for a detailed description of all supported generic parameters):

'method' is set to *'center_of_gravity'*

'min_gray': is set to 100

'num_profiles' is set to 512

'ambiguity_solving' is set to *'first'*

'score_type' is set to *'none'*

'calibration' is set to *'none'*

Modify the sheet-of-light model parameters

We recommend to adapt the default parameters to your specific measurement task, in order to enhance the quality of the measurement or to shorten the runtime. You will also have to modify the default values of the model parameters if you need calibrated results.

create_sheet_of_light_model provides the generic parameters GenParamName and GenParamValue to modify the default value of most of the model parameters. Note that model parameters can also be set by using the operator set_sheet_of_light_param. Nevertheless, with this second operator only one parameter can be set at the same time, whereas it is possible to set more than one parameter at the time with create_sheet_of_light_model. Refer to set_sheet_of_light_param for a detailed description of all supported generic parameters.

Please note that the following model parameters can not be set with the operator create_sheet_of_light_model, and thus have to be set with the operator set_sheet_of_light_param: *'camera_parameter'*, *'camera_pose'*, *'lightplane_pose'*, and *'movement_pose'*.

It is possible to query the value of the model parameters with the operator get_sheet_of_light_param. The names of all supported model parameters are returned by the operator query_sheet_of_light_params.

Use the simplified sheet-of-light model parameters

In case of a simple setup or if not a real metric calibration is necessary, the transformation of the observed disparities into 3D values can be controlled using a simplified parameter set of the sheet-of-light model:

By setting the calibration with the set_sheet_of_light_param to *'offset_scale'*, the poses and camera parameter are changed to such values, that an offset of one pixel corresponds to one unit in the 3D result. This allows to create a 3D object model and 3D images from an uncalibrated sheet-of-light model.

The transformation from disparity to 3D coordinates can be controlled by six parameters: *'scale_x'*, *'scale_y'*, *'scale_z'*, *'offset_x'*, *'offset_y'*, *'offset_z'*. Refer to set_sheet_of_light_param for a detailed description of all supported generic parameters.

Use of a handle in multiple threads

Please note that you have to take special care when using a handle of a sheet-of-light-model SheetOfLightModelID in multiple threads. One and the same handle cannot be used concurrently in different threads if they modify the handle. Thus, you have to be careful especially if the threads call operators that change the data of the handle. You can find an according hint in the *'Attention'* section of the operators. Anyway, if you still want to use the same handle in operators that concurrently write into the handle in different threads you have to synchronize the threads to assure that they do not access the same handle simultaneously. If you are not sure if the usage of the same handle is thread-safe, please see the *'Attention'* section of the respective reference manual entry if it contains a warning pointing to this problem. However, different handles can be used independently and safely in different threads.

Parameters

- ▷ **ProfileRegion** (input_object) ..... region ⇝ *object*
  Region of the images containing the profiles to be processed. If the provided region is not rectangular, its smallest enclosing rectangle will be used.
- ▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
  Names of the generic parameters that can be adjusted for the sheet-of-light model.
  Default: 'min_gray'
  List of values: GenParamName ∈ {'min_gray', 'method', 'ambiguity_solving', 'score_type', 'num_profiles', 'calibration', 'scale', 'scale_x', 'scale_y', 'scale_z', 'offset_x', 'offset_y', 'offset_z'}
- ▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *integer* / real / string
  Values of the generic parameters that can be adjusted for the sheet-of-light model.
  Default: 50
  Suggested values: GenParamValue ∈ {'default', 'center_of_gravity', 'last', 'first', 'brightest', 'none', 'intensity', 'width', 'offset_scale', 50, 100, 150, 180}
- ▷ **SheetOfLightModelID** (output_control) ..... sheet_of_light_model ⇝ *handle*
  Handle for using and accessing the sheet-of-light model.

Example

```
* Create the rectangular region in which the profiles are measured.
gen_rectangle1 (ProfileRegion, 120, 75, 195, 710)
*
* Create a model in order to measure profiles according to
* the sheet-of-light technique. Simultaneously set some
* parameters for the model.
create_sheet_of_light_model (ProfileRegion, ['min_gray','num_profiles', \
                             'ambiguity_solving','score_type'], \
                             [70,290,'first','width'], \
  SheetOfLightModelID)
*
* Measure the profile from successive images
for Index := 1 to 290 by 1
    read_image (ProfileImage, 'sheet_of_light/connection_rod_'+Index$'.3')
    dev_display (ProfileImage)
    dev_display (ProfileRegion)
    measure_profile_sheet_of_light (ProfileImage, SheetOfLightModelID, [])
endfor
*
* Get the resulting disparity and score images
get_sheet_of_light_result (Disparity, SheetOfLightModelID, 'disparity')
get_sheet_of_light_result (Score, SheetOfLightModelID, 'score')
*
* Close the sheet-of-light handle once the measurement
* has been performed
```

Result

The operator create_sheet_of_light_model returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

gen_rectangle1

Possible Successors

set_sheet_of_light_param, measure_profile_sheet_of_light

See also

clear_sheet_of_light_model, calibrate_sheet_of_light

Module

3D Metrology

deserialize_sheet_of_light_model (: : SerializedItemHandle : SheetOfLightModelID)

Deserialize a sheet-of-light model.

deserialize_sheet_of_light_model deserializes a sheet-of-light model that was serialized by serialize_sheet_of_light_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized model is defined by the handle SerializedItemHandle. The deserialized values are stored in a new sheet-of-light model with the handle SheetOfLightModelID.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **SheetOfLightModelID** (output_control) ..... sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

Result

The operator deserialize_sheet_of_light_model returns the value 2 (H_MSG_TRUE) if the sheet-of-light model can be correctly deserialized. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_sheet_of_light_model

Possible Successors

measure_profile_sheet_of_light

Alternatives

create_sheet_of_light_model

See also

serialize_sheet_of_light_model

Module

3D Metrology

get_sheet_of_light_param (: : SheetOfLightModelID, GenParamName : GenParamValue)

Get the value of a parameter, which has been set in a sheet-of-light model.

The operator get_sheet_of_light_param is used to query the values of the different parameters of a sheet-of-light model. The names of the desired parameters are passed in the generic parameter GenParamName, the corresponding values are returned in GenParamValue. All these parameters can be set and changed at any time with the operator set_sheet_of_light_param.

It is not possible to query the values of several parameters with a single operator call. In order to request the values of several parameters you have to successively call of the operator get_sheet_of_light_param.

The values of the following model parameters can be queried:

Measurement of the profiles:

'method': defines the method used to determine the position of the profile. The values *'default'* and *'center_of_gravity'* both refer to the same method, whereby the position of the profile is determined column by column with subpixel accuracy by computing the center of gravity of the gray values g_i of all pixels fulfilling the condition:

$$g_i \geq 'min\_gray'$$

'min_gray': the smallest gray values taken into account for the measurement of the position of the profile (see *'method'* above).

'num_profiles': number of profiles for which memory has been allocated within the sheet-of-light model. By default, *'num_profiles'* is set to 512. If this number of profiles is exceeded during the measurement, memory will be reallocated automatically at runtime. Since the reallocation process requires some time, we recommend to set *'num_profiles'* to a reasonable value before the measurement is started.

'ambiguity_solving': this model parameter determines which candidate shall be chosen, if the determination of the position of the light line is ambiguous.

'first': the first encountered candidate is returned. This method is the fastest.

'last': the last encountered candidate is returned.

'brightest': for each candidate, the brightness of the profile is computed and the candidate having the highest brightness is returned. The brightness is computed according to:

$$brightness = \frac{1}{n} \sum_{i=0}^{n} g_i \ ,$$

where g_i is the gray value of the pixel and n the number of pixels taken into consideration to determine the position of the profile.

'score_type': this model parameter selects which type of score will be calculated during the measurement of the disparity. The score values give an advice on the quality of the computed disparity.

'none': no score is computed.

'width': for each pixel of the disparity, a score value is set to the local width of the profile (i.e., the number of pixels used to compute the position of the profile).

'intensity': for each pixel of the disparity, a score value is evaluated by computing the local intensity of the profile according to:

$$score = \frac{1}{n} \sum_{i=0}^{n} g_i$$

where g_i is the gray value of the pixel and n the number of pixels taken into consideration to determine the position of the profile.

Calibration of the measurement:

'calibration': extent of the calibration transformation which shall be applied to the disparity image:

'none': no calibration transformation is applied.

'xz': the calibration transformations which describe the geometrical properties of the measurement system (camera and light line projector) are taken into account, but the movement of the object during the measurement is not taken into account.

'xyz': the calibration transformations which describe the geometrical properties of the measurement system (camera and light line projector) as well as the transformation which describe the movement of the object during the measurement are taken into account.

'offset_scale': a simplified set of parameters to describe the setup, that can be used with default parameters or can be controlled by six parameters. Three of the parameters describe an anisotropic scaling: *'scale_x'* describes the scaling of a pixel in column direction into the new x-axis, *'scale_y'* describes the linear movement between two profiles, and *'scale_z'* describes the scaling of to measured disparities into the new z-axis. The other three parameters describe the offset of the frame of reference of the resulting x,y,z values (*'offset_x'*, *'offset_y'*, *'offset_z'*).

'camera_parameter': the internal parameters of the camera used for the measurement. Those parameters are required when the calibration extent has been set to *'xz'* or *'xyz'*.

'camera_pose': the pose of the world coordinate system relative to the camera coordinate system. This pose is required when the calibration extent has been set to *'xz'* or *'xyz'*.

'lightplane_pose': the pose of the light-plane coordinate system relative to the world coordinate system. The light-plane coordinate system must be chosen so that its plane z=0 coincides with the light plane described by the light line projector. This pose is required when the calibration extent has been set to *'xz'* or *'xyz'*.

'movement_pose': a pose representing the movement of the object between two successive profile images with respect to the measurement system built by the camera and the laser. This pose is required when the calibration extent has been set to *'xyz'*.

'scale': with this parameter you can scale the 3D coordinates X, Y and Z that result when applying the calibration transformations to the disparity image. *'scale'* must be specified as the ratio *desired unit/original unit*. The original unit is determined by the coordinates of the calibration object. If you use the standard calibration plate the original unit is meter. This parameter can only be set if the calibration extent has been set to *'offset_scale'*, *'xz'* or *'xyz'*. By default, *'scale'* is set to 1.0.

'scale_x': This value defines the width of a pixel in 3D space. The value is only applicable if the calibration extend is set to *'offset_scale'*. By default, *'scale_x'* is set to *1.0*.

'scale_y': This value defines the linear movement between two profiles in 3D space. The value is only applicable if the calibration extend is set to *'offset_scale'*. By default, *'scale_y'* is set to *10.0*.

'scale_z': This value defines the height of disparities in 3D space. The value is only applicable if the calibration extend is set to *'offset_scale'*. By default, *'scale_z'* is set to *1.0*.

'offset_x': This value defines the x offset of reference frame for 3D results. The value is only applicable if the calibration extend is set to *'offset_scale'*. By default, *'offset_x'* is set to *0.0*.

'offset_y': This value defines the y offset of reference frame for 3D results. The value is only applicable if the calibration extend is set to *'offset_scale'*. By default, *'offset_y'* is set to *0.0*.

'offset_z': This value defines the z offset of reference frame for 3D results. The value is only applicable if the calibration extend is set to *'offset_scale'*. By default, *'offset_z'* is set to *0.0*.

Parameters

▷ **SheetOfLightModelID** (input_control) . . . . . sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Name of the generic parameter that shall be queried.
Default: 'method'
List of values: GenParamName ∈ {'min_gray', 'method', 'ambiguity_solving', 'score_type', 'num_profiles', 'calibration', 'camera_parameter', 'camera_pose', 'lightplane_pose', 'movement_pose', 'scale', 'scale_x', 'scale_y', 'scale_z', 'offset_x', 'offset_y', 'offset_z'}

▷ **GenParamValue** (output_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Value of the model parameter that shall be queried.

Result

The operator get_sheet_of_light_param returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_sheet_of_light_params, set_sheet_of_light_param

Possible Successors

measure_profile_sheet_of_light, set_sheet_of_light_param, apply_sheet_of_light_calibration

Module

3D Metrology

get_sheet_of_light_result (: ResultValue : SheetOfLightModelID, ResultName :)

Get the iconic results of a measurement performed with the sheet-of light technique.

The operator get_sheet_of_light_result provides access to the results of the calibrated and uncalibrated measurements performed with a given sheet-of-light model. The different kinds of results can be selected by setting the value of the parameter ResultName as described below:

Non-calibrated results:

'disparity': the measured disparity i.e., the subpixel row value at which the profile was detected is returned for each pixel. The disparity values can be considered as non-calibrated pseudo-range values.

'score': the score values computed according to the value of the parameter *'score_type'* is returned. If the parameter *'score_type'* has been set to *'none'*, no score value is computed during the measurement, therefore the returned image is empty. Refer to create_sheet_of_light_model and set_sheet_of_light_param for details on the possible values of the model parameter *'score_type'*.

Calibrated results:

'x': The calibrated X-coordinates of the reconstructed surface is returned as an image.

'y': The calibrated Y-coordinates of the reconstructed surface is returned as an image.

'z': The calibrated Z-coordinates of the reconstructed surface is returned as an image.

Please, note that the pixel values of the images returned when setting ResultName to *'x'*, *'y'* or *'z'* have the semantic of coordinates with respect to the world coordinate system that is implicitly defined during the calibration of the system. The unit of the returned coordinates depends on the value of the parameter *'scale'*. (see create_sheet_of_light_model and set_sheet_of_light_param for details on the possible values of the model parameter *'scale'*.)

The operator get_sheet_of_light_result returns an empty object if the desired result has not been computed.

Parameters

▷ **ResultValue** (output_object) ..... singlechannelimage ⇝ *object : real*
Desired measurement result.

▷ **SheetOfLightModelID** (input_control) ..... sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model to be used.

▷ **ResultName** (input_control) ..... string(-array) ⇝ *string*
Specify which result of the measurement shall be provided.
Default: 'disparity'
List of values: ResultName ∈ {'disparity', 'score', 'x', 'y', 'z'}

Result

The operator get_sheet_of_light_result returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Module

3D Metrology

get_sheet_of_light_result_object_model_3d (
: : SheetOfLightModelID : ObjectModel3D)

Get the result of a calibrated measurement performed with the sheet-of-light technique as a 3D object model.

The operator get_sheet_of_light_result_object_model_3d returns the result of a fully calibrated sheet-of-light measurement as a 3D object model. The handle of the sheet-of-light model with which the measurement is performed must be passed to SheetOfLightModelID. The calibration extent of the sheet-of-light model (*'calibration'*) must have been set to *'xyz'* or *'offset_scale'* before applying the measurement, otherwise the computed coordinates cannot be returned as a 3D object model and an exception is raised.

The handle of the 3D object model resulting from the measurement is returned in ObjectModel3D. For the 3D points within this 3D object model no triangular meshing is available, therefore no faces are stored in the 3D object model. If a 3D object model with triangular meshing is required for the subsequent processing, use the operator get_sheet_of_light_result in order to retrieve the *'x'*, *'y'*, and *'z'* coordinates from the sheet-of-light model and then call the operator xyz_to_object_model_3d with suitable parameters. Refer to xyz_to_object_model_3d for more information about 3D object models.

The unit of the returned coordinates depends on the value of the parameter *'scale'* that was set for the sheet-of-light model before applying the measurement. See create_sheet_of_light_model and set_sheet_of_light_param for details on the possible values of the model parameter *'scale'*. The operator get_sheet_of_light_result_object_model_3d returns a handle to an empty 3D object model if the desired result has not been measured yet.

Parameters

▷ **SheetOfLightModelID** (input_control) ..... sheet_of_light_model ⇝ *handle*
Handle for accessing the sheet-of-light model.

▷ **ObjectModel3D** (output_control) ..... object_model_3d ⇝ *handle*
Handle of the resulting 3D object model.

Result

The operator get_sheet_of_light_result_object_model_3d returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

create_sheet_of_light_model, measure_profile_sheet_of_light, calibrate_sheet_of_light

Possible Successors

clear_object_model_3d

Module

3D Metrology

measure_profile_sheet_of_light (
ProfileImage : : SheetOfLightModelID, MovementPose :)

Process the profile image provided as input and store the resulting disparity to the sheet-of-light model.

The operator measure_profile_sheet_of_light processes the ProfileImage and stores the resulting disparity values to the sheet-of-light model. Please note that ProfileImage will only be processed in the region defined by ProfileRegion as set with the operator create_sheet_of_light_model. Since ProfileImage is processed column by column, the profile must be oriented roughly horizontal.

Influence of different model parameters

If the model parameter *'score_type'* has been set to *'intensity'* or *'width'*, score values are also computed and stored into the model. Refer to set_sheet_of_light_param for details on the possible values of *'score_type'*.

If the model parameter *'calibration'* has been set to *'xz'*, *'xyz'*, or *'offset_scale'* and all parameters required to determine the calibration transformation have been set to the sheet-of-light model, the calibration transformations will be automatically applied to the disparity values after the measurement. Refer to set_sheet_of_light_param for details on setting the calibration parameters to the sheet-of-light model.

Setting MovementPose

MovementPose describes the movement of the object between the acquisition of the previous profile and the acquisition of the current profile.

If the model parameter *'calibration'* has been set to *'none'* or *'xz'* (see set_sheet_of_light_param) the movement of the object is not taken into consideration by the calibration transformation. Therefore, MovementPose is ignored, and it can be set to an empty tuple.

If the model parameter *'calibration'* has been set to *'xyz'*, the pose describing the movement of the object must be specified to the sheet-of-light model. This can be done here with MovementPose or with the parameter *'movement_pose'* in the operator set_sheet_of_light_param.

If the model parameter *'calibration'* has been set to *'offset_scale'*, a movement can be specified, but it should be considered, that the space to which this transformation is applied is most probably not metrically.

If the movement of the object between the recording of two successive profiles is constant, we recommend to set MovementPose here to an empty tuple, and to set the constant pose via the parameter *'movement_pose'* in the operator set_sheet_of_light_param. This configuration is often encountered, for example when the object under measurement is moved by a conveyor belt and measured by a fixed measurement system.

If the movement of the object between the recording of two successive profiles is not constant, for example because the measurement system is moved over the object by a robot, you must set MovementPose here for each call of measure_profile_sheet_of_light.

MovementPose must be expressed in the world coordinate system that is implicitly defined during the calibration of the measurement system.

Parameters

- ▷ **ProfileImage** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
  Input image.
- ▷ **SheetOfLightModelID** (input_control) ..... sheet_of_light_model ⇝ *handle*
  Handle of the sheet-of-light model.
- ▷ **MovementPose** (input_control) ..... number-array ⇝ *integer* / real
  Pose describing the movement of the scene under measurement between the previously processed profile image and the current profile image.

Result

The operator measure_profile_sheet_of_light returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SheetOfLightModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Successors

apply_sheet_of_light_calibration, get_sheet_of_light_result

See also

query_sheet_of_light_params, get_sheet_of_light_param, get_sheet_of_light_result, apply_sheet_of_light_calibration

Module

3D Metrology

query_sheet_of_light_params (: : SheetOfLightModelID, QueryName : GenParamName)

For a given sheet-of-light model get the names of the generic iconic or control parameters that can be used in the different sheet-of-light operators.

The operator query_sheet_of_light_params returns the names of the generic parameters that are supported by the following operators create_sheet_of_light_model, set_sheet_of_light_param, get_sheet_of_light_param and get_sheet_of_light_result. The parameter QueryName is used to select the desired **parameter group**:

'create_model_params': create_sheet_of_light_model – Parameters for adjusting the sheet-of-light model during its creation.

'set_model_params': set_sheet_of_light_param – Parameters for adjusting the parameters of an available sheet-of-light model.

'get_model_params': get_sheet_of_light_param – Parameters for querying the values of the parameters of a sheet-of-light model.

'get_result_objects': get_sheet_of_light_result – Parameters for accessing the iconic objects resulting from the measurement.

The returned parameter list does not depend on the current state of the model or its results.

Parameters

▷ **SheetOfLightModelID** (input_control) ..... sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

▷ **QueryName** (input_control) ..... attribute.name ⇝ *string*
Name of the parameter group.
Default: 'create_model_params'
List of values: QueryName ∈ {'create_model_params', 'set_model_params', 'get_model_params', 'get_result_objects'}

▷ **GenParamName** (output_control) ..... attribute.value-array ⇝ *string*
List containing the names of the supported generic parameters.

Result

The operator query_sheet_of_light_params returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

create_sheet_of_light_model, set_sheet_of_light_param, get_sheet_of_light_param, get_sheet_of_light_result

Module

3D Metrology

read_sheet_of_light_model (: : FileName : SheetOfLightModelID)

Read a sheet-of-light model from a file and create a new model.

The operator read_sheet_of_light_model reads the sheet-of-light model from the file FileName and creates a new model that is an identical copy of the saved model. The parameter SheetOfLightModelID returns the handle of the new model. The model file FileName must have been created by the operator write_sheet_of_light_model. The default HALCON file extension for sheet-of-light model is 'solm'.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the sheet-of-light model file.
Default: 'sheet_of_light_model.solm'
File extension: .solm

▷ **SheetOfLightModelID** (output_control) ..... sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

Result

The operator read_sheet_of_light_model returns the value 2 (H_MSG_TRUE) if the named file was found and correctly read. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

measure_profile_sheet_of_light

Alternatives

create_sheet_of_light_model

See also

write_sheet_of_light_model

Module

3D Metrology

reset_sheet_of_light_model (: : SheetOfLightModelID :)

Reset a sheet-of-light model.

The operator reset_sheet_of_light_model resets a sheet-of-light model that was created by create_sheet_of_light_model. All indices and result arrays used by the model are reset. The parameters of the model remain unchanged. The handle of the model is passed in SheetOfLightModelID.

Parameters

▷ **SheetOfLightModelID** (input_control) ..... sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

Result

The operator reset_sheet_of_light_model returns the value 2 (H_MSG_TRUE) if a valid handle is passed and the sheet-of-light model can be reset correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SheetOfLightModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

clear_sheet_of_light_model

Module

3D Metrology

serialize_sheet_of_light_model (
: : SheetOfLightModelID : SerializedItemHandle)

Serialize a sheet-of-light model.

serialize_sheet_of_light_model serializes a sheet-of-light model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_sheet_of_light_model is converted to a serialized item. The sheet-of-light model is defined by the handle SheetOfLightModelID. The serialized model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_sheet_of_light_model.

Parameters

▷ **SheetOfLightModelID** (input_control) . . . . . sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

The operator serialize_sheet_of_light_model returns the value 2 (H_MSG_TRUE) if the passed handle of the sheet-of-light model is valid and if the model can be serialized into the serialized item. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_sheet_of_light_model, set_sheet_of_light_param

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_sheet_of_light_model

See also

deserialize_sheet_of_light_model

Module

3D Metrology

set_profile_sheet_of_light (
ProfileDisparityImage : : SheetOfLightModelID, MovementPoses :)

Set sheet of light profiles by measured disparities.

set_profile_sheet_of_light adds sheet-of-light profiles to the sheet-of-light model SheetOfLightModelID. The profiles are specified as rows in a disparity image in ProfileDisparityImage. Each of the profiles can have an individual Pose set in MovementPoses which is interpreted as relative movement to the previous row. If no pose is set, the default transformation is used, which can be set by set_sheet_of_light_param. If only one pose is set, this pose will become the default transformation.

Parameters

- ▷ **ProfileDisparityImage** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
  Disparity image that contains several profiles.
- ▷ **SheetOfLightModelID** (input_control) ..... sheet_of_light_model ⇝ *handle*
  Handle of the sheet-of-light model.
- ▷ **MovementPoses** (input_control) ..... number-array ⇝ *integer* / real
  Poses describing the movement of the scene under measurement between the previously processed profile image and the current profile image.

Result

The operator set_profile_sheet_of_light returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SheetOfLightModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Successors

get_sheet_of_light_result, get_sheet_of_light_result_object_model_3d

See also

query_sheet_of_light_params, get_sheet_of_light_param, get_sheet_of_light_result, apply_sheet_of_light_calibration

Module

3D Metrology

set_sheet_of_light_param (: : SheetOfLightModelID, GenParamName, GenParamValue :)

Set selected parameters of the sheet-of-light model.

The operator set_sheet_of_light_param is used to set or change a single parameter of a sheet-of-light model in order to adapt the model to a particular measurement task. All parameters, except the internal camera parameters *'camera_parameters'* and the following poses *'camera_pose'*, *'lightplane_pose'*, and *'movement_pose'* can also be set while creating a sheet-of-light model with create_sheet_of_light_model. The current configuration of the sheet-of-light model can be queried with the operator get_sheet_of_light_param. A list with the names of all parameters that can be set for the sheet-of-light model is returned by query_sheet_of_light_params.

The following overview lists the different generic parameters with the respective value ranges and default values:

Measurement of the profiles:

'method': defines the method used to determine the position of the profile. The values *'default'* and *'center_of_gravity'* both refer to the same method, whereby the position of the profile is determined column

by column with subpixel accuracy by computing the center of gravity of the gray values g_i of all pixels fulfilling the condition:

$$g_i \geq 'min\_gray'$$

'min_gray': lowest gray values taken into account for the measurement of the position of the profile (see *'center_of_gravity'*).

Suggested values: 20, *50*, *100*, *128*, *200*, *220*, *250*

Default: 100

'num_profiles': number of profiles for which memory has been allocated within the sheet-of-light model. By default, *'num_profiles'* is set to 512. If this number of profiles is exceeded, memory will be reallocated automatically during the measurement.

Suggested values: 1, *2*, *50*, *100*, *512*, *1024*, *3000*

Default: 512

'ambiguity_solving': method applied to determine which candidate shall be chosen if the determination of the position of the profile is ambiguous.

'first': the first encountered candidate is returned. This method is the fastest.

'last': the last encountered candidate is returned.

'brightest': for each candidate, the brightness of the profile is computed and the candidate having the highest brightness is returned. The brightness is computed according to:

$$brightness = \frac{1}{n} \sum_{i=0}^{n} g_i \ ,$$

where g_i is the gray value of the pixel and n the number of pixels taken into consideration to determine the position of the profile.

Default: 'first'

'score_type': method used to calculate a score for the measurement of the position of the profile.

'none': no score is computed.

'width': for each pixel of the disparity, the score value is set to the number of pixels used to determine the disparity value.

'intensity': for each pixel of the disparity, a score value is evaluated by computing the local intensity of the profile according to:

$$score = \frac{1}{n} \sum_{i=0}^{n} g_i$$

where g_i is the gray value of the pixel and n the number of pixels taken into consideration to determine the position of the profile.

Default: 'none'

Calibration of the measurement:

'calibration': extent of the calibration transformation which shall be applied to the disparity image:

'none': no calibration transformation is applied.

'xz': the calibration transformations which describe the geometrical properties of the measurement system (camera and light line projector) are taken into account, but the movement of the object during the measurement is not taken into account.

'xyz': the calibration transformations which describe the geometrical properties of the measurement system (camera and light line projector) as well as the transformation which describe the movement of the object during the measurement are taken into account.

'offset_scale': a simplified set of parameters to describe the setup, that can be used with default parameters or can be controlled by six parameters. Three of the parameters describe an anisotropic scaling: *'scale_x'* describes the scaling of a pixel in column direction into the new x-axis, *'scale_y'* describes the linear movement between two profiles, and *'scale_z'* describes the scaling of to measured disparities into the new z-axis. The other three parameters describe the offset of the frame of reference of the resulting x,y,z values (*'offset_x'*, *'offset_y'*, *'offset_z'*).

Default: 'none'

'camera_parameter': the internal parameters of the camera used for the measurement. Those parameters are required if the calibration extent has been set to *'xz'* or *'xyz'*. If `calibrate_sheet_of_light` shall be used for calibration, this parameter is used to set the initial camera parameters.

'calibration_object': the calibration object used for calibration with `calibrate_sheet_of_light`. If `calibrate_sheet_of_light` shall be used for calibration, this parameter must be set to the filename of a calibration object created with `create_sheet_of_light_calib_object`.

'camera_pose': the pose that transforms the camera coordinate system into the world coordinate system, i.e., the pose that could be used to transform point coordinates from the world coordinate system into the camera coordinate system. This pose is required if the calibration extent has been set to *'xz'* or *'xyz'*.

Note that the world coordinate system is implicitly defined by setting the *'camera_pose'*.

'lightplane_pose': the pose that transforms the light plane coordinate system into the world coordinate system, i.e., the pose that could be used to transform point coordinates from the world coordinate system into the light plane coordinate system. The light plane coordinate system must be chosen such that the plane z=0 coincides with the light plane. This pose is required if the calibration extent has been set to *'xz'* or *'xyz'*.

'movement_pose': a pose representing the movement of the object between two successive profile images with respect to the measurement system built by the camera and the laser. This pose must be expressed in the world coordinate system. It is required if the calibration extent has been set to *'xyz'*.

'scale': with this value you can scale the 3D coordinates X, Y and Z that result when applying the calibration transformations to the disparity image. The model parameter *'scale'* must be specified as the ratio *desired unit/original unit*. The original unit is determined by the coordinates of the calibration object. If the original unit is meters (which is the case if you use the standard calibration plate), you can set *'scale'* to the desired unit directly by selecting *'m'*, *'cm'*, *'mm'*, *'microns'*, or *'um'*. This parameter can only be set if the calibration extent has been set to *'offset_scale'*, *'xz'* or *'xyz'*.

Suggested values: 'm', 'cm', 'mm', 'microns', 'um', 1.0, 0.01, 0.001, 1.0e-6

Default value: 1.0

'scale_x': This value defines the width of a pixel in the 3D space. This parameter can only be set if the calibration extent has been set to *'offset_scale'*.

Suggested values: 10.0, 1.0, 0.01, 0.001, 1.0e-6

Default value: 1.0

'scale_y': This value defines the linear movement between two profiles in the 3D space. This parameter can only be set if the calibration extent has been set to *'offset_scale'*.

Suggested values: 100.0, 10.0, 1.0, 0.1, 1.0e-6

Default value: 10.0

'scale_z': This value defines the height of a pixel in the 3D space. This parameter can only be set if the calibration extent has been set to *'offset_scale'*.

Suggested values: 10.0, 1.0, 0.01, 0.001, 1.0e-6

Default value: 1.0

'offset_x': This value defines the x offset of reference frame for 3D results. This parameter can only be set if the calibration extent has been set to *'offset_scale'*.

Suggested values: 10.0, 0.0, 0.01, 0.001, 1.0e-6

Default value: 0.0

'offset_y': This value defines the y offset of reference frame for xyz results. This parameter can only be set if the calibration extent has been set to *'offset_scale'*.

Suggested values: 10.0, 0.0, 0.01, 0.001, 1.0e-6

Default value: 0.0

'offset_z': This value defines the z offset of reference frame for 3D results. This parameter can only be set if the calibration extent has been set to *'offset_scale'*.

Suggested values: 10.0, 0.0, 0.01, 0.001, 1.0e-6

Default value: 0.0

Parameters

▷ **SheetOfLightModelID** (input_control) . . . . . sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Name of the model parameter that shall be adjusted for the sheet-of-light model.
Default: 'method'
List of values: GenParamName ∈ {'method', 'ambiguity_solving', 'score_type', 'num_profiles', 'min_gray', 'scale', 'calibration', 'calibration_object', 'camera_parameter', 'camera_pose', 'lightplane_pose', 'movement_pose', 'scale_x', 'scale_y', 'scale_z', 'offset_x', 'offset_y', 'offset_z'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Value of the model parameter that shall be adjusted for the sheet-of-light model.
Default: 'center_of_gravity'
Suggested values: GenParamValue ∈ {'default', 'center_of_gravity', 'last', 'first', 'brightest', 'none', 'intensity', 'width', 'xz', 'xyz', 'offset_scale', 'm', 'cm', 'mm', 'um', 'microns', 1.0, 1e-2, 1e-3, 1e-6}

Result

The operator set_sheet_of_light_param returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SheetOfLightModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Successors

get_sheet_of_light_param, measure_profile_sheet_of_light, apply_sheet_of_light_calibration

Alternatives

create_sheet_of_light_model

See also

query_sheet_of_light_params, get_sheet_of_light_param, get_sheet_of_light_result

Module

3D Metrology

write_sheet_of_light_model (: : SheetOfLightModelID, FileName :)

Write a sheet-of-light model to a file.

The operator write_sheet_of_light_model writes the sheet-of-light model SheetOfLightModelID to the file FileName. The model can be read again with read_sheet_of_light_model. The stored data contains all generic model parameters (see set_sheet_of_light_param) and the results of calibrate_sheet_of_light.

The default HALCON file extension for sheet-of-light model is 'solm'.

Parameters

▷ **SheetOfLightModelID** (input_control) . . . . . sheet_of_light_model ⇝ *handle*
Handle of the sheet-of-light model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the sheet-of-light model file.
Default: 'sheet_of_light_model.solm'
File extension: .solm

Result

The operator write_sheet_of_light_model returns the value 2 (H_MSG_TRUE) if the passed handle is valid and if the model can be written into the named file. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_sheet_of_light_model, set_sheet_of_light_param

See also

read_sheet_of_light_model

Module

3D Metrology

5.6 Structured Light

This chapter describes the usage of structured light for 3D reconstruction.

Concept of Structured Light

The basic concept behind structured light is to use a structured illumination, i.e. an illumination showing well known patterns. The way those patterns appear in the scene after hitting surfaces helps to further analyze (e.g., perform an inspection Inspection / Structured Light) or reconstruct the surfaces.

For non-specular (lambertian or diffuse) surfaces, a 3D surface can be reconstructed using a projector projecting light like an 'inverse camera'. For every projected pattern image, a camera image of the projection on the surface is acquired. Using the decoded correspondence between projector coordinates lighting the camera coordinates, as well as calibration information, the 3D surface is reconstructed.

In the following, the steps that are required to use structured light are described briefly.

Create a structured light model: In the first step, a structured light model is created with

- create_structured_light_model (ModelType='3d_reconstruction')

or read with

- read_structured_light_model.

Set the model parameters: The different structured light model parameters can then be set with

- set_structured_light_model_param

or queried with

- get_structured_light_model_param.

The pattern parameters *'pattern_width'*, *'pattern_height'*, *'pattern_orientation'*, and *'pattern_type'* specify along with the stripe parameters *'min_stripe_width'* and *'single_stripe_width'* the specifications of the pattern images to be used to illuminate the surface. Finally, the *'persistence'* parameter can be enabled to debug intermediate results.

Generate the pattern images: The pattern images are to be generated with `gen_structured_light_pattern` after setting all relevant parameters. Please ensure that the output images are as needed in the particular setup.

Use the patterns to illuminate the surface and acquire the camera images: At this stage, the pattern images are projected. The respective image of the illuminated surface is acquired by the camera for each pattern image.

When calibrating the system, images of the illuminated calibration object need to be acquired. The calibration process is shown in detail in the example program `structured_light_calibration.hdev`. The obtained calibration information can then be specified with the parameter *'camera_setup_model'* of `set_structured_light_model_param`.

Decode the acquired images: The acquired `CameraImages` can be decoded with `decode_structured_light_pattern`. Upon calling this operator, the correspondence images are created and stored in the model `StructuredLightModel`.

Get the results: The decoded *'correspondence_image'*, as well as other results can be queried with `get_structured_light_object`. For more details of the different objects that can be queried, please refer to the operator's documentation.

Perform the reconstruction: The reconstructed surface can be obtained with `reconstruct_surface_structured_light`.

Further operators

The structured light model offers various other operators that help access and update the various parameters of the model.

The operator `write_structured_light_model` enables writing the structured light model to a file. Please note that previously generated pattern images are not written in this file. A structured light model file can be read using `read_structured_light_model`.

Furthermore, it is possible to serialize and deserialize the structured light model using `serialize_structured_light_model` and `deserialize_structured_light_model`.

Further Information

See also the "`Solution Guide Basics`" for further details. For a list of operators, please refer to Inspection / Structured Light.

Chapter 6

Calibration

This chapter provides information regarding camera calibration.

General Objectives

To achieve maximum accuracy of measurement for your camera setup, you have to calibrate it accordingly. Therefore, a camera model is determined, which describes the projection of a 3D world point into a (sub-)pixel in the image.

HALCON provides a wide range of operators to approach diverse tasks related to calibration, such as

- describing and finding a calibration object (Calibration / Calibration Object),
- the projection of points from the 3D scene onto the image plane and the other way around (Calibration / Projection, Calibration / Inverse Projection),
- compensating perspective and radial distortions (Calibration / Rectification),
- handling the camera parameters (Calibration / Camera Parameters),
- performing a self-calibration (Calibration / Self-Calibration), and
- calibrating different setups consisting of
  - one camera (Calibration / Monocular),
  - multiple cameras (Calibration / Binocular, Calibration / Multi-View), or
  - a camera in combination with a robot (Calibration / Hand-Eye).

This chapter gives guidance regarding the basic concept of retrieving the internal and external parameters of your camera. The following paragraphs state how to successfully calibrate a camera. In particular, they describe

- the needed calibration object,
- the individual steps to calibrate the cameras, including
  - how to prepare the calibration input data,
  - how to perform the actual calibration with `calibrate_cameras`, and
  - how to check the success of the calibration,
- the camera parameters,
- additional information about the calibration process, including
  - how to obtain an appropriate calibration plate,
  - how to take a set of suitable images, and
  - which distortion model to use,

- the available 3D camera models and how 3D points are transformed into the image coordinate system, and
- limitations related to specific camera types.

Calibration Object

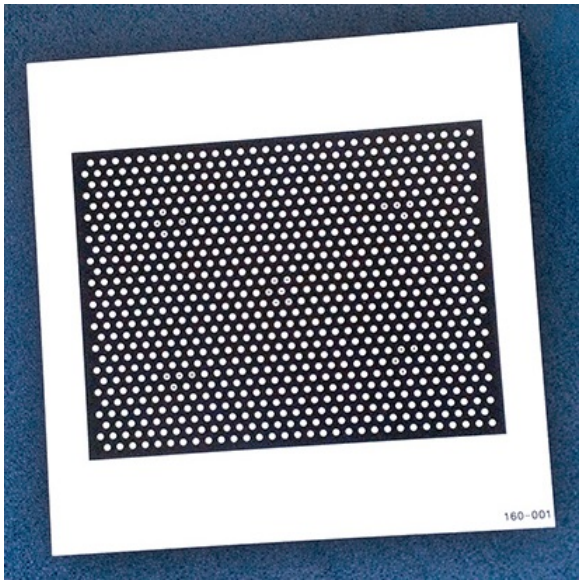
For a successful calibration of your camera setup, at least one calibration object with accurately known metric properties is needed, e.g., a HALCON calibration plate. For the calibration, take a series of images of the calibration object in different positions and orientations. The success of the calibration highly depends on the quality of the calibration object and the images. So you might want to exercise special diligence during the acquisition of the calibration images. See the section "How to take a set of suitable images?" for further information.

A calibration plate is covered by multiple calibration marks, which are extracted in the calibration images in order to retrieve their coordinates. The orientation of the plate has to be known distinctively, hence, a finder pattern is also part of the imprint.

Your distributor can provide you with two different types of standard HALCON calibration plates:

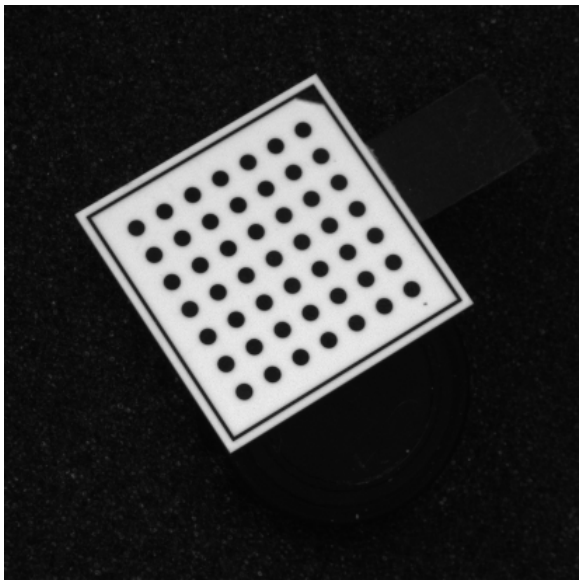
Calibration plate with hexagonally arranged marks: As finder pattern, there are special groups of mark hexagons where some of the marks contain dot-shaped holes (see `create_caltab`). One finder pattern has to be visible to locate the calibration plate. To make sure the plate is not inverted, at least a second one needs to be seen, but the plate does not have to be fully visible in the image. The origin of the coordinate system is located at the center of the central mark of the first finder pattern. The z-axis of the coordinate system is pointing into the calibration plate, its x-axis is pointing to the right, and its y-axis is pointing downwards with the direction of view along the z-axis.

When using `camera_calibration` instead of `calibrate_cameras`, this calibration plate is not applicable.



HALCON calibration plate with hexagonally arranged marks.

Calibration plate with rectangularly arranged marks: The finder pattern consists of the surrounding frame and the triangular corner marker (see `gen_caltab`). Thus, the plate has to be fully visible in the image. The origin is located in the middle of the surface of the calibration plate. The z-axis of the coordinate system is pointing into the calibration plate, its x-axis is pointing to the right, and its y-axis is pointing downwards with the direction of view along the z-axis.



HALCON calibration plate with rectangularly arranged marks.

When acquiring your calibration images, note that there are different recommendations on how to take them, depending on your used calibration plate (see section “How to take a set of suitable images?”).

Preparing the Calibration Input Data

Before calling a calibration operator (e.g., `calibrate_cameras`), you must create and adapt the calibration data model with the following steps:

1. **Create a calibration data model** with the operator `create_calib_data`, specifying the number of cameras in the setup and the number of used calibration objects.
2. **Specify the camera type and the initial internal camera parameters** with the operator `set_calib_data_cam_param`.
3. **Specify the description of all calibration objects** with the operator `set_calib_data_calib_object`.
4. **Collect observation data** with the operators `find_calib_object` or `set_calib_data_observ_points`, i.e., obtain the image coordinates of the extracted calibration marks of the calibration object and a roughly estimated pose of the calibration object relative to the observing camera.
5. **Configure the calibration process**, e.g., exclude certain camera parameters from the optimization. You can specify these parameters with the operator `set_calib_data`. For example, if the image sensor cell size of camera 0 is known precisely and only the rest of the parameters needs to be calibrated, you call

```
set_calib_data(CalibDataID, 'camera', 0, 'excluded_settings',
['sx','sy']).
```

Performing the Actual Camera Calibration and Obtaining its Results

Using all the information stored within the calibration data model, the actual calibration can be performed calling `calibrate_cameras`. Thereby, the input model is modified by optimizing the initial internal camera parameters, computing and adding further data like the external camera parameters or standard deviations. Furthermore, the standard deviations and covariances of the calibrated internal parameters and the root mean square error of the back projection are calculated in order to check the success of the calibration.

The results can then be queried with the operator `get_calib_data`.

Checking the Success of the Calibration

After a successful calibration, the root mean square error (RMSE) of the back projection of the optimization is returned in `Error` (in pixels). The error gives a general indication whether the optimization was successful as it corresponds to the average distance (in pixels) between the back projected calibration points and their extracted image coordinates.

If only a single camera is calibrated, an `Error` in the order of 0.1 pixel (the typical detection error by extraction of the coordinates of the projected calibration markers) is an indication that the optimization fits the observation data well. If `Error` strongly differs from 0.1 pixels, the calibration did not perform well. Reasons for this might be, e.g., a poor image quality, an insufficient number of calibration images, or an inaccurate calibration plate.

For information about how to check the success of the calibration using a multi-view camera setup, see the respective section in the chapter Calibration / Multi-View.

Camera Parameters

Regarding camera parameters, you can distinguish between internal and external camera parameters.

Internal camera parameters: These parameters describe the characteristics of the used camera, especially the dimension of the sensor itself and the projection properties of the used combination of lens, camera, and frame grabber. Below is an overview of all available camera types and their respective parameters `CameraParam`. In the list, "projective cameras" refers to the property that the lens performs a perspective projection on the object-side of the lens, while "telecentric cameras" refers to the property that the lens performs a telecentric projection on the object-side of the lens.

Area scan cameras have 9 to 16 internal parameters depending on the camera type.

For reasons explained below, parameters that are marked with an * asterisk are fixed and not estimated by the algorithm.

Area scan cameras with regular lenses

Projective area scan cameras with regular lenses

- *'area_scan_division'*:
  ['area_scan_division', Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*
- *'area_scan_polynomial'*:
  ['area_scan_polynomial', Focus, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*

Telecentric area scan cameras with regular lenses

- *'area_scan_telecentric_division'*:
  ['area_scan_telecentric_division', Magnification, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*
- *'area_scan_telecentric_polynomial'*:
  ['area_scan_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*

Area scan cameras with tilt lenses

Projective area scan cameras with tilt lenses

- *'area_scan_tilt_division'*:
  ['area_scan_tilt_division', Focus, Kappa, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*
- *'area_scan_tilt_polynomial'*:
  ['area_scan_tilt_polynomial', Focus, K1, K2, K3, P1, P2, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*
- *'area_scan_tilt_image_side_telecentric_division'*:
  ['area_scan_tilt_image_side_telecentric_division', Focus, Kappa, Tilt, Rot, Sx, Sy*, Cx, Cy, ImageWidth, ImageHeight]*
- *'area_scan_tilt_image_side_telecentric_polynomial'*:
  ['area_scan_tilt_image_side_telecentric_polynomial', Focus, K1, K2, K3, P1, P2, Tilt, Rot, Sx, Sy*, Cx, Cy, ImageWidth, ImageHeight]*

Telecentric area scan cameras with tilt lenses

- *'area_scan_tilt_bilateral_telecentric_division'*:
  ['area_scan_tilt_bilateral_telecentric_division', Magnification, Kappa, Tilt, Rot, Sx, Sy*, Cx, Cy, ImageWidth, ImageHeight]*

- *'area_scan_tilt_bilateral_telecentric_polynomial'*:
  ['area_scan_tilt_bilateral_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, Tilt, Rot, Sx, Sy*, Cx, Cy, ImageWidth, ImageHeight]*
- *'area_scan_tilt_object_side_telecentric_division'*:
  ['area_scan_tilt_object_side_telecentric_division', Magnification, Kappa, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*
- *'area_scan_tilt_object_side_telecentric_polynomial'*:
  ['area_scan_tilt_object_side_telecentric_polynomial', Magnification, K1, K2, K3, P1, P2, ImagePlaneDist, Tilt, Rot, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*

Area scan cameras with hypercentric lenses

Projective area scan cameras with hypercentric lenses

- *'area_scan_hypercentric_division'*:
  ['area_scan_hypercentric_division', Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*
- *'area_scan_hypercentric_polynomial'*:
  ['area_scan_hypercentric_polynomial', Focus, K1, K2, K3, P1, P2, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]*

Description of the internal camera parameters of area scan cameras:

CameraType: Type of the camera, as listed above.

Focus: Focal length of the lens (only for lenses that perform a perspective projection on the object side of the lens).
The initial value is the nominal focal length of the used lens, e.g., 0.008m.

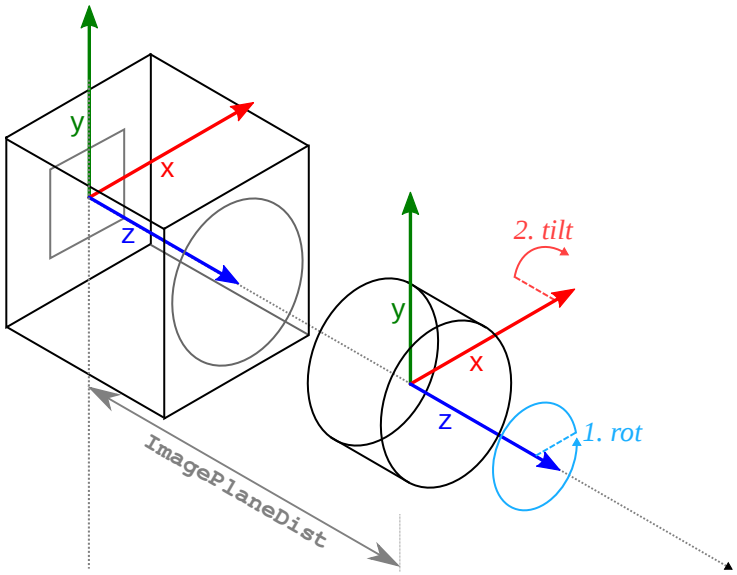
Magnification: Magnification of the lens (only for lenses that perform a telecentric projection on the object side of the lens).
The initial value is the nominal magnification of the used telecentric lens (the image size divided by the object size), e.g., 0.2.

Kappa (κ)**:** Distortion coefficient to model the radial lens distortions (only for the division model).
Use 0.0 m^{-2} as initial value.

K1, K2, K3, P1, P2: Distortion coefficients to model the radial (K_1, K_2, K_3) and decentering (P_1, P_2) lens distortions (only for the polynomial model).
Use 0.0 as initial value for all five coefficients.

ImagePlaneDist: Distance of the exit pupil of the lens to the image plane. The exit pupil is the (virtual) image of the aperture stop (typically the diaphragm), as viewed from the image side of the lens. Typical values are in the order of a few centimeters to very large values if the lens is close to being image-side telecentric.

Tilt, Rot: The tilt angle $tilt$ $(0° \leq tilt < 90°)$ describes the angle by which the optical axis is tilted with respect to the normal of the sensor plane (corresponds to a rotation around the x-axis). The rotation angle rot $(0° \leq rot < 360°)$ describes the rotation around the optical axis (z-axis). For a rotation $rot = 0°$ the optical axis gets tilted vertically down with respect to the camera housing, $rot = 90°$ corresponds to the optical axis being tilted horizontally to the left (direction of view along the z-axis), $rot = 180°$ corresponds to the optical axis being tilted vertically up, and $rot = 270°$ corresponds to the optical axis being tilted horizontally to the right by $tilt$.
These parameters are only used if a tilt lens is part of the camera setup.



The tilt of the lens is described by the parameters rot , $tilt$ and `ImagePlaneDist`. rot describes the orientation of the tilt axis in relation to the x-axis of the sensor and has to be applied first. $tilt$ describes the actual tilt of the lens. `ImagePlaneDist` is the distance of the exit pupil of the lens to the image plane.

These angles are typically roughly known based on the considerations that led to the use of the tilt lens or can be read off from the mechanism by which the lens is tilted.

Sx, Sy: Scale factors. They correspond to the horizontal and vertical distance between two neighboring cells on the sensor. Since in most cases the image signal is sampled line-synchronously, S_y is determined by the dimension of the sensor and does not need to be estimated by the calibration process.
The initial values depend on the dimensions of the used chip of the camera. See the technical specification of your camera for the actual values. Attention: These values increase if the image is subsampled!
As **projective cameras** are described through the pinhole camera model, it is impossible to determine `Focus`, S_x, and S_y simultaneously. Therefore, the algorithm will keep S_y fixed.
For **telecentric lenses**, it is impossible to determine `Magnification`, S_x, and S_y simultaneously. Therefore, the algorithm will keep S_y fixed.
For **image-side telecentric tilt lenses** (see chapter "Basics", section "Camera Model and Parameters" in the "`Solution Guide III-C 3D Vision`" for an overview of different types of tilt lenses), it is impossible to determine `Focus`, S_x, S_y, and the tilt parameters $tilt$ and rot simultaneously. Therefore, additionally to S_y, the algorithm will keep S_x fixed.
For **bilateral telecentric tilt lenses**, it is impossible to determine `Magnification`, S_x, S_y, and the tilt parameters $tilt$ and rot simultaneously. Therefore, additionally to S_y, the algorithm will keep S_x fixed.

Cx, Cy: Column (C_x) and row (C_y) coordinate of the principal point of the image (center of the radial distortion).
Use the half image width and height as initial values. Attention: These values decrease if the image is subsampled!

ImageWidth, Image Height: Width and height of the sampled image. Attention: These values decrease if the image is subsampled!

Line scan cameras have 12 or 16 internal parameters depending on the camera type.

For reasons explained below, parameters that are marked with an * asterisk are fixed and not estimated by the algorithm.

Line scan cameras with regular lenses

Projective line scan cameras with regular lenses

- *'line_scan_division'*:
  ['line_scan_division', Focus, Kappa, Sx, Sy*, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]*
- *'line_scan_polynomial'*:
  ['line_scan_polynomial', Focus, K1, K2, K3, P1, P2, Sx, Sy*, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz]*

Telecentric line scan cameras with regular lenses

- *'line_scan_telecentric_division'*:
  ['line_scan_telecentric_division', Magnification, Kappa, Sx, Sy*, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz*]*
- *'line_scan_telecentric_polynomial'*:
  ['line_scan_telecentric_polynomail', Magnification, K1, K2, K3, P1, P2, Sx, Sy*, Cx, Cy, ImageWidth, ImageHeight, Vx, Vy, Vz*]*

Description of the internal camera parameters of line scan cameras:

CameraType: Type of the camera, as listed above.

Focus: Focal length of the lens (only for lenses that perform a perspective projection on the object side of the lens).
The initial value is the nominal focal length of the used lens, e.g., 0.008m.

Magnification: Magnification of the lens (only for lenses that perform a telecentric projection on the object side of the lens).
The initial value is the nominal magnification of the used telecentric lens (the image size divided by the object size), e.g., 0.2.

Kappa (κ)**:** Distortion coefficient of the division model to model the radial lens distortions.
Use 0.0 m^{-2} as initial value.

K1, K2, K3, P1, P2: Distortion coefficients to model the radial (K_1, K_2, K_3) and decentering (P_1, P_2) lens distortions (only for the polynomial model).
Use 0.0 as initial value for all five coefficients.

Sx: Scale factor. Corresponds to the horizontal distance between two neighboring cells on the sensor. Note that `Focus` or `Magnification`, respectively, and S_x cannot be determined simultaneously. Therefore, S_x is kept fixed in the calibration. The initial value for S_x can be taken from the technical specifications of the camera. Attention: This value increases if the image is subsampled!

Sy: Scale factor. During the calibration, it appears only in the form $p_v = -S_y \cdot C_y$. Consequently, S_y and C_y cannot be determined simultaneously. Therefore, in the calibration, S_y is kept fixed. p_v describes the distance of the image center point from the sensor line in meters. The initial value for S_y can be taken from the technical specifications of the camera. Attention: This value increases if the image is subsampled!

Cx: Column coordinate of the image center point (center of the radial distortions). Use half of the image width as the initial value for C_x. Attention: This value decreases if the image is subsampled!

Cy: Distance of the image center point (center of the radial distortions) from the sensor line in scanlines. The initial value can normally be set to 0.

ImageWidth, ImageHeight: Width and height of the sampled image. Attention: These values decrease if the image is subsampled!

Vx, Vy, Vz: X-, Y-, and Z-component of the motion vector.
The initial values for the x-, y-, and z-component of the motion vector depend on the image acquisition setup. Assuming a camera that looks perpendicularly onto a conveyor belt and that is rotated around its optical axis such that the sensor line is perpendicular to the conveyor belt, i.e., the y-axis of the camera coordinate system is parallel to the conveyor belt, use the initial values $V_x = V_z = 0$. The initial value for V_y can then be determined, e.g., from a line scan image of an object with known size (e.g., calibration plate, ruler):

$$V_y = \frac{l[m]}{l[row]}$$

With

$$\begin{aligned} l[m] &= \text{Length of the object in object coordinates [meter]} \\ l[row] &= \text{Length of the object in image coordinates [rows]} \end{aligned}$$

If, compared to the above setup, the camera is rotated 30 degrees around its optical axis, i.e., around the z-axis of the camera coordinate system, the above determined initial values must be changed as follows:

$$\begin{aligned} V_x^z &= \sin(30^\circ)V_y \\ V_y^z &= \cos(30^\circ)V_y \\ V_z^z &= V_z = 0 \end{aligned}$$

If, compared to the first setup, the camera is rotated -20 degrees around the x-axis of the camera coordinate system, the following initial values result:

$$\begin{aligned} V_x^x &= V_x = 0 \\ V_y^x &= \cos(-20^\circ)V_y \\ V_z^x &= \sin(-20^\circ)V_y \end{aligned}$$

The quality of the initial values for V_x, V_y, and V_z are crucial for the success of the whole calibration. If they are not precise enough, the calibration may fail.
Note that for telecentric line scan cameras, the value of V_z has no influence on the image position of 3D points and therefore cannot be determined. Consequently, V_z is not optimized and left at its initial value for telecentric line scan cameras. Therefore, the initial value of V_z should be set to *0*. For setups with multiple telecentric line scan cameras that share a common motion vector (for a detailed explanation, see Calibration / Multi-View), however, V_z can be determined based on the camera poses. Therefore, in this case V_z is optimized.

Restrictions for internal camera parameters Note that the term *focal length* is not quite correct and would be appropriate only for an infinite object distance. To simplify matters, always the term *focal length* is used even if the *image distance* is meant.

For all operators that use camera parameters as input the respective parameter values are checked as to whether they fulfill the following restrictions:

$$\begin{aligned} S_x &> 0 \\ S_y &\geq 0 \\ Focus &> 0 \\ Magnification &> 0 \\ ImageWidth &> 0 \\ ImageHeight &> 0 \\ ImagePlaneDist &> 0 \\ 0 \leq tilt &< 90 \\ 0 \leq rot &< 360 \\ V_x^2 + V_y^2 + V_z^2 &\neq 0 \end{aligned}$$

For some operators the restrictions differ slightly. In particular, for operators that do not support line scan cameras the following restriction applies:

$$S_y > 0$$

External camera parameters: The following 6 parameters describe the 3D pose, i.e., the position and orientation of the world coordinate system relative to the camera coordinate system. The x- and y-axis of the camera coordinate system are parallel to the column and row axes of the image, while the z-axis is perpendicular to the image plane. For line scan cameras, the pose of the world coordinate system refers to the camera coordinate system of the first image line.

TransX: Translation along the x-axis of the camera coordinate system.

TransY: Translation along the y-axis of the camera coordinate system.

TransZ: Translation along the z-axis of the camera coordinate system.

RotX: Rotation around the x-axis of the camera coordinate system.

RotY: Rotation around the y-axis of the camera coordinate system.

RotZ: Rotation around the z-axis of the camera coordinate system.

The pose tuple contains one more element, which is the representation type of the pose. It codes the combination of the parameters `OrderOfTransform`, `OrderOfRotation`, and `ViewOfTransform`. See `create_pose` for more information about 3D poses.

When using a standard HALCON calibration plate, the world coordinate system is defined by the coordinate system of the calibration plate. See the section "Calibration Object" above for further information.

If a HALCON calibration plate is used, you can use the operator `find_calib_object` to determine initial values for all parameters. Using HALCON calibration plates with rectangularly arranged marks, a combination of the two operators `find_caltab` and `find_marks_and_pose` will have the same effect.

Parameter units: HALCON calibration plates use meters as unit. The camera parameters use corresponding units. Of course, calibration can be done using different units, but in this case the related parameters have to be adapted. Here, we list the HALCON default units for the different camera parameters:

	Parameter	**Unit**
External	RotX, RotY, RotZ	deg, deg, deg
	TransX, TransY, TransZ	m, m, m
Internal	Cx, Cy	px, px
	Focus	m
	ImagePlaneDist	m
	ImageWidth, ImageHeight	px, px
	K1, K2, K3	m^{-2}, m^{-4}, m^{-6}
	Kappa (κ)	m^{-2}
	P1, P2	m^{-1}, m^{-1}
	Magnification	- (scalar)
	Sx, Sy	m/px, m/px
	Tilt, Rot	deg, deg
	Vx, Vy, Vz	$m/scanline$, $m/scanline$, $m/scanline$

Additional Information about the Calibration Process

The use of `calibrate_cameras` leads to some questions, which are addressed in the following sections:

How to obtain an appropriate calibration plate? You can obtain high-precision calibration plates in various sizes and materials from your local distributor. These calibration plates come with associated description files and can be easily extracted with `find_calib_object`.

It is also possible to use any arbitrary object for calibration. The only requirement is that the object has characteristic points that can be robustly detected in the image and that the 3D world position of these points is known with high accuracy. See the "`Solution Guide III-C 3D Vision`" for details.

Self-printed calibration objects are usually not accurate enough for high-precision applications.

How to take a set of suitable images? With the combination of lens (fixed focus setting!), camera, and frame grabber to be calibrated, a set of images of the calibration plate must be taken (see `open_framegrabber` and `grab_image`).

Your local distributor can provide you with two different types of standard HALCON calibration plates: Calibration plates with hexagonally arranged marks (see `create_caltab`) and calibration plates with rectangularly arranged marks (see `gen_caltab`). Since these two calibration plates substantially differ from each other, in some cases additional particularities apply (see below).

The parameters and hints listed below should be considered when taking the calibration images. For a successful calibration, the setup and the used set of images should have certain qualities. These qualities may vary for the specific task and demand. In order to give guidance, values and hints suitable for a basic monocular camera setup are mentioned.

Regarding the camera setup:

- **Aperture**
  The aperture of the camera must not be changed during the acquisition of the images. If the aperture is changed after the calibration, the camera must be calibrated anew.
- **Camera pose**
  The position of the camera must not be changed during the image acquisition.
- **Focus**
  The calibration images should be sharply focused, i.e., transitions between objects should be clearly delimited. The focus, respectively the focal length, must not be changed during the image acquisition.

Regarding the placement of the calibration plates:

- **Field of view coverage and orientation**
  Within the set of calibration images, every part of the field of view should be covered by the plate at least once. The calibration plate may also fill the entire image. The orientation of the plate should vary within the set of images.
- **Tilt angles**
  The set of calibration images should also contain images with tilted calibration plates. Thereby the plate should be tilted in different directions at an angle of about 30-45°. Note, that if the recommended angle cannot be realized due to, e.g., limited depth of field, you should at least tilt your plate as steeply as possible with your setup.
- **Number of images/ calibration plate poses**
  - Plate with hexagonally arranged marks: At least 6 images (Not all, but at least 4 of them with tilted calibration plates).
  - Plate with rectangularly arranged marks: At least 15 images.

  Nevertheless, you must make sure that the poses of the calibration plates also fulfill the other requirements.
- **Inverted acquisition of the calibration plate**
  The calibration marks must not be acquired inverted. This can, e.g., happen if a calibration plate made of glass is acquired from its backside or if a line scan camera is not moving downwards with respect to the image coordinate system (i.e., V_y is negative).

Regarding image properties and content:

- **Pattern coverage**
  How much of the calibration pattern must at least be contained in the images depends on the used plate.
  - Plate with hexagonally arranged marks: At least one finder pattern needs to be visible. If at least two finder patterns are visible in the image, it is possible to detect whether the calibration plate is mirrored or not. In a mirrored case, a suitable error will be returned.
  - Plate with rectangularly arranged marks: Plate needs to be completely visible, as the finder pattern is the frame surrounding the point marks.

  Nevertheless, of course, the more of the calibration pattern is visible to the camera and the more of the field of view is filled by the calibration plate, the better.
- **Mark diameter**
  The marks of the calibration plates should have a diameter of at least 20 pixels in each image. This requirement is essential for a successful calibration.
- **Contrast**
  The contrast between the light and dark areas of the calibration plate should be at least 100 gray values (regarding byte images).

- **Overexposure**
  To avoid overexposed images, make sure that gray values of the light parts of the calibration plate do not exceed 240 (regarding byte images), especially not in the neighborhood of the calibration marks.
- **Homogeneity**
  The calibration plate should be illuminated homogeneously and reflections should be avoided. As a rule of thumb, the range of gray values of the light parts of the plate should not exceed 45 (regarding byte images).

Regarding image format and preprocessing:

- **Image format**
  Calibration images should be saved in an uncompressed format. Compression artifacts which occur, e.g., when using JPG format and high compression rates need to be avoided.
- **Preprocessing**
  Calibration images should not be preprocessed. If image properties like contrast or focus are insufficient (see above), the issues need to be resolved by adjusting the camera setup instead of processing the images ahead of the calibration.

Which distortion model should be used? Two distortion models can be used: The division model and the polynomial model. The division model uses one parameter to model the radial distortions while the polynomial model uses five parameters to model radial and decentering distortions (see the sections "Camera parameters" and "The Used 3D camera model").

The advantages of the division model are that the distortions can be applied faster, especially the inverse distortions, i.e., if world coordinates are projected into the image plane. Furthermore, if only few calibration images are used or if the field of view is not covered sufficiently, the division model typically yields more stable results than the polynomial model. The main advantage of the polynomial model is that it can model the distortions more accurately because it uses higher order terms to model the radial distortions and because it also models the decentering distortions. Note that the polynomial model cannot be inverted analytically. Therefore, the inverse distortions must be calculated iteratively, which is slower than the calculation of the inverse distortions with the (analytically invertible) division model.

Typically, the division model should be used for the calibration. If the accuracy of the calibration is not high enough, the polynomial model can be used. Note, however, that the calibration sequence used for the polynomial model must provide an even better coverage of the area in which measurements will later be performed. The distortions may be modeled inaccurately outside of the area that was covered by the calibration plate. This holds for the image border as well as for areas inside the field of view that were not covered by the calibration plate.

The Used 3D Camera Model

In general, camera calibration means the exact determination of the parameters that model the (optical) projection of any 3D world point $\mathbf{p}^w$ into a (sub-)pixel (r, c) in the image. This is important if the original 3D pose of an object must be computed from the image (e.g., for measuring industrial parts). The appropriate projection model depends on the camera type used in your setup.

For the modeling of this projection process, which is determined by the used combination of camera, lens, and frame grabber, HALCON provides the following 3D camera models:

Area scan pinhole camera: The combination of an area scan camera with a lens that effects a perspective projection on the object side of the lens and that may show radial and decentering distortions. The lens may be a tilt lens, i.e., the optical axis of the lens may be tilted with respect to the camera's sensor (this is sometimes called a Scheimpflug lens). Since hypercentric lenses also perform a perspective projection, cameras with hypercentric lenses are pinhole cameras. The models for regular (i.e., non-tilt) pinhole and image-side telecentric lenses are identical. In contrast, the models for pinhole and image-side telecentric tilt lenses differ substantially, as described below.

Area scan telecentric camera: The combination of an area scan camera with a lens that is telecentric on the object-side of the lens, i.e., that effects a parallel projection on the object-side of the lens, and that may show radial and decentering distortions. The lens may be a tilt lens. The models for regular (i.e., non-tilt) bilateral and object-side telecentric lenses are identical. In contrast, the models for bilateral and object-side telecentric tilt lenses differ substantially, as described below.

Line scan pinhole camera: The combination of a line scan camera with a lens that effects a perspective projection and that may show radial distortions. Tilt lenses are currently not supported for line scan cameras.

Line scan telecentric camera: The combination of a line scan camera with a lens that effects a telecentric projection and that may show radial distortions. Tilt lenses are currently not supported for line scan cameras.

To transform a 3D point $\mathbf{p}^w = (\mathbf{x}^w, \mathbf{y}^w, \mathbf{z}^w)^T$ which is given in world coordinates, into a 2D point $\mathbf{q}^i = (\mathbf{r}, \mathbf{c})^T$, which is given in pixel coordinates, a chain of transformations is needed:

$$\mathbf{p}^w \rightarrow \mathbf{p}^c \rightarrow \mathbf{q}^c \rightarrow \tilde{\mathbf{q}}^c \left[\rightarrow \mathbf{q}^t\right] \rightarrow \mathbf{q}^i$$

$\mathbf{p}^w$	3D world point
$\mathbf{p}^c$	Transformed into camera coordinate system
$\mathbf{q}^c$	Projected into image plane (2D point, still in metric coordinates)
$\tilde{\mathbf{q}}^c$	Lens distortion applied
$\mathbf{q}^t$	If a tilted lens is used, the point $\tilde{\mathbf{q}}^c$ is projected on the point $\mathbf{q}^t$ in the tilted image plane. In this case the distorted point $\tilde{\mathbf{q}}^c$ only lies on a virtual image plane of a system without tilt.
$\mathbf{q}^i$	Pixel coordinates

The following paragraphs describe these steps in more detail for area scan cameras and subsequently for line scan cameras. For a even more detailed description of the different 3D camera models as well as some explanatory diagrams please refer to the chapter "Basics", section "Camera Model and Parameters" in the "`Solution Guide III-C 3D Vision`".

Transformation step 1: $\mathbf{p}^w \rightarrow \mathbf{p}^c$ The point $\mathbf{p}^w$ is transformed from world into camera coordinates (points as homogeneous vectors, compare `affine_trans_point_3d`) by :

$$\begin{pmatrix} \mathbf{p}^c \\ 1 \end{pmatrix} = \begin{pmatrix} x^c \\ y^c \\ z^c \\ 1 \end{pmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ 0\,0\,0 & 1 \end{bmatrix} \cdot \begin{pmatrix} \mathbf{p}^w \\ 1 \end{pmatrix}$$

with $\mathbf{R}$ and $\mathbf{T}$ being the rotation and translation matrices (refer to the chapter "Basics", section "3D Transformations and Poses'" in the "`Solution Guide III-C 3D Vision`" for detailed information).

Transformation step 2: $\mathbf{p}^c \rightarrow \mathbf{q}^c$ If the underlying camera model is an **area scan pinhole camera**, the projection of $\mathbf{p}^c = (\mathbf{x}^c, \mathbf{y}^c, \mathbf{z}^c)^T$ into the image plane is described by the following equation:

$$\mathbf{q}^c = \begin{pmatrix} u \\ v \end{pmatrix} = \frac{f}{\mathbf{z}^c} \begin{pmatrix} \mathbf{x}^c \\ \mathbf{y}^c \end{pmatrix}$$

where $f =$ `Focus`. For cameras with hypercentric lenses, the following equation holds instead:

$$\mathbf{q}^c = \begin{pmatrix} u \\ v \end{pmatrix} = \frac{-f}{\mathbf{z}^c} \begin{pmatrix} \mathbf{x}^c \\ \mathbf{y}^c \end{pmatrix}$$

If an **area scan telecentric camera** is used, the corresponding equation is:

$$\mathbf{q}^c = \begin{pmatrix} u \\ v \end{pmatrix} = m \begin{pmatrix} \mathbf{x}^c \\ \mathbf{y}^c \end{pmatrix}$$

where $m =$ `Magnification`.

Transformation step 3: $\mathbf{q}^c \rightarrow \tilde{\mathbf{q}}^c$ For all types of cameras, the lens distortions can be modeled either by the division model or by the polynomial model.

The **division model** uses one parameter `Kappa` to model the radial distortions.

The following equations transform the distorted image plane coordinates into undistorted image plane coordinates if the division model is used:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{1+\kappa(\tilde{u}^2+\tilde{v}^2)} \begin{pmatrix} \tilde{u} \\ \tilde{v} \end{pmatrix}$$

These equations can be inverted analytically, which leads to the following equations that transform undistorted coordinates into distorted coordinates:

$$\tilde{\mathbf{q}}^c = \begin{pmatrix} \tilde{u} \\ \tilde{v} \end{pmatrix} = \frac{2}{1+\sqrt{1-4\kappa(u^2+v^2)}} \begin{pmatrix} u \\ v \end{pmatrix}$$

The **polynomial model** uses three parameters (K_1, K_2, K_3) to model the radial distortions and two parameters (P_1, P_2) to model the decentering distortions.

The following equations transform the distorted image plane coordinates into undistorted image plane coordinates if the polynomial model is used:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \tilde{u}+\tilde{u}(K_1r^2+K_2r^4+K_3r^6)+P_1(r^2+2\tilde{u}^2)+2P_2\tilde{u}\tilde{v} \\ \tilde{v}+\tilde{v}(K_1r^2+K_2r^4+K_3r^6)+2P_1\tilde{u}\tilde{v}+P_2(r^2+2\tilde{v}^2) \end{pmatrix}$$

with $r = \sqrt{\tilde{u}^2+\tilde{v}^2}$

These equations cannot be inverted analytically. Therefore, distorted image plane coordinates must be calculated from undistorted image plane coordinates numerically.

Additional transformation step for tilt lenses: $\tilde{\mathbf{q}}^c \rightarrow \mathbf{q}^t$ If the camera lens is a **tilt lens**, the tilt of the lens with respect to the image plane is described by the rotation angle rot and the tilt angle $tilt$.

In this step you have to further distinguish between different types of tilt lenses as described below. See chapter "Basics", section "Camera Model and Parameters" in the "`Solution Guide III-C 3D Vision`" for an overview of different types of tilt lenses.

For **projective tilt lenses** and **object-side telecentric tilt lenses** (which perform a perspective projection on the image side of the lens) the projection of $\tilde{\mathbf{q}}^c = (\tilde{u}, \tilde{v})^T$ into the point $\mathbf{q}^t = (\hat{u}, \hat{v})^T$, which lies in the tilted image plane, is described by a projective 2D transformation, i.e., by the homogeneous 3×3 matrix $\mathbf{H}$ (see `projective_trans_point_2d`):

$$\begin{pmatrix} \mathbf{q}^t \\ q_w^t \end{pmatrix} = \mathbf{H} \cdot \begin{pmatrix} \tilde{\mathbf{q}}^c \\ 1 \end{pmatrix}$$

where q_w^t is the additional coordinate from the projective transformation of a homogeneous point.

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} = \begin{pmatrix} q_{11}q_{33}-q_{13}q_{31} & q_{21}q_{33}-q_{23}q_{31} & 0 \\ q_{12}q_{33}-q_{13}q_{32} & q_{22}q_{33}-q_{23}q_{32} & 0 \\ q_{13}/d & q_{23}/d & q_{33} \end{pmatrix}$$

where $d =$ `ImagePlaneDist` and

$$\begin{aligned} \mathbf{Q} &= \begin{pmatrix} q_{11} & q_{12} & q_{13} \\ q_{21} & q_{22} & q_{23} \\ q_{31} & q_{32} & q_{33} \end{pmatrix} \\ &= \begin{pmatrix} (\cos\rho)^2(1-\cos\tau)+\cos\tau & \cos\rho\sin\rho(1-\cos\tau) & \sin\rho\sin\tau \\ \cos\rho\sin\rho(1-\cos\tau) & (\sin\rho)^2(1-\cos\tau)+\cos\tau & -\cos\rho\sin\tau \\ -\sin\rho\sin\tau & \cos\rho\sin\tau & \cos\tau \end{pmatrix} \end{aligned}$$

with $\rho = rot$ and $\tau = tilt$.

For **image-side telecentric tilt lenses** and **bilateral telecentric tilt lenses** (which perform a parallel projection on the image side of the lens), the projection onto the tilted image plane is described by a linear 2D transformation, i.e., by a 2×2 matrix:

$$\mathbf{H} = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix} = \frac{1}{q_{11}q_{22} - q_{12}q_{21}} \begin{pmatrix} q_{22} & -q_{12} \\ -q_{21} & q_{11} \end{pmatrix}$$

where $\mathbf{Q}$ is defined as above for projective lenses.

Transformation step 4: $\mathbf{q}^t \rightarrow \mathbf{q}^i$ / $\tilde{\mathbf{q}}^c \rightarrow \mathbf{q}^i$ Finally, the point $\tilde{\mathbf{q}}^c = (\tilde{u}, \tilde{v})^T$ (or $\mathbf{q}^t$ if a tilt lens is present) is transformed from the image plane coordinate system into the image coordinate system (the pixel coordinate system):

$$\mathbf{q}^i = \begin{pmatrix} r \\ c \end{pmatrix} = \begin{pmatrix} \frac{\tilde{v}}{S_y} + C_y \\ \frac{\tilde{u}}{S_x} + C_x \end{pmatrix}$$

For **line scan cameras**, also the relative motion between the camera and the object must be modeled. In HALCON, the following assumptions for this motion are made:

1. The camera moves with constant velocity along a straight line.
2. The orientation of the camera is constant.
3. The motion is equal for all images.

The motion is described by the motion vector $V = (V_x, V_y, V_z)^T$ that must be given in [meter/row] in the camera coordinate system. The motion vector describes the motion of the camera, assuming a fixed object. In fact, this is equivalent to the assumption of a fixed camera with the object traveling along $-V$.

The camera coordinate system of line scan cameras is defined as follows: The origin of the coordinate system is the center of projection (for pinhole cameras) or the center of distortion (for telecentric cameras), respectively. The z-axis is identical to the optical axis and directed so that the visible points have positive z coordinates. The y-axis is perpendicular to the sensor line and to the z-axis. It is directed so that the motion vector has a positive y-component. The x-axis is perpendicular to the y- and z-axis, so that the x-, y-, and z-axis form a right-handed coordinate system.

As the camera moves over the object during the image acquisition, also the camera coordinate system moves relatively to the object, i.e., each image line has been imaged from a different position. This means there would be an individual pose for each image line. To make things easier, in HALCON all transformations from world coordinates into camera coordinates and vice versa are based on the pose of the first image line only. The motion V is taken into account during the projection of the point $\mathbf{p}^c$ into the image. Consequently, only the pose of the first image line is computed by the operator `find_calib_object` (and stored by `calibrate_cameras` in the calibration results).

For **line scan cameras**, the transformation from world to camera coordinates ($\mathbf{p}^w \rightarrow \mathbf{p}^c$) works in the same way. Therefore, you can also apply transformation step 1 as described for area scan cameras above.

For **line scan pinhole cameras**, the projection of the point $\mathbf{p}^c$ that is given in the camera coordinate system into (sub-)pixel coordinates (r, c) in the image is modeled as follows:

Assuming

$$\mathbf{p}^c = \begin{pmatrix} x \\ y \\ z \end{pmatrix},$$

the following set of equations must be solved for m, $\tilde{u}$, and t:

$$\begin{aligned} m \cdot u(\tilde{u}, p_v) &= x - t \cdot V_x \\ m \cdot v(\tilde{u}, p_v) &= y - t \cdot V_y \\ m \cdot \text{Focus} &= z - t \cdot V_z \end{aligned}$$

were $u(\tilde{u}, \tilde{v})$ and $v(\tilde{u}, \tilde{v})$ are the undistortion functions that are described above for area scan cameras and $p_v = -S_y \cdot C_y$.

For **line scan telecentric cameras**, the following set of equations must be solved for $\tilde{u}$ and t:

$$\begin{aligned} u(\tilde{u}, p_v)/\text{Magnification} &= x - t \cdot V_x \\ v(\tilde{u}, p_v)/\text{Magnification} &= y - t \cdot V_y \end{aligned}$$

with $u(\tilde{u}, \tilde{v})$, $v(\tilde{u}, \tilde{v})$ and p_v as defined above. Note that neither z nor V_z influence the projection for telecentric cameras.

The above formulas already include the compensation for image distortions.

Finally, the point is transformed into the image coordinate system, i.e., the pixel coordinate system:

$$\mathbf{q}^i = \begin{pmatrix} r \\ c \end{pmatrix} = \begin{pmatrix} t \\ \frac{\tilde{u}}{S_x} + C_x \end{pmatrix}.$$

Further Limitations Related to Specific Camera Types

For **pinhole cameras**, if the calibration plates are parallel to each other in all images (in particular, if they all lie in the same plane), it is impossible to determine `Focus` together with all six of the external camera parameters. For example, it is impossible to determine `Focus` and the distance of the calibration plates to the camera in this case. To be able to calibrate all camera parameters uniquely, make sure that you acquire images of the calibration plate tilted in different orientations.

For **telecentric lenses**, the distance of the calibration plate from the camera cannot be determined. Therefore, the z-component of the resulting calibration plate pose is set to 1 m in the calibration results. Furthermore, as described previously, for telecentric line scan cameras, V_z cannot be determined and is left at its initial value, except for multi-camera setups that have a common motion vector, in which case V_z can be determined.

For **tilt lenses**, the greater the lens distortion is, the more accurately the tilt can be determined. For lenses with small distortions, the tilt cannot be determined robustly. Therefore, the optimized tilt parameters may differ significantly from the nominal tilt parameters of the setup. If this is the case, please check `Error`. If `Error` is small, the resulting camera parameters describe the imaging geometry consistently within the calibrated volume and can be used for accurate measurements.

For **perspective tilt lenses** and **object-side telecentric tilt lenses**, the image plane distance can only be determined uniquely if the tilt is not 0 degrees. The smaller the tilt, the less accurately the image plane distance can be determined. Therefore, the optimized image plane distance may differ significantly from the nominal image plane distance of the setup. If this is the case, please check `Error`. If `Error` is small, the resulting camera parameters describe the imaging geometry consistently within the calibrated volume and can be used for accurate measurements.

For **perspective tilt lenses** and **object-side telecentric tilt lenses** that are tilted around the horizontal or vertical axis, i.e., for which the rotation angle is 0, 90, 180, or 270 degrees, the tilt angle $tilt$, scale factor S_x, the focal length f (for perspective tilt lenses) or the magnification m (for object-side telecentric tilt lenses), and the distance of the tilted image plane from the perspective projection center d cannot be determined uniquely. In this case, S_x should be excluded from the optimization by calling

```
set_calib_data(CalibDataID, 'camera', 'general', 'excluded_settings', 'sx').
```

Additionally, note that for tilt lenses it is only possible to determine $tilt$ and rot simultaneously. This is an implementation choice that makes the optimization numerically more robust. Consequently, the parameters $tilt$ and rot are excluded simultaneously from the optimization by calling

`set_calib_data(CalibDataID, 'camera', 'general', 'excluded_settings', 'tilt')`.

Pinhole cameras with tilt lenses with large focal lengths have nearly telecentric projection characteristics. Therefore, as described before, S_x and the tilt parameters $tilt$ and rot are correlated and can only be determined imprecisely simultaneously. In this case, it is again advisable to exclude S_x from the optimization.

For telecentric lenses, there are always two possible poses of a calibration plate for a single image. Therefore, it is not possible to decide which one of the two poses is actually present in the image. This ambiguity also effects the tilt parameters $tilt$ and rot of a telecentric tilt lens. Consequently, depending on the initial parameters for $tilt$ and rot the camera calibration may return the alternative parameters instead of the nominal ones. If this is the case, please check `Error`. If `Error` is small, the resulting camera parameters describe the imaging geometry consistently within the calibrated volume and can be used for accurate measurements.

For **line scan cameras with the polynomial distortion model** (for cameras with perspective as well as telecentric lenses), the parameters P_1 and P_2 are highly correlated with other parameters in the camera model. Therefore, they typically cannot be determined reliably and should be excluded from the calibration by calling

`set_calib_data(CalibDataID, 'camera', 'general', 'excluded_settings', 'poly_tan_2')`.

Further Information

Learn about camera calibration any many other topics in interactive online courses at our MVTec Academy .

6.1 Binocular

binocular_calibration (: : NX, NY, NZ, NRow1, NCol1, NRow2, NCol2, StartCamParam1, StartCamParam2, NStartPose1, NStartPose2, EstimateParams : CamParam1, CamParam2, NFinalPose1, NFinalPose2, RelPose, Errors)

Determine all camera parameters of a binocular stereo system.

In general, binocular calibration means the exact determination of the parameters that model the 3D reconstruction of a 3D point from the corresponding images of this point in a binocular stereo system. This reconstruction is specified by the internal parameters `CamParam1` of camera 1 and `CamParam2` of camera 2 describing the underlying camera model, and the external parameters `RelPose` describing the relative pose of camera system 2 in relation to camera system 1.

Thus, known 3D model points (with coordinates `NX`, `NY`, `NZ`) are projected in the image planes of both cameras (camera 1 and camera 2) and the sum of the squared distances between these projections and the corresponding measured image points (with coordinates `NRow1`, `NCol1` for camera 1 and `NRow2`, `NCol2` for camera 2) is minimized. It should be noted that all these model points must be visible in both images. The used camera model is described in Calibration. The camera model is represented (for each camera separately) by a tuple of 9 to 16 parameters that correspond to perspective or telecentric area scan or telecentric line scan cameras (see Calibration). The projection uses the initial values `StartCamParam1` and `StartCamParam2` of the internal parameters of camera 1 and camera 2, which can be obtained from the camera data sheets. In addition, the initial guesses `NStartPose1` and `NStartPose2` of the poses of the 3D calibration model in relative to the camera coordinate systems (*ccs*) of camera 1 and camera 2 are needed as well. These poses are expected in the form ${}^{ccs}\mathbf{P}_{wcs}$, where *wcs* denotes the world coordinate system (see Transformations / Poses and "`Solution Guide III-C - 3D Vision`"). They can be determined by the operator `find_marks_and_pose`. Since this calibration algorithm simultaneously handles correspondences between measured image and known model points from different image pairs, poses (`NStartPose1`,`NStartPose2`), and measured points (`NRow1`,`NCol1`,`NRow2`, `NCol2`) must be passed concatenated in a corresponding order.

The input parameter `EstimateParams` is used to select the parameters to be estimated. Usually this parameter is set to *'all'*, i.e., all external camera parameters (translation and rotation) and all internal camera parameters are determined. Otherwise, `EstimateParams` contains a tuple of strings indicating the combination of parameters to estimate. For instance, if the internal camera parameters already have been determined (e.g., by previous calls to `binocular_calibration`), it is often desired to only determine relative the pose of the two cameras to each other (`RelPose`). In this case, `EstimateParams` can be set to *'pose_rel'*. The internal parameters can be

subsumed by the parameter values *'cam_param1'* and *'cam_param2'* as well. Note that if the polynomial model is used to model the lens distortions, the values *'k1_i'*, *'k2_i'* and *'k3_i'* can be specified individually, whereas *'p1'* and *'p2'* can only be specified in the group *'poly_tan_2_i'* (with *'i'* indicating the index of the camera). *'poly_i'* specifies the group *'k1_i'*, *'k2_i'*, *'k3_i'* and *'poly_tan_2_i'*.

The following list contains all possible strings that can be passed to the tuple:

Allowed strings for `EstimateParams`	*Determined parameters*
'all' (default)	All internal camera parameters, as well as the relative pose of both cameras and the poses of the calibration objects.
'pose'	Relative pose between the two cameras and poses of the calibration objects.
'pose_rel'	Relative pose between the two cameras.
'alpha_rel', *'beta_rel'*, *'gamma_rel'*, *'transx_rel'*, *'transy_rel'*, *'transz_rel'*	Rotation angles and translation parameters of the relative pose between the two cameras.
'pose_caltabs'	Poses of the calibration objects.
'alpha_caltabs', *'beta_caltabs'*, *'gamma_caltabs'*, *'transx_caltabs'*, *'transy_caltabs'*, *'transz_caltabs'*	Rotation angles and translation parameters of the relative poses of the calibration objects.
'cam_param1', *'cam_param2'*	All internal camera parameters of camera 1 and camera 2, respectively.
'focus1', *'magnification1'*, *'kappa1'*, *'poly_1'*, *'k1_1'*, *'k2_1'*, *'k3_1'*, *'poly_tan_2_1'*, *'image_plane_dist1'*, *'tilt1'*, *'cx1'*, *'cy1'*, *'sx1'*, *'sy1'*, *'focus2'*, *'magnification2'*, *'kappa2'*, *'poly_2'*, *'k1_2'*, *'k2_2'*, *'k3_2'*, *'poly_tan_2_2'*, *'image_plane_dist2'*, *'tilt2'*, *'cx2'*, *'cy2'*, *'sx2'*, *'sy2'*	Individual internal camera parameters of camera 1 and camera 2, respectively.
'common_motion_vector'	Determines whether two line scan cameras have a common motion vector. This is the case if the two cameras are mounted rigidly and the object is moved linearly in front of the cameras or if the two rigidly mounted cameras are moved by the same linear actuator. This is assumed to be the default. Therefore, you only need to set '~common_motion_vector' if the cameras are moving independently in different directions.

In addition, parameters can be excluded from estimation by using the prefix '~'. For example, the values *['pose_rel','~transx_rel']* have the same effect as *['alpha_rel','beta_rel','gamma_rel','transy_rel','transz_rel']*. On the other hand, *['all','~focus1']* determines all internal and external parameters except the focus of camera 1, for instance. The prefix '~' can be used with all parameter values except *'all'*.

The underlying camera model is explained in the chapter Calibration. The calibrated internal camera parameters are returned in `CamParam1` for camera 1 and in `CamParam2` for camera 2.

The external parameters are returned analogously to `camera_calibration`, the 3D transformation poses of the calibration model to the respective camera coordinate system (*ccs*) are returned in `NFinalPose1` and `NFinalPose2`. Thus, the poses are in the form ${}^{ccs}\mathbf{P}_{wcs}$, where *wcs* denotes the world coordinate system of the 3D calibration model (see Transformations / Poses and "`Solution Guide III-C - 3D Vision`"). The relative pose ${}^{ccs1}\mathbf{P}_{ccs2}$, `RelPose`, specifies the transformation of points in *ccs2* into *ccs1*. Therewith, the final poses are related with each other (neglecting differences due to the balancing effects of the multi image calibration) by:

HomMat3D_NFinalPose2 = INV(HomMat3D_RelPose) * HomMat3D_NFinalPose1,

where HomMat3D_* denotes a homogeneous transformation matrix of the respective poses and INV() inverts a homogeneous matrix.

The computed average errors returned in `Errors` give an impression of the accuracy of the calibration. Using the determined camera parameters, they denote the average euclidean distance between the projection of the mark centers to their extracted image coordinates.

For cameras with telecentric lenses, additional conditions must be fulfilled for the setup. They can be found in the chapter Calibration.

Attention

Stereo setups that contain cameras with and without hypercentric lenses at the same time are not supported. Furthermore, stereo setups that contain area scan and line scan cameras at the same time are not supported.

Parameters

- ▷ **NX** (input_control) . . . . . number-array ⇝ *real* / integer
  Ordered Tuple with all X-coordinates of the calibration marks (in meters).
- ▷ **NY** (input_control) . . . . . number-array ⇝ *real* / integer
  Ordered Tuple with all Y-coordinates of the calibration marks (in meters).
  Number of elements: NY == NX
- ▷ **NZ** (input_control) . . . . . number-array ⇝ *real* / integer
  Ordered Tuple with all Z-coordinates of the calibration marks (in meters).
  Number of elements: NZ == NX
- ▷ **NRow1** (input_control) . . . . . number-array ⇝ *real* / integer
  Ordered Tuple with all row-coordinates of the extracted calibration marks of camera 1 (in pixels).
- ▷ **NCol1** (input_control) . . . . . number-array ⇝ *real* / integer
  Ordered Tuple with all column-coordinates of the extracted calibration marks of camera 1 (in pixels).
  Number of elements: NCol1 == NRow1
- ▷ **NRow2** (input_control) . . . . . number-array ⇝ *real* / integer
  Ordered Tuple with all row-coordinates of the extracted calibration marks of camera 2 (in pixels).
  Number of elements: NRow2 == NRow1
- ▷ **NCol2** (input_control) . . . . . number-array ⇝ *real* / integer
  Ordered Tuple with all column-coordinates of the extracted calibration marks of camera 2 (in pixels).
  Number of elements: NCol2 == NRow1
- ▷ **StartCamParam1** (input_control) . . . . . campar ⇝ *real* / integer / string
  Initial values for the internal parameters of camera 1.
- ▷ **StartCamParam2** (input_control) . . . . . campar ⇝ *real* / integer / string
  Initial values for the internal parameters of camera 2.
- ▷ **NStartPose1** (input_control) . . . . . pose(-array) ⇝ *real* / integer
  Ordered tuple with all initial values for the poses of the calibration model in relation to camera 1.
  Number of elements: NStartPose1 == 7 * NRow1 / NX
- ▷ **NStartPose2** (input_control) . . . . . pose(-array) ⇝ *real* / integer
  Ordered tuple with all initial values for the poses of the calibration model in relation to camera 2.
  Number of elements: NStartPose2 == 7 * NRow1 / NX
- ▷ **EstimateParams** (input_control) . . . . . string-array ⇝ *string*
  Camera parameters to be estimated.
  Default: 'all'
  List of values: `EstimateParams` ∈ {'all', 'pose', 'pose_caltabs', 'pose_rel', 'cam_param1', 'cam_param2', 'alpha_rel', 'beta_rel', 'gamma_rel', 'transx_rel', 'transy_rel', 'transz_rel', 'alpha_caltabs', 'beta_caltabs', 'gamma_caltabs', 'transx_caltabs', 'transy_caltabs', 'transz_caltabs', 'focus1', 'magnification1', 'kappa1', 'poly_1', 'k1_1', 'k2_1', 'k3_1', 'poly_tan_2_1', 'image_plane_dist1', 'tilt1', 'cx1', 'cy1', 'sx1', 'sy1', 'focus2', 'magnification2', 'kappa2', 'poly_2', 'k1_2', 'k2_2', 'k3_2', 'poly_tan_2_2', 'image_plane_dist2', 'tilt2', 'cx2', 'cy2', 'sx2', 'sy2', 'common_motion_vector'}
- ▷ **CamParam1** (output_control) . . . . . campar ⇝ *real* / integer / string
  Internal parameters of camera 1.
- ▷ **CamParam2** (output_control) . . . . . campar ⇝ *real* / integer / string
  Internal parameters of camera 2.
- ▷ **NFinalPose1** (output_control) . . . . . pose(-array) ⇝ *real* / integer
  Ordered tuple with all poses of the calibration model in relation to camera 1.
  Number of elements: NFinalPose1 == 7 * NRow1 / NX
- ▷ **NFinalPose2** (output_control) . . . . . pose(-array) ⇝ *real* / integer
  Ordered tuple with all poses of the calibration model in relation to camera 2.
  Number of elements: NFinalPose2 == 7 * NRow1 / NX

▷ **RelPose** (output_control) . . . . . pose ⇝ *real* / integer
Pose of camera 2 in relation to camera 1.

▷ **Errors** (output_control) . . . . . real(-array) ⇝ *real*
Average error distances in pixels.

Example

```
* Open image source.
open_framegrabber ('File', 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1, \
                   'default', 'images_l.seq', 'default', 0, -1, AcqHandle1)
open_framegrabber ('File', 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1, \
                   'default', 'images_r.seq', 'default', 1, -1, AcqHandle2)

* Initialize the start parameters.
caltab_points ('caltab_30mm.descr', X, Y, Z)
StartCamParam1 := ['area_scan_division', 0.0125, 0, 7.4e-6, 7.4e-6, \
                   Width/2.0, Height/2.0, Width, Height]
StartCamParam2 := StartCamParam1
Rows1 := []
Cols1 := []
StartPoses1 := []
Rows2 := []
Cols2 := []
StartPoses2 := []

* Find calibration marks and startposes.
for i := 0 to 11 by 1
  grab_image_async (Image1, AcqHandle1, -1)
  grab_image_async (Image2, AcqHandle2, -1)
  find_caltab (Image1, CalPlate1, 'caltab_30mm.descr', 3, 120, 5)
  find_caltab (Image2, CalPlate2, 'caltab_30mm.descr', 3, 120, 5)
  find_marks_and_pose (Image1, CalPlate1, 'caltab_30mm.descr', \
                       StartCamParam1, 128, 10, 20, 0.7, 5, 100, \
                       RCoord1, CCoord1, StartPose1)
  Rows1 := [Rows1,RCoord1]
  Cols1 := [Cols1,CCoord1]
  StartPoses1 := [StartPoses1,StartPose1]
  find_marks_and_pose (Image2, CalPlate2, 'caltab_30mm.descr', \
                       StartCamParam2, 128, 10, 20, 0.7, 5, 100, \
                       RCoord2, CCoord2, StartPose2)
  Rows2 := [Rows2,RCoord2]
  Cols2 := [Cols2,CCoord2]
  StartPoses2 := [StartPoses2,StartPose2]
endfor

* Calibrate the stereo rig.
binocular_calibration (X, Y, Z, Rows1, Cols1, Rows2, Cols2, StartCamParam1, \
                       StartCamParam2, StartPoses1, StartPoses2, 'all', \
                       CamParam1, CamParam2, NFinalPose1, NFinalPose2, \
                       RelPose, Errors)
* Archive the results.
write_cam_par (CamParam1, 'cam_left-125.dat')
write_cam_par (CamParam2, 'cam_right-125.dat')
write_pose (RelPose, 'rel_pose.dat')

* Rectify the stereo images.
gen_binocular_rectification_map (Map1, Map2, CamParam1, CamParam2, \
                                 RelPose, 1, 'viewing_direction', 'bilinear', \
                                 CamParamRect1, CamParamRect2, \
                                 CamPoseRect1, CamPoseRect2, \
```

```
                                    RelPoseRect)
map_image (Image1, Map1, ImageMapped1)
map_image (Image2, Map2, ImageMapped2)
```

Result

`binocular_calibration` returns `2` (`H_MSG_TRUE`) if all parameter values are correct and the desired parameters have been determined by the minimization algorithm. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`find_marks_and_pose`, `caltab_points`, `read_cam_par`

Possible Successors

`write_pose`, `write_cam_par`, `pose_to_hom_mat3d`, `disp_caltab`, `gen_binocular_rectification_map`

See also

`find_caltab`, `sim_caltab`, `read_cam_par`, `create_pose`, `convert_pose_type`, `read_pose`, `hom_mat3d_to_pose`, `create_caltab`, `binocular_disparity`, `binocular_distance`

Module

3D Metrology

6.2 Calibration Object

caltab_points (: : CalPlateDescr : X, Y, Z)

Read the mark center points from the calibration plate description file.

`caltab_points` reads the mark center points from the calibration plate description file `CalPlateDescr` (see `gen_caltab` for calibration plates with rectangularly arranged marks and `create_caltab` for calibration plates with hexagonally arranged marks) and returns their coordinates in `X`, `Y` and `Z`. The mark center points are 3D coordinates in the calibration plate coordinate system and describe the 3D model of the calibration plate. The calibration plate coordinate system is located in the middle of the surface of the calibration plate for calibration plates with rectangularly arranged marks and at the center of the central mark of the first finder pattern for calibration plates with hexagonally arranged marks. Its z-axis points into the calibration plate, its x-axis to the right, and its y-axis downwards.

The mark center points are typically used as input parameters for the operator `camera_calibration`.

Parameters

▷ **CalPlateDescr** (input_control) . . . . . filename.read ⇝ *string*
File name of the calibration plate description.
Default: 'calplate_320mm.cpd'
List of values: `CalPlateDescr` ∈ {'calplate_5mm.cpd', 'calplate_10mm.cpd', 'calplate_20mm.cpd', 'calplate_40mm.cpd', 'calplate_80mm.cpd', 'calplate_160mm.cpd', 'calplate_320mm.cpd', 'calplate_640mm.cpd', 'calplate_1200mm.cpd', 'calplate_20mm_dark_on_light.cpd', 'calplate_40mm_dark_on_light.cpd', 'calplate_80mm_dark_on_light.cpd', 'caltab_650um.descr', 'caltab_2500um.descr', 'caltab_6mm.descr', 'caltab_10mm.descr', 'caltab_30mm.descr', 'caltab_100mm.descr', 'caltab_200mm.descr', 'caltab_800mm.descr', 'caltab_small.descr', 'caltab_big.descr'}
File extension: `.cpd`, `.descr`

▷ **X** (output_control) . . . . . real-array ⇝ *real*
X coordinates of the mark center points in the coordinate system of the calibration plate.

▷ **Y** (output_control) . . . . . real-array ⇝ *real*
Y coordinates of the mark center points in the coordinate system of the calibration plate.

▷ **Z** (output_control) . . . . . real-array ⇝ *real*
Z coordinates of the mark center points in the coordinate system of the calibration plate.

Example

```
* Read calibration image.
read_image(Image, 'calib/calib-3d-coord-03')
CalTabDescr := 'caltab_100mm.descr'
* Find calibration pattern.
find_caltab(Image, CalPlate1, CalTabDescr, 3, 112, 5)
* Find calibration marks and start poses.
StartCamPar := ['area_scan_division', 0.008, 0.0, 0.000011, 0.000011, \
                384, 288, 768, 576]
find_marks_and_pose(Image,CalPlate1,CalTabDescr, StartCamPar, \
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord1, CCoord1, \
                    StartPose1)
* Read 3D positions of calibration marks.
caltab_points(CalTabDescr, NX, NY, NZ)
*  Calibrate camera.
camera_calibration(NX, NY, NZ, RCoord1, CCoord1, StartCamPar, \
                   StartPose1, 'all', CameraParam, FinalPose, Errors)
*  Visualize calibration result.
dev_display(Image)
disp_caltab(WindowHandle, CalTabDescr, CameraParam, FinalPose, 1.0)
```

Result

`caltab_points` returns `2` (`H_MSG_TRUE`) if all parameter values are correct and the file `CalPlateDescr` has been read successfully. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`camera_calibration`

See also

`find_caltab`, `find_marks_and_pose`, `camera_calibration`, `disp_caltab`, `sim_caltab`, `project_3d_point`, `get_line_of_sight`, `gen_caltab`

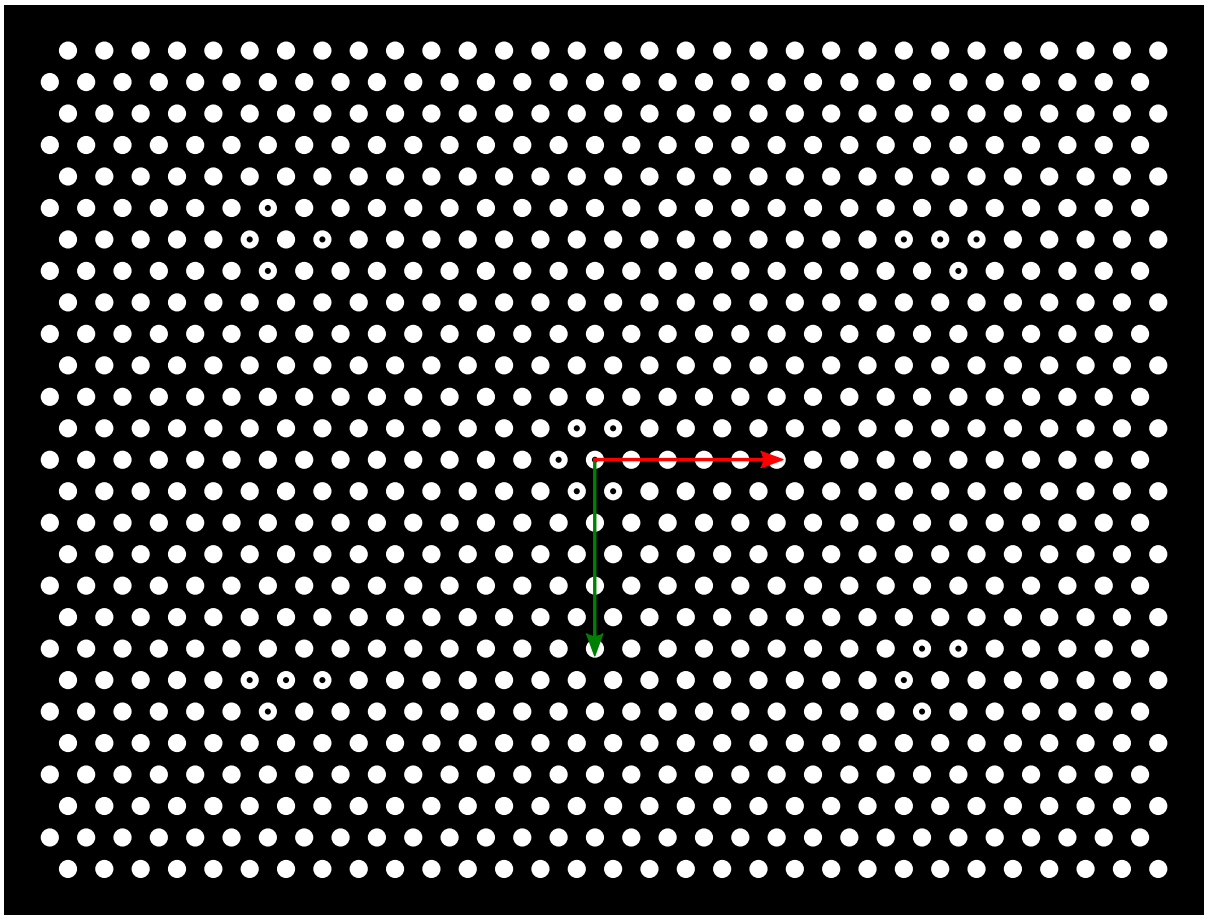
Module

Foundation

create_caltab (: : NumRows, MarksPerRow, Diameter, FinderRow, FinderColumn, Polarity, CalPlateDescr, CalPlatePSFile :)

Generate a calibration plate description file and a corresponding PostScript file for a calibration plate with hexagonally arranged marks.

`create_caltab` creates the description file of a standard HALCON calibration plate with hexagonally arranged marks. This calibration plate contains `MarksPerRow` times `NumRows` circular marks. These marks are arranged in a hexagonal lattice such that each mark (except the ones at the border) has six equidistant neighbors.



A standard HALCON calibration plate with hexagonally arranged marks and its coordinate system.

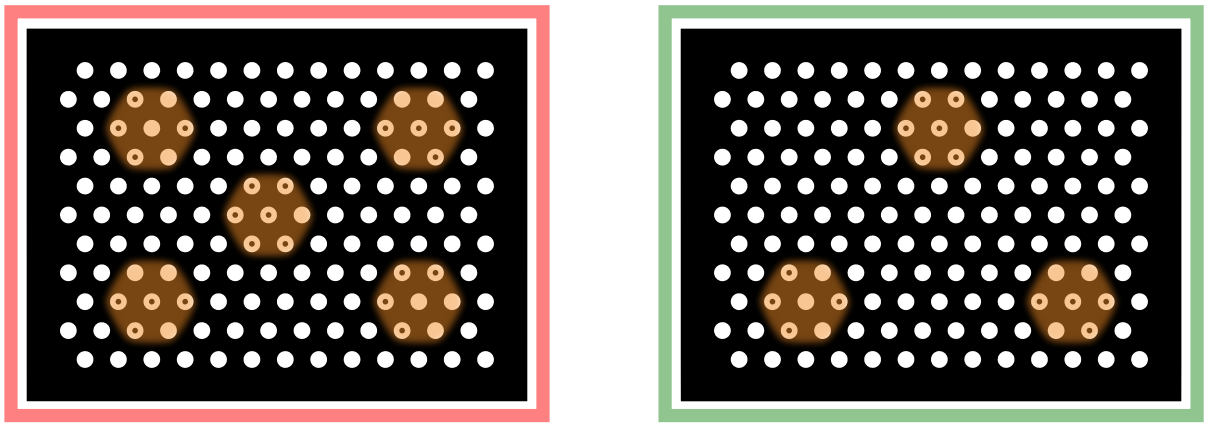
The diameter of the marks is given by the parameter `Diameter` in meters. The distance between the centers of horizontally neighboring calibration marks is given by $2 \cdot Diameter$. The distance between neighboring rows of calibration marks is given by $2 \cdot Diameter \cdot \sqrt{0.75}$. The width and the height of the generated calibration plate can be calculated with the following equations:

$$\texttt{Width} = (2 \left\lfloor \frac{\texttt{MarksPerRow}-1}{2} \right\rfloor + 3) \cdot 2 \cdot \texttt{Diameter}$$

$$\texttt{Height} = (2 \left\lfloor \frac{\texttt{NumRows}-1}{2} \right\rfloor \cdot \sqrt{3} + 5) \cdot \texttt{Diameter}$$

The calibration plate contains one to five finder patterns. A finder pattern is a special mark hexagon (i.e. a mark and its six neighbors) where either four or six marks contain a hole. Each of these up to five finder patterns is unique such that it can be used to determine the orientation of the calibration plate and the position of the finder pattern on the calibration plate. As a consequence, the calibration plate can only be found by `find_calib_object` if at least one of these finder patterns is completely visible. The position of the central mark of each finder pattern is given in `FinderRow` and `FinderColumn`. Thus, the length of the tuples given in `FinderRow` and `FinderColumn`, respectively determine the number of finder patterns on the calibration plate. Be aware that two finder patterns must not overlap. It is recommended to keep a certain distance between the finder patterns, so every mark containing a hole can be assigned to a finder pattern distinctly. As a rule of thumb, if the calibration plate contains too few marks to place all finder patterns in distinct positions, it is better to reduce the number of finder patterns so that they can be distributed more evenly. An example case is depicted below, but note that a successful detection of the patterns also depends on the used camera setup.

The coordinate system of the calibration plate is located in the center of the central mark of the first finder pattern.



The finder patterns on a calibration plate should not be too close to each other (left). If there are not enough marks on your plate to distribute the finder patterns further apart you should reduce the number of finder patterns (right).

Depending on `Polarity` the marks are either light on dark background (for *'light_on_dark'*, which is the default) or dark on light background (for *'dark_on_light'*).

The file `CalPlateDescr` contains the calibration plate description, and must be passed to all HALCON operations using the generated calibration plate (e.g., `set_calib_data_calib_object` or `sim_caltab`). The default HALCON file extension for the description of a calibration plate with hexagonally arranged marks is 'cpd'.

A calibration plate description file contains information about:

- the number of row and columns of the calibration plate
- the number of marks per row and column
- the offset of the coordinate system to the plate's surface in z-direction
- the rim of the calibration plate
- the polarity of the marks
- the number and position of finder patterns
- the x,y coordinates and radius of the calibration marks

A file generated by `create_caltab` looks like the following (comments are marked by a '#' at the beginning of a line):

```
\# Plate Description Version 3
\# HALCON Version 20.11 --  Wed Dec 16 11:02:00 2020
\# Description of the standard calibration plate
\# used for the camera calibration in HALCON
\# (generated by create\_caltab)
\#
\#

\# 27 rows x 31 columns
\# Width, height of calibration plate [meter]: 0.170323, 0.129118
\# Distance between mark centers [meter]: 0.0051613

\# Number of marks in y-dimension (rows)
r 27

\# Number of marks in x-dimension (columns)
c 31

\# offset of coordinate system in z-dimension [meter] (optional):
z 0

\# rim of the calibration plate (min x, max y, max x, min y) [meter]:
```

```
o -0.083871125 0.0645592449151841 0.086451775 -0.0645592449151841

\# polarity of the marks (light or dark):
p light

\# number of finder pattern marks:
f 5

\# position of the finder patterns (central mark):  x y [index]
15 13
6 6
24 6
6 20
24 20

\# calibration marks:  x y radius [meter]

\# calibration marks at y = -0.0581076 m
-0.07483885 -0.0581076199151841 0.001290325
-0.06967755 -0.0581076199151841 0.001290325
-0.06451625 -0.0581076199151841 0.001290325
-0.05935495 -0.0581076199151841 0.001290325
-0.05419365 -0.0581076199151841 0.001290325
-0.04903235 -0.0581076199151841 0.001290325
-0.04387105 -0.0581076199151841 0.001290325
-0.03870975 -0.0581076199151841 0.001290325
-0.03354845 -0.0581076199151841 0.001290325
-0.02838715 -0.0581076199151841 0.001290325
-0.02322585 -0.0581076199151841 0.001290325
-0.01806455 -0.0581076199151841 0.001290325
-0.01290325 -0.0581076199151841 0.001290325
-0.00774195 -0.0581076199151841 0.001290325
-0.00258065 -0.0581076199151841 0.001290325
0.00258065 -0.0581076199151841 0.001290325
0.00774195 -0.0581076199151841 0.001290325
0.01290325 -0.0581076199151841 0.001290325
0.01806455 -0.0581076199151841 0.001290325
0.02322585 -0.0581076199151841 0.001290325
0.02838715 -0.0581076199151841 0.001290325
0.03354845 -0.0581076199151841 0.001290325
0.03870975 -0.0581076199151841 0.001290325
0.04387105 -0.0581076199151841 0.001290325
0.04903235 -0.0581076199151841 0.001290325
0.05419365 -0.0581076199151841 0.001290325
0.05935495 -0.0581076199151841 0.001290325
0.06451625 -0.0581076199151841 0.001290325
0.06967755 -0.0581076199151841 0.001290325
0.07483885 -0.0581076199151841 0.001290325
0.08000015 -0.0581076199151841 0.001290325

\# calibration marks at y = -0.0536378 m
-0.0774195 -0.0536378029986315 0.001290325
-0.0722582 -0.0536378029986315 0.001290325
-0.0670969 -0.0536378029986315 0.001290325
-0.0619356 -0.0536378029986315 0.001290325
-0.0567743 -0.0536378029986315 0.001290325
-0.051613 -0.0536378029986315 0.001290325
-0.0464517 -0.0536378029986315 0.001290325
-0.0412904 -0.0536378029986315 0.001290325
```

```
-0.0361291 -0.0536378029986315 0.001290325
-0.0309678 -0.0536378029986315 0.001290325
-0.0258065 -0.0536378029986315 0.001290325
-0.0206452 -0.0536378029986315 0.001290325
-0.0154839 -0.0536378029986315 0.001290325
-0.0103226 -0.0536378029986315 0.001290325
-0.0051613 -0.0536378029986315 0.001290325
0 -0.0536378029986315 0.001290325
0.0051613 -0.0536378029986315 0.001290325
0.0103226 -0.0536378029986315 0.001290325
0.0154839 -0.0536378029986315 0.001290325
0.0206452 -0.0536378029986315 0.001290325
0.0258065 -0.0536378029986315 0.001290325
0.0309678 -0.0536378029986315 0.001290325
0.0361291 -0.0536378029986315 0.001290325
0.0412904 -0.0536378029986315 0.001290325
0.0464517 -0.0536378029986315 0.001290325
0.051613 -0.0536378029986315 0.001290325
0.0567743 -0.0536378029986315 0.001290325
0.0619356 -0.0536378029986315 0.001290325
0.0670969 -0.0536378029986315 0.001290325
0.0722582 -0.0536378029986315 0.001290325
0.0774195 -0.0536378029986315 0.001290325

\# calibration marks at y = -0.049168 m
...

\# calibration marks at y = -0.0446982 m
...

\# calibration marks at y = -0.0402284 m
...

\# calibration marks at y = -0.0357585 m
...

\# calibration marks at y = -0.0312887 m
...

\# calibration marks at y = -0.0268189 m
...

\# calibration marks at y = -0.0223491 m
...

\# calibration marks at y = -0.0178793 m
...

\# calibration marks at y = -0.0134095 m
...

\# calibration marks at y = -0.00893963 m
...

\# calibration marks at y = -0.00446982 m
...

\# calibration marks at y = 0 m
...
```

```
\# calibration marks at y = 0.00446982 m
...

\# calibration marks at y = 0.00893963 m
...

\# calibration marks at y = 0.0134095 m
...

\# calibration marks at y = 0.0178793 m
...

\# calibration marks at y = 0.0223491 m
...

\# calibration marks at y = 0.0268189 m
...

\# calibration marks at y = 0.0312887 m
...

\# calibration marks at y = 0.0357585 m
...

\# calibration marks at y = 0.0402284 m
...

\# calibration marks at y = 0.0446982 m
...

\# calibration marks at y = 0.049168 m
...

\# calibration marks at y = 0.0536378 m
...

\# calibration marks at y = 0.0581076 m
...
```

Note that only the coordinates and radius of the marks in the first two rows are listed completely. The corresponding coordinates and radius of the marks in the other rows are omitted for a better overview.

The file `CalPlatePSFile` contains the corresponding PostScript description of the calibration plate, which can be used to print the calibration plate.

Attention

Depending on the accuracy of the used output device (e.g., laser printer), a printed calibration plate may not match the values in the calibration plate description file `CalPlateDescr` exactly. Thus, the coordinates of the calibration marks in the calibration plate description file may have to be corrected!

For purchased calibration plates it is recommended to use the specific calibration description file that is supplied with your calibration plate.

Parameters

▷ **NumRows** (input_control) . . . . . integer ⇝ *integer*
Number of rows.
Default: 27
Recommended increment: 1
Restriction: NumRows > 2

▷ **MarksPerRow** (input_control) . . . . . integer ⇝ *integer*
Number of marks per row.
Default: 31
Recommended increment: 1
Restriction: MarksPerRow > 2

▷ **Diameter** (input_control) . . . . . real ⇝ *real*
Diameter of the marks.
Default: 0.00258065
Suggested values: `Diameter` ∈ {0.00258065, 0.1, 0.0125, 0.00375, 0.00125}

▷ **FinderRow** (input_control) . . . . . integer(-array) ⇝ *integer*
Row indices of the finder patterns.
Default: [13,6,6,20,20]

▷ **FinderColumn** (input_control) . . . . . integer(-array) ⇝ *integer*
Column indices of the finder patterns.
Default: [15,6,24,6,24]

▷ **Polarity** (input_control) . . . . . string ⇝ *string*
Polarity of the marks
Default: 'light_on_dark'
Suggested values: `Polarity` ∈ {'light_on_dark', 'dark_on_light'}

▷ **CalPlateDescr** (input_control) . . . . . filename.write ⇝ *string*
File name of the calibration plate description.
Default: 'calplate.cpd'
List of values: `CalPlateDescr` ∈ {'calplate.cpd'}
File extension: `.cpd`

▷ **CalPlatePSFile** (input_control) . . . . . filename.write ⇝ *string*
File name of the PostScript file.
Default: 'calplate.ps'
File extension: `.ps`

Example

```
* Parameters to create the descriptor for the 160mm wide calibration
* plate.
create_caltab (27, 31, 0.00258065, [13, 6, 6, 20, 20], [15, 6, 24, 6, 24], \
              'light_on_dark', 'calplate.cpd', 'caltab.ps')
```

Result

`create_caltab` returns `2` (`H_MSG_TRUE`) if all parameter values are correct and both files have been written successfully. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`read_cam_par`, `caltab_points`

Alternatives

`gen_caltab`

See also

`find_caltab`, `find_marks_and_pose`, `camera_calibration`, `disp_caltab`, `sim_caltab`

Module

Foundation

disp_caltab (: : WindowHandle, CalPlateDescr, CameraParam, CalPlatePose, ScaleFac :)

Project and visualize the 3D model of the calibration plate in the image.

disp_caltab is used to visualize the calibration marks and the connecting lines between the marks of the used calibration plate (CalPlateDescr) in the window specified by WindowHandle. Additionally, the x- and y-axes of the plate's coordinate system are printed on the plate's surface. For this, the 3D model of the calibration plate is projected into the image plane using the internal (CameraParam) and external camera parameters (CalPlatePose). Thereby the pose is in the form ${}^{ccs}\mathbf{P}_{wcs}$, where *ccs* denotes the camera coordinate system and *wcs* the world coordinate system (see Transformations / Poses and "Solution Guide III-C - 3D Vision"), thus the pose of the calibration plate in camera coordinates. The underlying camera model is described in Calibration.

Typically, disp_caltab is used to verify the result of the camera calibration (see Calibration or camera_calibration) by superimposing it onto the original image. The current line width can be set by set_line_width, the current color can be set by set_color. Additionally, the font type of the labels of the coordinate axes can be set by set_font.

The parameter ScaleFac influences the number of supporting points to approximate the elliptic contours of the calibration marks. You should increase the number of supporting points, if the image part in the output window is displayed with magnification (see set_part).

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window in which the calibration plate should be visualized.

▷ **CalPlateDescr** (input_control) ..... filename.read ⇝ *string*
File name of the calibration plate description.
Default: 'calplate_320.cpd'
List of values: CalPlateDescr ∈ {'calplate_5mm.cpd', 'calplate_10mm.cpd', 'calplate_20mm.cpd', 'calplate_40mm.cpd', 'calplate_80mm.cpd', 'calplate_160mm.cpd', 'calplate_320mm.cpd', 'calplate_640mm.cpd', 'calplate_1200mm.cpd', 'calplate_20mm_dark_on_light.cpd', 'calplate_40mm_dark_on_light.cpd', 'calplate_80mm_dark_on_light.cpd', 'caltab_650um.descr', 'caltab_2500um.descr', 'caltab_6mm.descr', 'caltab_10mm.descr', 'caltab_30mm.descr', 'caltab_100mm.descr', 'caltab_200mm.descr', 'caltab_800mm.descr', 'caltab_small.descr', 'caltab_big.descr'}
File extension: .cpd, .descr

▷ **CameraParam** (input_control) ..... campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **CalPlatePose** (input_control) ..... pose ⇝ *real* / integer
External camera parameters (3D pose of the calibration plate in camera coordinates).
Number of elements: 7

▷ **ScaleFac** (input_control) ..... real ⇝ *real*
Scaling factor for the visualization.
Default: 1.0
Suggested values: ScaleFac ∈ {0.5, 1.0, 2.0, 3.0}
Recommended increment: 0.05
Restriction: 0.0 < ScaleFac

Example

```
* Read image of calibration plate.
read_image (Image, 'calib/calib_single_camera_01')
get_image_size (Image, Width, Height)
* Create and setup the calibration model.
create_calib_data ('calibration_object', 1, 1, CalibDataID)
CalPlateDescr := 'calplate_80mm.cpd'
set_calib_data_calib_object (CalibDataID, 0, CalPlateDescr)
CamParam := ['area_scan_division', 0.008, -1500, 3.7e-6, 3.7e-6, \
             640, 470, 1292, 964]
set_calib_data_cam_param (CalibDataID, 0, [], CamParam)
```

```
* Localize calibration plate in the image.
find_calib_object (Image, CalibDataID, 0, 0, 0, [], [])
get_calib_data_observ_pose (CalibDataID, 0, 0, 0, StartPose)
* Display calibration plate.
disp_caltab (WindowHandle, CalPlateDescr, CamParam, StartPose, 1)
```

Result

`disp_caltab` returns 2 (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`camera_calibration`, `read_cam_par`, `read_pose`

See also

`find_marks_and_pose`, `camera_calibration`, `sim_caltab`, `write_cam_par`, `read_cam_par`, `create_pose`, `write_pose`, `read_pose`, `project_3d_point`, `get_line_of_sight`

Module

Foundation

find_calib_object (Image : : CalibDataID, CameraIdx, CalibObjIdx, CalibObjPoseIdx, GenParamName, GenParamValue :)

Find the HALCON calibration plate and set the extracted points and contours in a calibration data model.

`find_calib_object` searches in `Image` for a HALCON calibration plate corresponding to the description of the calibration object with the index `CalibObjIdx` from the calibration data model `CalibDataID`. If a calibration plate is found, `find_calib_object` extracts the centers and the contours of its marks and estimates the pose of the plate relative to the observing camera `CameraIdx`. All collected observation data is stored in the calibration data model for the calibration object pose `CalibObjPoseIdx`. In order to ensure a successful detection of the calibration plate, at least one finder pattern has to be visible in the image. For calibration plates with hexagonally arranged marks this is a special mark hexagon where either four or six marks contain a hole, while for calibration plates with rectangularly arranged marks this is the border of the calibration plate with a triangle in one corner.

Preparation of the input data

Before the operator `find_calib_object` can be called, a calibration data model has to be defined performing the following steps:

1. **Create a calibration data model** with the operator `create_calib_data`, specifying the number of cameras in the setup and the number of used calibration objects.
2. **Specify the camera type and the initial internal camera parameters** for all cameras with the operator `set_calib_data_cam_param`. Note that only cameras of the same type can be calibrated in a single setup.
3. **Specify the description of all calibration objects** with the operator `set_calib_data_calib_object`. Note that for a successful call of `find_calib_object` a valid description file of the calibration plate is necessary. This description file has to be set beforehand via the operator `set_calib_data_calib_object`. As a consequence, the usage of a user-defined calibration object can only be made by the operator `set_calib_data_observ_points`.

Collecting observation data

`find_calib_object` is used to collect observations in a calibration data model. Beyond, it stores additional observation data that cannot be added to the model with `set_calib_data_observ_points` and that is dependent on the used calibration plate. While for calibration plates with rectangularly arranged marks (see

gen_caltab) the rim of the calibration plate is added to the observations, calibration plates with hexagonal pattern (see create_caltab) store one of their finder pattern. Additionally and irrespective of the used calibration plate, the contour of each mark is added to the calibration model.

Setting additional parameters

Using **calibration plates with hexagonally arranged marks**, the following additional parameter can be set via GenParamName and GenParamValue:

'sigma': Smoothing factor for the extraction of the mark contours. For increasing values of *'sigma'*, the filter width and thereby the amount of smoothing increases (see also edges_sub_pix for the influence of the filter width on the Canny filter).

Suggested values: *0.5*, *0.7*, *0.9*, *1.0*, *1.2*, *1.5*

Default: *1.0*

For **calibration plates with rectangularly arranged marks**, find_calib_object essentially encapsulates the sequence of three operator calls: find_caltab, find_marks_and_pose and set_calib_data_observ_points. For this kind of calibration plates the following parameters can be set using GenParamName and GenParamValue:

'alpha': Smoothing factor for the extraction of the mark contours. For increasing values of *'alpha'*, the filter width and thereby the amount of smoothing decreases (see also edges_sub_pix for the influence of the filter width on the Lanser2 filter).

Suggested values: *0.5*, *0.7*, *0.9*, *1.0*, *1.2*, *1.5*

Default: *0.9*

'gap_tolerance': Tolerance factor for gaps between the marks. If the marks appear closer to each other than expected, you might set *'gap_tolerance'* < *1.0* to avoid disturbing patterns outside the calibration plate to be associated with the calibration plate. This can typically happen if the plate is strongly tilted and positioned in front of a background that exposes mark-like patterns. If the distances between single marks vary in a wide range, e.g., if the calibration plate appears with strong perspective distortion in the image, you might set *'gap_tolerance'* > *1.0* to enforce the marks grouping (see also find_caltab).

Suggested values: *0.75*, *0.9*, *1.0*, *1.1*, *1.2*, *1.5*

Default: *1.0*

'max_diam_marks': Maximum expected diameter of the marks (needed internally by find_marks_and_pose). By default, this value is estimated by the preceding internal call to find_caltab. However, if the estimation is erroneous for no obvious reason or the internal call to find_caltab fails or is simply skipped (see *'skip_find_caltab'* below), you might have to adjust this value.

Suggested values: *50.0*, *100.0*, *150.0*, *200.0*, *300.0*

'skip_find_caltab': Skip the internal call to find_caltab. If activated, only the domain of Image reduces the search area for the internal call of find_marks_and_pose. Thus, a user defined calibration plate region can be incorporated by setting *'skip_find_caltab'*=*'false'* and reducing the Image domain to the user region.

List of values: *'false'*, *'true'*

Default: *'false'*

If using a HALCON calibration plate as calibration object, it is recommended to use find_calib_object instead of set_calib_data_observ_points where possible, since the contour information, which it stores in the calibration data model, enables a more precise calibration procedure with calibrate_cameras.

After a successful call to find_calib_object, the extracted points can be queried by get_calib_data_observ_points and the extracted contours can be accessed by get_calib_data_observ_contours.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image.

▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **CameraIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observing camera.
Default: 0
Suggested values: CameraIdx ∈ {0, 1, 2}

▷ **CalibObjIdx** (input_control) . . . . . number ⇝ *integer*
Index of the calibration object.
Default: 0
Suggested values: CalibObjIdx ∈ {0, 1, 2}

▷ **CalibObjPoseIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observed calibration object.
Default: 0
Suggested values: CalibObjPoseIdx ∈ {0, 1, 2}
Restriction: CalibObjPoseIdx >= 0

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters to be set.
Default: []
List of values: GenParamName ∈ {'gap_tolerance', 'alpha', 'sigma', 'max_diam_marks', 'skip_find_caltab'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / real / integer
Values of the generic parameters to be set.
Default: []
Suggested values: GenParamValue ∈ {0.5, 0.9, 1.0, 1.2, 1.5, 2.0, 'true', 'false'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

read_image, find_marks_and_pose, set_calib_data_cam_param, set_calib_data_calib_object

Possible Successors

set_calib_data, calibrate_cameras

Alternatives

find_caltab, find_marks_and_pose, set_calib_data_observ_points

Module

Calibration

find_caltab (Image : CalPlate : CalPlateDescr, SizeGauss, MarkThresh, MinDiamMarks :)

Segment the region of a standard calibration plate with rectangularly arranged marks in the image.

find_caltab is used to determine the region of a plane calibration plate with circular marks in the input image Image. The region must correspond to a standard calibration plate with rectangularly arranged marks described in the file CalPlateDescr. The successfully segmented region is returned in CalPlate. The operator provides two algorithms. By setting appropriate integer values in SizeGauss, MarkThresh, and MinDiamMarks, respectively, you invoke the *standard algorithm*. If you pass a tuple of parameter names in SizeGauss and a corresponding tuple of parameter values in MarkThresh, or just two empty tuples, respectively, you invoke the *advanced algorithm* instead. In this case the value passed in MinDiamMarks is ignored.

Standard algorithm

First, the input image is smoothed (see gauss_image); the size of the used filter mask is given by SizeGauss. Afterwards, a threshold operator (see threshold) with a minimum gray value MarkThresh is applied. Among the extracted connected regions the most convex region with an almost correct number of holes (corresponding to the dark marks of the calibration plate) is selected. Holes with a diameter smaller than the expected size of the marks MinDiamMarks are eliminated to reduce the impact of noise. The number of marks is read from the calibration plate description file CalPlateDescr. The complete explanation of this file can be found within the description of gen_caltab.

Advanced algorithm

First, an image pyramid based on Image is built. Starting from the highest pyramid level, round regions are segmented with a dynamic threshold. Then, they are associated in groups based on their mutual proximity and it is evaluated whether they can represent marks of a potential calibration plate. The search is terminated once the expected number of marks has been identified in one group. The surrounding lighter area is returned in CalPlate.

The image pyramid makes the search independent from the size of the image and the marks. The dynamic threshold makes the algorithm immune to bad or irregular illumination. Therefore, in general, no parameter is required. Yet, you can adjust some auxiliary parameters of the advanced algorithm by passing a list of parameter names (strings) to SizeGauss and a list of corresponding parameter values to MarkThresh. Currently the following parameter is supported:

'gap_tolerance': Tolerance factor for gaps between the marks. If the marks appear closer to each other than expected, you might set *'gap_tolerance'* < *1.0* to avoid disturbing patterns outside the calibration plate to be associated with the calibration plate. This can typically happen if the plate is strongly tilted and positioned in front of a background that exposes mark-like patterns. If the distances between single marks deviate significantly, e.g., if the calibration plate appears with strong perspective distortion in the image, you might set *'gap_tolerance'* > *1.0* to enforce the grouping for the more distant marks.

Suggested values: *0.75*, *0.9*, *1.0*, *1.1*, *1.2*, *1.5*

Default: *1.0*

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object* : byte / uint2
Input image.

▷ **CalPlate** (output_object) ..... region ⇝ *object*
Output region.

▷ **CalPlateDescr** (input_control) ..... filename.read ⇝ *string*
File name of the calibration plate description.
Default: 'caltab_100.descr'
List of values: CalPlateDescr ∈ {'caltab_650um.descr', 'caltab_2500um.descr', 'caltab_6mm.descr', 'caltab_10mm.descr', 'caltab_30mm.descr', 'caltab_100mm.descr', 'caltab_200mm.descr', 'caltab_800mm.descr', 'caltab_small.descr', 'caltab_big.descr'}
File extension: .descr

▷ **SizeGauss** (input_control) ..... integer(-array) ⇝ *integer* / string
Filter size of the Gaussian.
Default: 3
List of values: SizeGauss ∈ {0, 3, 5, 7, 9, 11, 'gap_tolerance'}

▷ **MarkThresh** (input_control) ..... integer(-array) ⇝ *integer* / real
Threshold value for mark extraction.
Default: 112
Suggested values: MarkThresh ∈ {48, 64, 80, 96, 112, 128, 144, 160, 0.5, 0.9, 1.0, 1.1, 1.5}

▷ **MinDiamMarks** (input_control) ..... integer ⇝ *integer*
Expected minimal diameter of the marks on the calibration plate.
Default: 5
Suggested values: MinDiamMarks ∈ {3, 5, 9, 15, 30, 50, 70}

Example

```
* Read calibration image.
read_image(Image, 'calib/calib_distorted_01')
```

```
* Find calibration pattern.
find_caltab(Image, CalPlate, 'caltab_100mm.descr', 3, 112, 5)
```

Result

`find_caltab` returns `2` (`H_MSG_TRUE`) if all parameter values are correct and an image region is found. The behavior in case of empty input (no image given) can be set via `set_system(::'no_object_result',<Result>:)` and the behavior in case of an empty result region via `set_system(::'store_empty_region',<'true'/'false'>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`read_image`

Possible Successors

`find_marks_and_pose`

See also

`find_marks_and_pose`, `camera_calibration`, `disp_caltab`, `sim_caltab`, `caltab_points`, `gen_caltab`

Module

Foundation

find_marks_and_pose (Image, CalPlateRegion : : CalPlateDescr, StartCamParam, StartThresh, DeltaThresh, MinThresh, Alpha, MinContLength, MaxDiamMarks : RCoord, CCoord, StartPose)

Extract rectangularly arranged 2D calibration marks from the image and calculate initial values for the external camera parameters.

`find_marks_and_pose` is used to determine the input data for a subsequent camera calibration using a calibration plate with rectangularly arranged marks (see Calibration or `camera_calibration`): First, the 2D center points [`RCoord`,`CCoord`] of the calibration marks within the region `CalPlateRegion` of the input image `Image` are extracted and ordered. Secondly, a rough estimate for the external camera parameters (`StartPose`) is computed, i.e., the 3D pose (= position and orientation) of the calibration plate relative to the camera coordinate system (see `create_pose` for more information about 3D poses).

In the input image `Image` an edge detector is applied (see `edges_image`, mode 'lanser2') to the region `CalPlateRegion`, which can be found by applying the operator `find_caltab`. The filter parameter for this edge detection can be tuned via `Alpha`. Use a smaller value for `Alpha` to achieve a stronger smoothing effect. In the edge image closed contours are searched for: The number of closed contours must correspond to the number of calibration marks as described in the calibration plate description file `CalPlateDescr` and the contours have to be elliptically shaped. Contours shorter than `MinContLength` are discarded, just as contours enclosing regions with a diameter larger than `MaxDiamMarks` (e.g., the border of the calibration plate).

For the detection of contours a threshold operator is applied on the resulting amplitudes of the edge detector. All points with a high amplitude (i.e., borders of marks) are selected.

First, the threshold value is set to `StartThresh`. If the search for the closed contours or the successive pose estimate fails, this threshold value is successively decreased by `DeltaThresh` down to a minimum value of `MinThresh`.

Each of the found contours is refined with subpixel accuracy (see `edges_sub_pix`) and subsequently approximated by an ellipse. The center points of these ellipses represent a good approximation of the desired 2D image coordinates [`RCoord`,`CCoord`] of the calibration mark center points. The order of the values within these two tuples must correspond to the order of the 3D coordinates of the calibration marks in the calibration plate description file `CalPlateDescr`, since this fixes the correspondences between extracted image marks and known model marks (given by `caltab_points`)! If a triangular orientation mark is defined in a corner of the plate by the

plate description file (see gen_caltab), the mark will be detected and the point order is returned in row-major order beginning with the corner mark in the (barycentric) negative quadrant with respect to the defined coordinate system of the plate. Else, if no orientation mark is defined, the order of the center points is in row-major order beginning at the upper left corner mark in the image.

Based on the ellipse parameters for each calibration mark, a rough estimate for the external camera parameters is finally computed. For this purpose the fixed correspondences between extracted image marks and known model marks are used. The estimate StartPose describes the pose of the calibration plate in the camera coordinate system as required by the operator camera_calibration.

Parameters

- ▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image.
- ▷ **CalPlateRegion** (input_object) . . . . . region ⇝ *object*
  Region of the calibration plate.
- ▷ **CalPlateDescr** (input_control) . . . . . filename.read ⇝ *string*
  File name of the calibration plate description.
  Default: 'caltab_100.descr'
  List of values: CalPlateDescr ∈ {'caltab_650um.descr', 'caltab_2500um.descr', 'caltab_6mm.descr', 'caltab_10mm.descr', 'caltab_30mm.descr', 'caltab_100mm.descr', 'caltab_200mm.descr', 'caltab_800mm.descr', 'caltab_small.descr', 'caltab_big.descr'}
  File extension: .descr
- ▷ **StartCamParam** (input_control) . . . . . campar ⇝ *real* / integer / string
  Initial values for the internal camera parameters.
- ▷ **StartThresh** (input_control) . . . . . number ⇝ *integer*
  Initial threshold value for contour detection.
  Default: 128
  Suggested values: StartThresh ∈ {80, 96, 112, 128, 144, 160}
  Restriction: StartThresh > 0
- ▷ **DeltaThresh** (input_control) . . . . . number ⇝ *integer*
  Loop value for successive reduction of StartThresh.
  Default: 10
  Suggested values: DeltaThresh ∈ {6, 8, 10, 12, 14, 16, 18, 20, 22}
  Restriction: DeltaThresh > 0
- ▷ **MinThresh** (input_control) . . . . . number ⇝ *integer*
  Minimum threshold for contour detection.
  Default: 18
  Suggested values: MinThresh ∈ {8, 10, 12, 14, 16, 18, 20, 22}
  Restriction: MinThresh > 0
- ▷ **Alpha** (input_control) . . . . . real ⇝ *real*
  Filter parameter for contour detection, see edges_image.
  Default: 0.9
  Suggested values: Alpha ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1}
  Value range: 0.2 ≤ Alpha ≤ 2.0
  Restriction: Alpha > 0.0
- ▷ **MinContLength** (input_control) . . . . . real ⇝ *real*
  Minimum length of the contours of the marks.
  Default: 15.0
  Suggested values: MinContLength ∈ {10.0, 15.0, 20.0, 30.0, 40.0, 100.0}
  Restriction: MinContLength > 0.0
- ▷ **MaxDiamMarks** (input_control) . . . . . real ⇝ *real*
  Maximum expected diameter of the marks.
  Default: 100.0
  Suggested values: MaxDiamMarks ∈ {50.0, 100.0, 150.0, 200.0, 300.0}
  Restriction: MaxDiamMarks > 0.0
- ▷ **RCoord** (output_control) . . . . . real-array ⇝ *real*
  Tuple with row coordinates of the detected marks.
- ▷ **CCoord** (output_control) . . . . . real-array ⇝ *real*
  Tuple with column coordinates of the detected marks.

▷ **StartPose** (output_control) . . . . . pose ⇝ *real* / integer
Estimation for the external camera parameters.
Number of elements: 7

Example

```
* Read calibration image.
read_image(Image, 'calib/calib_distorted_01')
* Find calibration pattern.
find_caltab(Image,CalPlate, 'caltab_100mm.descr', 3, 112, 5)
* Find calibration marks and start pose.
find_marks_and_pose(Image, CalPlate, 'caltab_100mm.descr' , \
                    ['area_scan_division', 0.008, 0.0, \
                     0.000011, 0.000011, 384, 288, 640, 512], \
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord, CCoord, StartPose)
```

Result

`find_marks_and_pose` returns `2` (`H_MSG_TRUE`) if all parameter values are correct and an estimation for the external camera parameters has been determined successfully. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`find_caltab`

Possible Successors

`camera_calibration`

See also

`find_caltab`, `camera_calibration`, `disp_caltab`, `sim_caltab`, `read_cam_par`, `read_pose`, `create_pose`, `pose_to_hom_mat3d`, `caltab_points`, `gen_caltab`, `edges_sub_pix`, `edges_image`

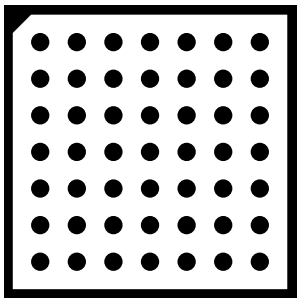
Module

Foundation

gen_caltab (: : XNum, YNum, MarkDist, DiameterRatio, CalPlateDescr, CalPlatePSFile :)

Generate a calibration plate description file and a corresponding PostScript file for a calibration plate with rectangularly arranged marks.

`gen_caltab` generates the description of a standard HALCON calibration plate with rectangularly arranged marks. This calibration plate consists of `XNum` times `YNum` black circular marks on a white plane which are surrounded by a black frame.



A standard HALCON calibration plate with rectangularly arranged marks

The marks are arranged in a rectangular grid with YNum and XNum equidistant rows and columns. The distances between these rows and columns defines the parameter MarkDist in meter. The marks' diameter can be set by the parameter DiameterRatio and is defined by the equation Diameter = MarkDist · DiameterRatio . Using a distance between marks of 0.01 m and a diameter ratio of 0.5, the width of the dark surrounding frame becomes 8 cm, and the radius of the marks is set to 2.5 mm. The coordinate system of the calibration plate is located in the barycenter of all marks, its z-axis points into the calibration plate, its x-axis to the right, and its y-axis downwards.

The black frame of the calibration plate encloses a triangular black orientation mark in the top left corner to uniquely determine the position of the calibration plate. The width and the height of the generated calibration plate can be calculated with the following equations:

$\texttt{Width} = \texttt{MarkDist} \cdot (\texttt{XNum} + 1)$

$\texttt{Height} = \texttt{MarkDist} \cdot (\texttt{YNum} + 1)$

The file CalPlateDescr contains the calibration plate description, e.g., the number of rows and columns of the calibration plate, the geometry of the surrounding frame (see find_caltab), the triangular orientation mark, an offset of the coordinate system to the plate's surface in z-direction, and the x,y coordinates and the radius of all calibration plate marks given in the calibration plate coordinate system. The definition of the orientation and the offset, indicated by t and z, is optional and can be commented out. The default HALCON file extension for the calibration plate description is 'descr'. A file generated by gen_caltab looks like the following (comments are marked by a '#' at the beginning of a line):

```
\# Plate Description Version 2
\# HALCON Version 7.1 --  Fri Jun 24 16:41:00 2005
\# Description of the standard calibration plate
\# used for the camera calibration in HALCON
\# (generated by gen\_caltab)
\#
\#
\# 7 rows x 7 columns
\# Width, height of the black frame [meter]: 0.1, 0.1
\# Distance between mark centers [meter]: 0.0125

\# Number of marks in y-dimension (rows)
r 7

\# Number of marks in x-dimension (columns)
c 7

\#   offset of coordinate system in z-dimension [meter] (optional):
z 0

\# Rectangular border (rim and black frame) of calibration plate
\#   rim of the calibration plate (min x, max y, max x, min y) [meter]:
o -0.05125 0.05125 0.05125 -0.05125
\#   outer border of the black frame (min x, max y, max x, min y) [meter]:

i -0.05 0.05 0.05 -0.05
\#   triangular corner mark given by two corner points (x,y, x,y) [meter]
\#   (optional):
t -0.05 -0.0375 -0.0375 -0.05

\#   width of the black frame [meter]:
w 0.003125

\# calibration marks:  x y radius [meter]

\# calibration marks at y = -0.0375 m
-0.0375 -0.0375 0.003125
-0.025 -0.0375 0.003125
-0.0125 -0.0375 0.003125
```

```
-3.46945e-018 -0.0375 0.003125
0.0125 -0.0375 0.003125
0.025 -0.0375 0.003125
0.0375 -0.0375 0.003125

\# calibration marks at y = -0.025 m
-0.0375 -0.025 0.003125
-0.025 -0.025 0.003125
-0.0125 -0.025 0.003125
-3.46945e-018 -0.025 0.003125
0.0125 -0.025 0.003125
0.025 -0.025 0.003125
0.0375 -0.025 0.003125

\# calibration marks at y = -0.0125 m
-0.0375 -0.0125 0.003125
-0.025 -0.0125 0.003125
-0.0125 -0.0125 0.003125
-3.46945e-018 -0.0125 0.003125
0.0125 -0.0125 0.003125
0.025 -0.0125 0.003125
0.0375 -0.0125 0.003125

\# calibration marks at y = -3.46945e-018 m
-0.0375 -3.46945e-018 0.003125
-0.025 -3.46945e-018 0.003125
-0.0125 -3.46945e-018 0.003125
-3.46945e-018 -3.46945e-018 0.003125
0.0125 -3.46945e-018 0.003125
0.025 -3.46945e-018 0.003125
0.0375 -3.46945e-018 0.003125

\# calibration marks at y = 0.0125 m
-0.0375 0.0125 0.003125
-0.025 0.0125 0.003125
-0.0125 0.0125 0.003125
-3.46945e-018 0.0125 0.003125
0.0125 0.0125 0.003125
0.025 0.0125 0.003125
0.0375 0.0125 0.003125

\# calibration marks at y = 0.025 m
-0.0375 0.025 0.003125
-0.025 0.025 0.003125
-0.0125 0.025 0.003125
-3.46945e-018 0.025 0.003125
0.0125 0.025 0.003125
0.025 0.025 0.003125
0.0375 0.025 0.003125

\# calibration marks at y = 0.0375 m
-0.0375 0.0375 0.003125
-0.025 0.0375 0.003125
-0.0125 0.0375 0.003125
-3.46945e-018 0.0375 0.003125
0.0125 0.0375 0.003125
0.025 0.0375 0.003125
0.0375 0.0375 0.003125
```

The file `CalPlatePSFile` contains the corresponding PostScript description of the calibration plate.

Attention

Depending on the accuracy of the used output device (e.g., laser printer), the printed calibration plate may not match the values in the calibration plate description file `CalPlateDescr` exactly. Thus, the coordinates of the calibration marks in the calibration plate description file may have to be corrected!

Parameters

▷ **XNum** (input_control) . . . integer ⇝ *integer*
Number of marks in x direction.
Default: 7
Suggested values: XNum ∈ {5, 7, 9}
Recommended increment: 1
Restriction: XNum > 1

▷ **YNum** (input_control) . . . integer ⇝ *integer*
Number of marks in y direction.
Default: 7
Suggested values: YNum ∈ {5, 7, 9}
Recommended increment: 1
Restriction: YNum > 1

▷ **MarkDist** (input_control) . . . real ⇝ *real*
Distance of the marks in meters.
Default: 0.0125
Suggested values: MarkDist ∈ {0.1, 0.0125, 0.00375, 0.00125}
Restriction: 0.0 < MarkDist

▷ **DiameterRatio** (input_control) . . . real ⇝ *real*
Ratio of the mark diameter to the mark distance.
Default: 0.5
Suggested values: DiameterRatio ∈ {0.5, 0.55, 0.6, 0.65}
Restriction: 0.0 < DiameterRatio < 1.0

▷ **CalPlateDescr** (input_control) . . . filename.write ⇝ *string*
File name of the calibration plate description.
Default: 'caltab.descr'
List of values: CalPlateDescr ∈ {'caltab.descr'}
File extension: .descr

▷ **CalPlatePSFile** (input_control) . . . filename.write ⇝ *string*
File name of the PostScript file.
Default: 'caltab.ps'
File extension: .ps

Example

```
* Create calibration plate with width = 80 cm.
gen_caltab( 7, 7, 0.1, 0.5, 'caltab.descr', 'caltab.ps')
```

Result

`gen_caltab` returns `2` (`H_MSG_TRUE`) if all parameter values are correct and both files have been written successfully. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`read_cam_par`, `caltab_points`

Alternatives

`create_caltab`

See also

find_caltab, find_marks_and_pose, camera_calibration, disp_caltab, sim_caltab

Module

Foundation

sim_caltab (: SimImage : CalPlateDescr, CameraParam, CalPlatePose, GrayBackground, GrayPlate, GrayMarks, ScaleFac :)

Simulate an image with calibration plate.

sim_caltab is used to generate a simulated calibration image. The calibration plate description is read from the file CalPlateDescr and will be projected into the image plane using the given camera parameters, thus internal camera parameters CameraParam and external camera parameters CalPlatePose (see also project_3d_point). Thereby the pose is expected to be in the form ${}^{ccs}\mathbf{P}_{wcs}$, where *ccs* denotes the camera coordinate system and *wcs* the world coordinate system (see Transformations / Poses and "Solution Guide III-C - 3D Vision").

In the simulated image only the calibration plate is shown. The image background is set to the gray value GrayBackground, the calibration plate background is set to GrayPlate, and the calibration marks are set to the gray value GrayMarks. The parameter ScaleFac influences the number of supporting points to approximate the elliptic contours of the calibration marks, see also disp_caltab. Increasing the number of supporting points causes a more accurate determination of the mark boundary, but increases the computation time, too. For each pixel of the simulated image which touches a subpixel-boundary of this kind, the gray value is set linearly between GrayMarks and GrayPlate dependent on the proportion Inside/Outside.

By applying the operator sim_caltab you can generate synthetic calibration images (with known camera parameters!) to test the quality of the calibration algorithm (see Calibration).

Parameters

▷ **SimImage** (output_object) . . . . . image ⇝ *object :* byte
Simulated calibration image.

▷ **CalPlateDescr** (input_control) . . . . . filename.read ⇝ *string*
File name of the calibration plate description.
Default: 'calplate_320mm.cpd'
List of values: CalPlateDescr ∈ {'calplate_5mm.cpd', 'calplate_10mm.cpd', 'calplate_20mm.cpd', 'calplate_40mm.cpd', 'calplate_80mm.cpd', 'calplate_160mm.cpd', 'calplate_320mm.cpd', 'calplate_640mm.cpd', 'calplate_1200mm.cpd', 'calplate_20mm_dark_on_light.cpd', 'calplate_40mm_dark_on_light.cpd', 'calplate_80mm_dark_on_light.cpd', 'caltab_650um.descr', 'caltab_2500um.descr', 'caltab_6mm.descr', 'caltab_10mm.descr', 'caltab_30mm.descr', 'caltab_100mm.descr', 'caltab_200mm.descr', 'caltab_800mm.descr', 'caltab_small.descr', 'caltab_big.descr'}
File extension: .cpd, .descr

▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **CalPlatePose** (input_control) . . . . . pose ⇝ *real* / integer
External camera parameters (3D pose of the calibration plate in camera coordinates).
Number of elements: 7

▷ **GrayBackground** (input_control) . . . . . integer ⇝ *integer*
Gray value of image background.
Default: 128
Suggested values: GrayBackground ∈ {0, 32, 64, 96, 128, 160}
Restriction: 0 <= GrayBackground <= 255

▷ **GrayPlate** (input_control) . . . . . integer ⇝ *integer*
Gray value of calibration plate.
Default: 80
Suggested values: GrayPlate ∈ {144, 160, 176, 192, 208, 224, 240}
Restriction: 0 <= GrayPlate <= 255

▷ **GrayMarks** (input_control) ..... integer ⇝ *integer*
Gray value of calibration marks.
Default: 224
Suggested values: GrayMarks ∈ {16, 32, 48, 64, 80, 96, 112}
Restriction: 0 <= GrayMarks <= 255

▷ **ScaleFac** (input_control) ..... real ⇝ *real*
Scaling factor to reduce oversampling.
Default: 1.0
Suggested values: ScaleFac ∈ {1.0, 0.5, 0.25, 0.125}
Recommended increment: 0.05
Restriction: 1.0 >= ScaleFac

Example

```
* Read calibration image.
read_image(Image1, 'calib-01')
* Find calibration pattern.
CameraType := 'area_scan_division'
StartCamPar := [CameraType, Focus, Kappa, Sx, Sy, Cx, Cy, \
               ImageWidth, ImageHeight]
create_calib_data ('calibration_object', 1, 1, CalibDataID)
set_calib_data_cam_param (CalibDataID, 0, [], StartCamPar)
set_calib_data_calib_object (CalibDataID, 0, 'calplate.cpd')
find_caltab(Image1, CalPlate1, 'caltab.descr', 3, 112, 5)
* Find calibration marks and initial pose.
find_calib_object (Image1, CalibDataID, 0, 0, 0, [], [])
* Camera calibration.
calibrate_cameras (CalibDataID, Error)
* Simulate calibration image.
get_calib_data (CalibDataID, 'calib_obj_pose', [0, 0], 'pose', FinalPose)
get_calib_data (CalibDataID, 'camera', 0, 'params', CameraParam)
sim_caltab(Image1Sim, 'calplate.cpd', CameraParam, FinalPose, 128, \
           80, 224, 1)
```

Result

sim_caltab returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

camera_calibration, find_marks_and_pose, read_pose, read_cam_par, hom_mat3d_to_pose

Possible Successors

find_caltab

See also

find_caltab, find_marks_and_pose, camera_calibration, disp_caltab, create_pose, hom_mat3d_to_pose, project_3d_point, gen_caltab

Module

Calibration

6.3 Camera Parameters

cam_mat_to_cam_par (: : CameraMatrix, Kappa, ImageWidth, ImageHeight : CameraParam)

Compute the internal camera parameters from a camera matrix.

cam_mat_to_cam_par computes internal camera parameters from the camera matrix CameraMatrix, the radial distortion coefficient Kappa, the image width ImageWidth, and the image height ImageHeight. The camera parameters are returned in CameraParam. The parameters CameraMatrix and Kappa can be determined with stationary_camera_self_calibration. cam_mat_to_cam_par converts this representation of the internal camera parameters into the representation used by camera_calibration. The conversion can only be performed if the skew of the image axes is set to 0 in stationary_camera_self_calibration, i.e., if the parameter *'skew'* is *not* being determined.

Parameters

- ▷ **CameraMatrix** (input_control) ..... hom_mat2d ⇝ *real*
  3×3 projective camera matrix that determines the internal camera parameters.
- ▷ **Kappa** (input_control) ..... number ⇝ *real*
  Kappa.
- ▷ **ImageWidth** (input_control) ..... extent.x ⇝ *integer*
  Width of the images that correspond to CameraMatrix.
  Restriction: ImageWidth > 0
- ▷ **ImageHeight** (input_control) ..... extent.y ⇝ *integer*
  Height of the images that correspond to CameraMatrix.
  Restriction: ImageHeight > 0
- ▷ **CameraParam** (output_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameters.

Example

```
* For the input data to stationary_camera_self_calibration, please
* refer to the example for stationary_camera_self_calibration.
stationary_camera_self_calibration (4, 640, 480, 1, From, To, \
                                    HomMatrices2D, Rows1, Cols1, \
                                    Rows2, Cols2, NumMatches, \
                                    'gold_standard', \
                                    ['focus','principal_point','kappa'], \
                                    'true', CameraMatrix, Kappa, \
                                    RotationMatrices, X, Y, Z, Error)
cam_mat_to_cam_par (CameraMatrix, Kappa, 640, 480, CameraParam)
```

Result

If the parameters are valid, the operator cam_mat_to_cam_par returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

stationary_camera_self_calibration

See also

camera_calibration, cam_par_to_cam_mat

Module

Calibration

cam_par_to_cam_mat (: : CameraParam : CameraMatrix, ImageWidth, ImageHeight)

Compute a camera matrix from internal camera parameters.

cam_par_to_cam_mat computes the camera matrix CameraMatrix as well as the image width ImageWidth, and the image height ImageHeight from the internal camera parameters CameraParam. The internal camera parameters CameraParam can be determined with camera_calibration. cam_par_to_cam_mat converts this representation of the internal camera parameters into the representation used by stationary_camera_self_calibration. The conversion can only be performed if the camera is an area scan pinhole camera and the distortion coefficients in CameraParam are 0. If necessary, change_radial_distortion_cam_par must be used to set the distortion coefficients to 0.

Parameters

▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **CameraMatrix** (output_control) . . . . . hom_mat2d ⇝ *real*
3×3 projective camera matrix that corresponds to CameraParam.

▷ **ImageWidth** (output_control) . . . . . extent.x ⇝ *integer*
Width of the images that correspond to CameraMatrix.
Assertion: ImageWidth > 0

▷ **ImageHeight** (output_control) . . . . . extent.y ⇝ *integer*
Height of the images that correspond to CameraMatrix.
Assertion: ImageHeight > 0

Example

```
* For the input data to calibrate_cameras, please refer to the
* example for calibrate_cameras.
calibrate_cameras (CalibDataID, Error)
get_calib_data (CalibDataID, 'camera', 0, 'params', CameraParam)
cam_par_to_cam_mat (CameraParam, CameraMatrix, ImageWidth, ImageHeight)

* Alternatively, the following calls can be used.
change_radial_distortion_cam_par ('adaptive', CameraParam, 0, CamParamOut)
cam_par_to_cam_mat (CamParamOut, CameraMatrix, ImageWidth, ImageHeight)
```

Result

If the parameters are valid, the operator cam_par_to_cam_mat returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

camera_calibration

See also

stationary_camera_self_calibration, cam_mat_to_cam_par

Module

Calibration

deserialize_cam_par (: : SerializedItemHandle : CameraParam)

Deserialize the serialized internal camera parameters.

deserialize_cam_par deserializes the internal camera parameters, that were serialized by serialize_cam_par (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized camera parameters are defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created tuple with the handle CameraParam.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **CameraParam** (output_control) ..... campar ⇝ *real* / integer / string
Internal camera parameters.

Result

If the parameters are valid, the operator deserialize_cam_par returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_cam_par

Module

Foundation

read_cam_par (: : CamParFile : CameraParam)

Read internal camera parameters from a file.

read_cam_par reads the internal camera parameters CameraParam from a file with name CamParFile. The file must have been written by write_cam_par.

The default HALCON file extension for the camera parameters is 'dat'.

The number of values in CameraParam depends on the specified camera type. See the description of set_calib_data_cam_param for a list of values and the chapter Calibration for details on camera types and camera parameters.

Parameters

▷ **CamParFile** (input_control) ..... filename.read ⇝ *string*
File name of internal camera parameters.
Default: 'campar.dat'
List of values: CamParFile ∈ {'campar.dat', 'campar.initial', 'campar.final'}
File extension: .dat

▷ **CameraParam** (output_control) ..... campar ⇝ *real* / integer / string
Internal camera parameters.

Example

```
* Create sample camera parameters and write them to file.
gen_cam_par_area_scan_division (0.01, -731, 5.2e-006, 5.2e-006, \
                                654, 519, 1280, 1024, CameraParamTmp)
write_cam_par (CameraParamTmp, 'campar_tmp.dat')
* Read internal camera parameters.
read_cam_par('campar_tmp.dat', CameraParam)
```

Result

read_cam_par returns 2 (H_MSG_TRUE) if all parameter values are correct and the file has been read successfully. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

find_marks_and_pose, sim_caltab, gen_caltab, disp_caltab, camera_calibration

See also

find_caltab, find_marks_and_pose, camera_calibration, disp_caltab, sim_caltab, write_cam_par, write_pose, read_pose, project_3d_point, get_line_of_sight

Module

Foundation

serialize_cam_par (: : CameraParam : SerializedItemHandle)

Serialize the internal camera parameters.

serialize_cam_par serializes the internal camera parameters (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_cam_par is converted to a serialized item. The camera parameters are defined by the tuple CameraParam. The serialized camera parameters are returned by the handle SerializedItemHandle and can be deserialized by deserialize_cam_par.

Parameters

▷ **CameraParam** (input_control) ..... campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_cam_par returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_cam_par

Module

Foundation

write_cam_par (: : CameraParam, CamParFile :)

Write internal camera parameters into a file.

write_cam_par stores the internal camera parameters CameraParam into a file specified by its file name CamParFile.

The number of values in CameraParam depends on the specified camera type. See the description of set_calib_data_cam_param for a list of values and the chapter Calibration for details on camera types and camera parameters.

The default HALCON file extension for the camera parameters is 'dat'.

The internal camera parameters can be later read with read_cam_par.

Parameters

▷ **CameraParam** (input_control) ..... campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **CamParFile** (input_control) ..... filename.write ⇝ *string*
File name of internal camera parameters.
Default: 'campar.dat'
List of values: CamParFile ∈ {'campar.dat', 'campar.initial', 'campar.final'}
File extension: .dat

Example

```
*
* Calibrate the camera.
*
StartCamPar := ['area_scan_division', 0.016, 0, 0.0000074, 0.0000074, \
                326, 247, 652, 494]
create_calib_data ('calibration_object', 1, 1, CalibDataID)
set_calib_data_cam_param (CalibDataID, 0, [], StartCamPar)
set_calib_data_calib_object (CalibDataID, 0, 'caltab_30mm.descr')
NumImages := 10
for I := 1 to NumImages by 1
  read_image (Image, '3d_machine_vision/calib/calib_' + I$'02d')
  find_calib_object (Image, CalibDataID, 0, 0, I, [], [])
  get_calib_data_observ_contours (Caltab, CalibDataID, 'caltab', 0, 0, I)
endfor
calibrate_cameras (CalibDataID, Error)
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
* Write the internal camera parameters to a file.
write_cam_par (CamParam, 'camera_parameters.dat')
```

Result

write_cam_par returns 2 (H_MSG_TRUE) if all parameter values are correct and the file has been written successfully. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

camera_calibration

See also

find_caltab, find_marks_and_pose, camera_calibration, disp_caltab, sim_caltab, read_cam_par, write_pose, read_pose, project_3d_point, get_line_of_sight

Module

Foundation

6.4 Hand-Eye

calibrate_hand_eye (: : CalibDataID : Errors)

Perform a hand-eye calibration.

The operator calibrate_hand_eye determines the 3D pose of a robot ("hand") relative to a camera or 3D sensor ("eye") based on the calibration data model CalibDataID. With the determined 3D poses, the poses of the calibration object in the camera coordinate system can be transformed into the coordinate system of the robot which can then, e.g., grasp an inspected part. There are two possible configurations of robot-camera (hand-eye) systems: The camera can be mounted on the robot or be stationary and observe the robot. Note that the term robot is used in place of a mechanism that moves objects. Thus, you can use calibrate_hand_eye to calibrate many different systems, from pan-tilt heads to multi-axis manipulators.

In essence, systems suitable for hand-eye calibration are described by a closed chain of four Euclidean transformations. In this chain two non-consecutive transformations are either known from the robot controller or computed from camera data, e.g., calibration object poses observed by a camera. The two unknown constant transformations are computed by the hand-eye calibration procedure.

A hand-eye calibration is performed similarly to the calibration of the external camera parameters (see Calibration): You acquire a set of poses of a calibration object in the camera coordinate system, and a corresponding set of poses of the tool in robot base coordinates and set them in the calibration data model CalibDataID.

In contrast to the camera calibration, the calibration object is not moved manually. This task is delegated to the robot. Basically, two hand-eye calibration scenarios can be distinguished. A robot either moves the camera (moving camera) or it moves the calibration object (stationary camera). The robot's movements are assumed to be known. They are used as an input for the hand-eye calibration and are set in the calibration data model CalibDataID using set_calib_data.

The results of a hand-eye calibration are two poses: For the moving camera scenario, the 3D pose of the tool in the camera coordinate system (*'tool_in_cam_pose'*) and the 3D pose of the calibration object in the robot base coordinate system (*'obj_in_base_pose'*) are computed. For the stationary camera scenario, the 3D pose of the robot base in the camera coordinate system (*'base_in_cam_pose'*) and the 3D pose of the calibration object in the tool coordinate system (*'obj_in_tool_pose'*) are computed. Their pose type is identical to the pose type of the input poses. If the input poses have different pose types, poses of type 0 are returned.

The two hand-eye calibration scenarios are discussed in more detail below, followed by general information about the data for and the preparation of the calibration data model.

Moving camera (mounted on a robot)

In this configuration, the calibration object remains stationary. The camera is mounted on the robot and is moved to different positions by the robot. The main idea behind the hand-eye calibration is that the information extracted from an observation of the calibration object, i.e., the pose of the calibration object relative to the camera, can be seen as a chain of poses or homogeneous transformation matrices from the calibration object via the base of the robot to its tool (end-effector) and finally to the camera:

Moving camera:

$$^{camera}\mathbf{H}_{cal} = {}^{camera}\mathbf{H}_{tool} \cdot \left({}^{base}\mathbf{H}_{tool}\right)^{-1} \cdot {}^{base}\mathbf{H}_{cal}$$

'obj_in_cam_pose' 'tool_in_cam_pose' 'tool_in_base_pose' 'obj_in_base_pose'

From the set of calibration object poses (*'obj_in_cam_pose'*) and the poses of the tool in the robot base coordinate system (*'tool_in_base_pose'*), the operator calibrate_hand_eye determines the two missing transformations at the ends of the chain, i.e., the pose of the robot tool in the camera coordinate system ($^{camera}\mathbf{H}_{tool}$, *'tool_in_cam_pose'*) and the pose of the calibration object in the robot base coordinate system ($^{base}\mathbf{H}_{cal}$, *'obj_in_base_pose'*). These two poses are constant.

In contrast, the transformation in the middle of the chain, $^{base}\mathbf{H}_{tool}$, is known but changes for each observation of the calibration object, because it describes the pose of the tool with respect to the robot base coordinate system. In the equation the inverted transformation matrix is used. The inversion is performed internally.

Note that when calibrating SCARA robots, it is not possible to determine the Z translation of *'obj_in_base_pose'*. To eliminate this ambiguity the Z translation *'obj_in_base_pose'* is internally set to 0.0 and the *'tool_in_cam_pose'* is calculated accordingly. It is necessary to determine the true translation in Z after the calibration by moving the robot to a pose of known height in the camera coordinate system. For this, the following approach can be applied: The calibration plate is placed at an arbitrary position. The robot is then moved such that the camera can observe the calibration plate. Now, an image of the calibration plate is acquired and the current robot pose is queried (*ToolInBasePose1*). From the image, the pose of the calibration plate in the camera coordinate system can be

determined (*ObjInCamPose1*). Afterwards, the tool of the robot is manually moved to the origin of the calibration plate and the robot pose is queried again (*ToolInBasePose2*). These three poses and the result of the calibration (*ToolInCamPose*) can be used to fix the Z ambiguity by using the following lines of code:

```
pose_invert(ToolInCamPose, CamInToolPose)
pose_compose(CamInToolPose, ObjInCamPose1, ObjInToolPose1)
pose_invert(ToolInBasePose1, BaseInToolPose1)
pose_compose(BaseInToolPose1, ToolInBasePose2, Tool2InTool1Pose)
ZCorrection := ObjInToolPose1[2]-Tool2InTool1Pose[2]
set_origin_pose(ToolInCamPose, 0, 0, ZCorrection,
ToolInCamPoseFinal)
```

The *'optimization_method' 'stochastic'* also estimates the uncertainty of observations. Besides the input poses described above, it also uses the extracted calibration marks and is thus only available for use with a camera and a calibration plate, not for use with a 3D sensor. For articulated robots, the hand-eye poses and camera parameters are refined simultaneously.

Stationary camera

In this configuration, the robot grasps the calibration object and moves it in front of the camera. Again, the information extracted from an observation of the calibration object, i.e., the pose of the calibration object in the camera coordinate system (e.g., the external camera parameters), are equal to a chain of poses or homogeneous transformation matrices, this time from the calibration object via the robot's tool to its base and finally to the camera:

Stationary camera:

$$^{camera}\mathbf{H}_{cal} = {}^{camera}\mathbf{H}_{base} \cdot {}^{base}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{cal}$$

'obj_in_cam_pose' 'base_in_cam_pose' 'tool_in_base_pose' 'obj_in_tool_pose'

Analogously to the configuration with a moving camera, the operator `calibrate_hand_eye` determines the two transformations at the ends of the chain, here the pose of the robot base coordinate system in camera coordinates ($^{camera}\mathbf{H}_{base}$, *'base_in_cam_pose'*) and the pose of the calibration object relative to the robot tool ($^{tool}\mathbf{H}_{cal}$, *'obj_in_tool_pose'*).

The transformation in the middle of the chain, $^{base}\mathbf{H}_{tool}$, describes the pose of the tool relative to the robot base coordinate system. The transformation $^{camera}\mathbf{H}_{cal}$ describes the pose of the calibration object relative to the camera coordinate system.

Note that when calibrating SCARA robots, it is not possible to determine the Z translation of *'obj_in_tool_pose'*. To eliminate this ambiguity the Z translation of *'obj_in_tool_pose'* is internally set to 0.0 and the *'base_in_cam_pose'* is calculated accordingly. It is necessary to determine the true translation in Z after the calibration by moving the robot to a pose of known height in the camera coordinate system. For this, the following approach can be applied: A calibration plate (that is not attached to the robot) is placed at an arbitrary position such that it can be observed by the camera. The pose of the calibration plate must then be determined in the camera coordinate system (*ObjInCamPose*). Afterwards the tool of the robot is manually moved to the origin of the calibration plate and the robot pose is queried (*ToolInBasePose*). The two poses and the result of the calibration (*BaseInCamPose*) can be used to fix the Z ambiguity by using the following lines of code:

```
pose_invert(BaseInCamPose, CamInBasePose)
pose_compose(CamInBasePose, ObjInCamPose, ObjInBasePose)
ZCorrection := ObjInBasePose[2]-ToolInBasePose[2]
set_origin_pose(BaseInCamPose, 0, 0, ZCorrection,
BaseInCamPoseFinal)
```

The *'optimization_method' 'stochastic'* also estimates the uncertainty of observations. Besides the input poses described above, it also uses the extracted calibration marks and is thus only available for use with a camera and a

calibration plate, not for use with a 3D sensor. For articulated robots, the hand-eye poses and camera parameters are refined simultaneously.

Preparing the calibration input data

Before calling `calibrate_hand_eye`, you must create and fill the calibration data model with the following steps:

1. **Create a calibration data model** with the operator `create_calib_data`, specifying the number of cameras in the setup and the number of used calibration objects. Depending on your scenario, `CalibSetup` has to be set to *'hand_eye_moving_camera'*, *'hand_eye_stationary_camera'*, *'hand_eye_scara_moving_camera'*, or *'hand_eye_scara_stationary_camera'*. These four scenarios on the one hand distinguish whether the camera or the calibration object is moved by the robot and on the other hand distinguish whether an articulated robot or a SCARA robot is calibrated. The arm of an articulated robot has three rotary joints typically covering 6 degrees of freedom (3 translations and 3 rotations). SCARA robots have two parallel rotary joints and one parallel prismatic joint covering only 4 degrees of freedom (3 translations and 1 rotation). Loosely speaking, an articulated robot is able to tilt its end effector while a SCARA robot is not.
2. **Specify the optimization method** with the operator `set_calib_data`. For the parameter `DataName='optimization_method'`, three options for `DataValue` are available, `DataValue='linear'`, `DataValue='nonlinear'` and `DataValue='stochastic'` (see paragraph 'Performing the actual hand-eye calibration').
3. **Specify the poses of the calibration object**
   (a) For each observation of the calibration object, the 3D pose can be set directly using the operator `set_calib_data_observ_pose`. This operator is intended to be used with generic 3D sensors that observe the calibration object.
   (b) The pose of the calibration object can also be estimated using camera images. The calibration object has to be set in the calibration data model `CalibDataID` with the operator `set_calib_data_calib_object`. Initial camera parameters have to be set with the operator `set_calib_data_cam_param`. If a standard HALCON calibration plate is used, the operator `find_calib_object` determines the pose of the calibration plate relative to the camera and saves it in the calibration data model `CalibDataID`.
   The operator `calibrate_hand_eye` for articulated (i.e., non-SCARA) robots in this case calibrates the camera before performing the hand-eye calibration. If *'optimization_method'* is set to *'stochastic'*, the hand-eye poses and camera parameters are then refined simultaneously. If the provided camera parameters are already calibrated, the camera calibration can be switched off by calling `set_calib_data (CalibDataID,'camera','general','excluded_settings','params')`.
   In contrast, for SCARA robots `calibrate_hand_eye` always assumes that the provided camera parameters are already calibrated. Therefore, in this case the internal camera calibration is never performed automatically during hand-eye calibration. This is because the internal camera parameters cannot be calibrated reliably without significantly tilting the calibration plate with respect to the camera. For hand-eye calibration, the calibration plate is often approximately parallel to the image plane. Therefore, for SCARA robots all camera poses are approximately parallel. Therefore, the camera must be calibrated beforehand by using a different set of calibration images.
4. **Specify the poses of the tool** in robot base coordinates. For each pose of the calibration object in the camera coordinate system, the corresponding pose of the tool in the robot base coordinate system has to be set with the operator `set_calib_data(CalibDataID,'tool', PoseNumber, 'tool_in_base_pose', ToolInBasePose)`.

Performing the actual hand-eye calibration

The operator `calibrate_hand_eye` can perform the calibration in three different ways. In all cases, all provided calibration object poses in camera coordinates and the corresponding poses of the tool in robot base coordinates are used for the calibration. The method *'stochastic'* also uses the extracted calibration marks, and is thus only available for use with a camera and a calibration plate, not for use with a 3D sensor. The method to be used is specified with `set_calib_data`.

For the parameter combination `DataName='optimization_method'` and `DataValue='linear'`, the calibration is performed using a linear algorithm which is fast but in many practical situations not accurate enough.

For the parameter DataName='*optimization_method*' and DataValue='*nonlinear*', the calibration is performed using a non-linear algorithm, which results in more accurately calibrated poses.

For the parameter DataName='*optimization_method*' and DataValue='*stochastic*', the calibration algorithm models the uncertainty of all measured observations including the input robot poses, which results in more robustly calibrated hand-eye poses. The estimation will be better the more input poses are used. However, the method is only available for use with a camera and a calibration plate, not for use with a 3D sensor. For articulated robots, the hand-eye poses and camera parameters are refined simultaneously.

Checking the success of the calibration

The operator calibrate_hand_eye returns the pose error of the complete chain of transformations in Errors. To be more precise, a tuple with four elements is returned, where the first element is the root-mean-square error of the translational part, the second element is the root-mean-square error of the rotational part, the third element is the maximum translational error and the fourth element is the maximum rotational error. Using these error measures, it can be determined, whether the calibration was successful.

The Errors are returned in the same units in which the input poses were given, i.e., the translational errors are typically given in meters and the rotational errors are always given in degrees.

If *'optimization_method'* is set to *'stochastic'*, get_calib_data can be used to obtain *'hand_eye_calib_error_corrected_tool'*, which differs from Errors only in that it uses the corrected robot tool poses instead of the input robot tool poses.

For articulated robots, get_calib_data can be used to obtain the *'camera_calib_error'* of the camera calibration, the root mean square error (RMSE) of the direct back projection of calibration mark centers into camera images. If *'optimization_method'* is set to *'stochastic'*, *'camera_calib_error_corrected_tool'* returns the back projection error via the pose chain using corrected tool poses.

Getting the calibration results

The poses that are computed with the operator calibrate_hand_eye can be queried with get_calib_data. For the moving camera scenario, the 3D pose of the tool in the camera coordinate system (*'tool_in_cam_pose'*) and the 3D pose of the calibration object in the robot base coordinate system (*'obj_in_base_pose'*) can be obtained. For the stationary camera scenario, the 3D pose of the robot base in the camera coordinate system (*'base_in_cam_pose'*) and the 3D pose of the calibration object in the coordinate system of the tool (*'obj_in_tool_pose'*) can be obtained.

Querying the input data

If the poses of the calibration object relative to a camera were computed with find_calib_object, then for articulated (i.e., non-SCARA) robots they are used in an internal camera calibration step preceding the hand-eye calibration and are calibrated as well. For *'optimization_method'* set to *'stochastic'*, the hand-eye poses and camera parameters are refined simultaneously, the poses of the calibration object are then updated relative to the resulting new camera parameters. The calibrated 3D poses can be queried using get_calib_data with the parameter ItemType='*calib_obj_pose*'.

If the poses of the calibration object were observed with a generic 3D sensor, they cannot be calibrated and are set by set_calib_data_observ_pose. These raw 3D poses can be queried using get_calib_data_observ_pose.

The corresponding 3D poses of the tool in the coordinate system of the robot base can be queried using get_calib_data.

Acquiring a suitable set of observations

The following conditions, especially if using a standard calibration plate, should be considered:

- The position of the calibration object (moving camera: relative to the robot's base; stationary camera: relative to the robot's tool) and the position of the camera (moving camera: relative to the robot's tool; stationary camera: relative to the robot's base) must not be changed between the calibration poses.
- Even though a lower limit of three calibration object poses is theoretically possible, it is recommended to acquire 10 or more poses, in which the pose of the camera or the robot hand are sufficiently different. If *'optimization_method'* is set to *'stochastic'*, at least 25 poses are recommended. The estimation will be better the more poses are used.

  For articulated (i.e., non-SCARA) robots the amount of rotation between the calibration object poses is essential and should be at least 30 degrees or better 60 degrees. The rotations between the poses must exhibit at least two different axes of rotation. Very different orientations lead to more precise results of the hand-eye

calibration. For SCARA robots there is only one axis of rotation. The amount of rotation between the images should also be large.

- For cameras, the internal camera parameters must be constant during and after the calibration. Note that changes of the image size, the focal length, the aperture, or the focus cause a change of the internal camera parameters.
- As mentioned, the camera must not be modified between the acquisition of the individual images. Please make sure that the focus is sufficient for the expected changes of the camera to calibration plate distance. Therefore, bright lighting conditions for the calibration plate are important, because then you can use smaller apertures, which result in a larger depth of focus.

Obtaining the poses of the robot tool

We recommend to create the robot poses in a separate program and save them in files using `write_pose`. In the calibration program you can then import them and set them in the calibration data model `CalibDataID`.

Via the Cartesian interface of the robot, you can typically obtain the pose of the tool in robot base coordinates in a notation that corresponds to the pose representations with the codes 0 or 2 (`OrderOfRotation` = *'gba'* or *'abg'*, see `create_pose`). In this case, you can directly use the pose values obtained from the robot as input for `create_pose`.

If the Cartesian interface of your robot describes the orientation in a different way, e.g., with the representation ZYZ ($\mathbf{R}_z(\varphi 1) \cdot \mathbf{R}_y(\varphi 2) \cdot \mathbf{R}_z(\varphi 3)$), you can create the corresponding homogeneous transformation matrix step by step using the operators `hom_mat3d_rotate` and `hom_mat3d_translate` and then convert the resulting matrix into a pose using `hom_mat3d_to_pose`. The following example code creates a pose from the ZYZ representation described above:

```
hom_mat3d_identity(HomMat3DIdent)
hom_mat3d_rotate(HomMat3DIdent, phi3, 'z', 0, 0, 0, HomMat3DRotZ)
hom_mat3d_rotate(HomMat3DRotZ, phi2, 'y', 0, 0, 0, HomMat3DRotZY)
hom_mat3d_rotate(HomMat3DRotZY, phi1, 'z', 0, 0, 0,
HomMat3DRotZYZ)
hom_mat3d_translate(HomMat3DRotZYZ, Tx, Ty, Tz, base_H_tool)
hom_mat3d_to_pose(base_H_tool, RobPose)
```

Please note that the hand-eye calibration only works if the poses of the tool in robot base coordinates are specified with high accuracy. Of the provided methods, *'optimization_method'* set to *'stochastic'* will yield the most robust results with respect to noise on the poses of the tool in robot base coordinates. The estimation will be better the more input poses are used.

Please note that this operator supports canceling timeouts and interrupts if *'optimization_method'* is set to *'stochastic'*.

Parameters

▷ **CalibDataID** (input_control) ..... calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **Errors** (output_control) ..... number-array ⇝ *real*
Average residual error of the optimization.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_calib_data, set_calib_data_cam_param, set_calib_data_calib_object, set_calib_data_observ_pose, find_calib_object, set_calib_data, remove_calib_data, remove_calib_data_observ

Possible Successors

get_calib_data

References

K. Daniilidis: "Hand-Eye Calibration Using Dual Quaternions"; International Journal of Robotics Research, Vol. 18, No. 3, pp. 286-298; 1999.

M. Ulrich, C. Steger: "Hand-Eye Calibration of SCARA Robots Using Dual Quaternions"; Pattern Recognition and Image Analysis, Vol. 26, No. 1, pp. 231-239; January 2016.

M. Ulrich, M. Hillemann: "Generic Hand–Eye Calibration of Uncertain Robots"; 2021 IEEE International Conference on Robotics and Automation (ICRA), pp. 11060-11066; 2021.

Module

Calibration

get_calib_data_observ_pose (: : CalibDataID, CameraIdx, CalibObjIdx, CalibObjPoseIdx : ObjInCameraPose)

Get observed calibration object poses from a calibration data model.

The operator get_calib_data_observ_pose reads the poses of the calibration object given in the camera coordinate system from a calibration data model CalibDataID. The observation data was previously stored by set_calib_data_observ_pose, find_calib_object, or set_calib_data_observ_points.

Note that if the model CalibDataID uses a general sensor and no calibration object (i.e., the model was created by create_calib_data with NumCameras=0 and NumCalibObjects=0), then both CameraIdx and CalibObjIdx must be set to 0.

Parameters

- ▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
  Handle of a calibration data model.
- ▷ **CameraIdx** (input_control) . . . . . number ⇝ *integer*
  Index of the observing camera.
  Default: 0
- ▷ **CalibObjIdx** (input_control) . . . . . number ⇝ *integer*
  Index of the observed calibration object.
  Default: 0
- ▷ **CalibObjPoseIdx** (input_control) . . . . . number ⇝ *integer*
  Index of the observed calibration object pose.
  Default: 0
- ▷ **ObjInCameraPose** (output_control) . . . . . pose ⇝ *real* / integer
  Stored observed calibration object pose relative to the observing camera.
  Number of elements: 7

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Calibration

hand_eye_calibration (: : X, Y, Z, Row, Col, NumPoints, RobotPoses, CameraParam, Method, QualityType : CameraPose, CalibrationPose, Quality)

Perform a hand-eye calibration.

The operator hand_eye_calibration determines the 3D pose of a robot ("hand") relative to a camera ("eye"). With this information, the results of image processing can be transformed into the coordinate system of the robot which can then, e.g., grasp an inspected part. Please note that the operator hand_eye_calibration does not support 3D sensors. A hand-eye calibration including 3D sensors is only supported by the operator calibrate_hand_eye. That operator furthermore provides a more user-friendly interface to the hand-eye calibration than the operator hand_eye_calibration, since the reference coordinate systems are explicitly indicated.

There are two possible configurations of robot-camera (hand-eye) systems: The camera can be mounted on the robot or be stationary and observe the robot. Note that the term robot is used in place for a mechanism that moves objects. Thus, you can use hand_eye_calibration to calibrate many different systems, from pan-tilt heads to multi-axis manipulators.

In essence, systems suitable for hand-eye calibration are described by a closed chain of four Euclidean transformations. In this chain two non-consecutive transformations are either known from the robot controller or computed from calibration points seen by a camera system. The two other constant transformations are computed by the hand-eye calibration procedure.

A hand-eye calibration is performed similarly to the calibration of the external camera parameters (see camera_calibration): You acquire a set of images of a calibration object, determine correspondences between known calibration points and their projection in the images and pass them to hand_eye_calibration via the parameters X, Y, Z, Row, Col, and NumPoints. If you use the standard calibration plate, the correspondences can be determined very easily with the operators find_caltab and find_marks_and_pose. Furthermore, the camera is identical for the complete calibration sequence and is specified by the internal camera parameters in CameraParam. The internal camera parameters are calibrated beforehand deploying the operator calibrate_cameras or camera_calibration.

In contrast to the camera calibration, the calibration object is not moved manually. This task is delegated to the robot, which either moves the camera (mounted camera) or the calibration object (stationary camera). The robot's movements are assumed to be known and therefore are also used as an input for the calibration (parameter RobotPoses).

The output of hand_eye_calibration are the two poses CameraPose and CalibrationPose. Their pose type is identical to the pose type of the first input robot pose.

Basically, two hand-eye configurations can be distinguished and are discussed in more detail below, followed by general information about the process of hand-eye calibration.

Moving camera (mounted on a robot)

In this configuration, the calibration object remains stationary and the camera is moved to different positions by the robot. The main idea behind the hand-eye calibration is that the information extracted from a calibration image, i.e., the pose of the calibration object relative to the camera (i.e., the external camera parameters), can be seen as a chain of poses or homogeneous transformation matrices, from the calibration object via the base of the robot to its tool (end-effector) and finally to the camera:

Moving camera:

$$ {}^{cam}\mathbf{H}_{cal} = {}^{cam}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{base} \cdot {}^{base}\mathbf{H}_{cal} $$

CameraPose RobotPoses CalibrationPose

From the set of calibration images, the operator hand_eye_calibration determines the two transformations at the ends of the chain, i.e., the pose of the robot tool in camera coordinates (${}^{cam}\mathbf{H}_{tool}$,CameraPose) and the pose of the calibration object in the robot base coordinate system (${}^{base}\mathbf{H}_{cal}$,CalibrationPose).

In contrast, the transformation in the middle of the chain, ${}^{tool}\mathbf{H}_{base}$, is known but changes for each calibration image, because it describes the pose of the robot moving the camera, or to be more exact its inverse pose (pose of

the base coordinate system in robot tool coordinates). You must specify the inverse robot poses in the calibration images in the parameter RobotPoses.

Note that when calibrating SCARA robots it is not possible to determine the Z translation of CalibrationPose. To eliminate this ambiguity the Z translation of CalibrationPose is internally set to 0.0 and the CameraPose is calculated accordingly. It is necessary to determine the true translation in Z after the calibration (see calibrate_hand_eye).

Stationary camera

In this configuration, the robot grasps the calibration object and moves it in front of the camera. Again, the information extracted from a calibration image, i.e., the pose of the calibration object in camera coordinates (the external camera parameters), are equal to a chain of poses or homogeneous transformation matrices, this time from the calibration object via the robot's tool to its base and finally to the camera:

Stationary camera:

$$^{cam}\mathbf{H}_{cal} = {}^{cam}\mathbf{H}_{base} \cdot {}^{base}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{cal}$$

CameraPose RobotPoses CalibrationPose

Analogously to the configuration with a moving camera, the operator hand_eye_calibration determines the two transformations at the ends of the chain, here the pose of the robot base coordinate system in camera coordinates ($^{cam}\mathbf{H}_{base}$,CameraPose) and the pose of the calibration object relative to the robot tool ($^{tool}\mathbf{H}_{cal}$,CalibrationPose).

The transformation in the middle of the chain, $^{base}\mathbf{H}_{tool}$, describes the pose of the robot moving the calibration object, i.e., the pose of the tool relative to the base coordinate system. You must specify the robot poses in the calibration images in the parameter RobotPoses.

Note that when calibrating SCARA robots it is not possible to determine the Z translation of CalibrationPose. To eliminate this ambiguity the Z translation of CalibrationPose is internally set to 0.0 and the CameraPose is calculated accordingly. It is necessary to determine the true translation in Z after the calibration (see calibrate_hand_eye).

Additional information about the calibration process

The following sections discuss individual questions arising from the use of hand_eye_calibration. They are intended to be a guideline for using the operator in an application, as well as to help understanding the operator.

How do I get 3D calibration points and their projections? 3D calibration points given in the world coordinate system (X, Y, Z) and their associated projections in the image (Row, Col) form the basis of the hand-eye calibration. In order to be able to perform a successful hand-eye calibration, you need at least three images of the 3D calibration points that were obtained under different poses of the manipulator. In each image at least four points must be available, in order to compute internally the pose transferring the calibration points from their world coordinate system into the camera coordinate system.

In principle, you can use arbitrary known points for the calibration. However, it is usually most convenient to use the standard calibration plate, e.g., the one that can be generated with gen_caltab. In this case, you can use the operators find_caltab and find_marks_and_pose to extract the position of the calibration plate and of the calibration marks and the operator caltab_points to read the 3D coordinates of the calibration marks (see also the description of camera_calibration).

The parameter NumPoints specifies the number of 3D calibration points used for each pose of the manipulator, i.e., for each image. With this, the 3D calibration points which are stored in a linearized fashion in X, Y, Z, and their corresponding projections (Row, Col) can be associated with the corresponding pose of the manipulator (RobotPoses). Note that in contrast to the operator camera_calibration the 3D coordinates of the calibration points must be specified for each calibration image, not only once, and thus can vary for each image of the sequence.

How do I acquire a suitable set of images? The following conditions, especially if using a standard calibration plate, should be considered:

- The position of the calibration marks (moving camera: relative to the robot's base; stationary camera: relative to the robot's tool) and the position of the camera (moving camera: relative to the robot's tool; stationary camera: relative to the robot's base) must not be changed between the images.

- The internal camera parameters (CameraParam) must be constant and must be determined in a previous camera calibration step (see camera_calibration). Note that changes of the image size, the focal length, the aperture, or the focus cause a change of the internal camera parameters.
- The theoretical lower limit of the number of image to acquire is three. Nevertheless, it is recommended to have 10 or more images at hand, in which the position of the camera or the robot hand are sufficiently different.
  For articulated (i.e., non-SCARA) robots the amount of rotation between the images is essential and should be at least 30 degrees or better 60 degrees. The rotations between the images must exhibit at least two different axes of rotation. Very different orientations lead to precise calibration results. For SCARA robots there is only one axis of rotation. The amount of rotation between the images should also be large.
- In each image, the calibration plate must be completely visible (including its border).
- Reflections or other disturbances should not impair the detection of the calibration plate and its calibration marks.
- If individual calibration marks instead of the standard calibration plate are used at least four marks must be present in each image.
- In each image, the calibration plate should at least fill one quarter of the entire image for a precise computation of the calibration to camera transformation, which is performed internally during hand-eye calibration.
- As mentioned, the camera must not be modified between the acquisition of the individual images. Please make sure that the focus is sufficient for the expected changes of the camera to calibration plate distance. Therefore, bright lighting conditions for the calibration plate are important, because then you can use smaller apertures, which result in a larger depth of focus.

How do I obtain the poses of the robot? In the parameter RobotPoses you must pass the poses of the robot in the calibration images (moving camera: pose of the robot base in robot tool coordinates; stationary camera: pose of the robot tool in robot base coordinates) in a linearized fashion. We recommend to create the robot poses in a separate program and save in files using write_pose. In the calibration program you can then read and accumulate them in a tuple as shown in the example program below. In addition, we recommend to save the pose of the robot tool in robot base coordinates independent of the hand-eye configuration. When using a moving camera, you then invert the read poses before accumulating them. This is also shown in the example program.

Via the Cartesian interface of the robot, you can typically obtain the pose of the tool in base coordinates in a notation that corresponds to the pose representations with the codes 0 or 2 (OrderOfRotation = *'gba'* or *'abg'*, see create_pose). In this case, you can directly use the pose values obtained from the robot as input for create_pose.

If the Cartesian interface of your robot describes the orientation in a different way, e.g., with the representation ZYZ ($\mathbf{R}_z(\varphi 1) \cdot \mathbf{R}_y(\varphi 2) \cdot \mathbf{R}_z(\varphi 3)$), you can create the corresponding homogeneous transformation matrix step by step using the operators hom_mat3d_rotate and hom_mat3d_translate and then convert the matrix into a pose using hom_mat3d_to_pose. The following example code creates a pose from the ZYZ representation described above:

```
hom_mat3d_identity(HomMat3DIdent)
hom_mat3d_rotate(HomMat3DIdent, phi3, 'z', 0, 0, 0,
HomMat3DRotZ)
hom_mat3d_rotate(HomMat3DRotZ, phi2, 'y', 0, 0, 0,
HomMat3DRotZY)
hom_mat3d_rotate(HomMat3DRotZY, phi1, 'z', 0, 0, 0,
HomMat3DRotZYZ)
hom_mat3d_translate(HomMat3DRotZYZ, Tx, Ty, Tz, base_H_tool)
hom_mat3d_to_pose(base_H_tool, RobPose)
```

Please note that the hand-eye calibration only works if the robot poses RobotPoses are specified with high accuracy!

What is the order of the individual parameters? The length of the tuple NumPoints corresponds to the number of different positions of the manipulator and thus to the number of calibration images. The parameter NumPoints determines the number of calibration points used in the individual positions. If the standard calibration plate is used, this means 49 points per position (image). If, for example, 15 images were acquired, NumPoints is a tuple of length 15, where all elements of the tuple have the value 49.

The number of images in the sequence, which is determined by the length of NumPoints, must also be taken into account for the tuples of the 3D calibration points and the extracted 2D marks, respectively. Hence, for 15 calibration images with 49 calibration points each, the tuples X, Y, Z, Row, and Col must contain $15 \cdot 49 = 735$ values each. These tuples are ordered according to the image the respective points lie in, i.e., the first 49 values correspond to the 49 calibration points in the first image. The order of the 3D calibration points and the extracted 2D calibration points must be the same in each image.

The length of the tuple RobotPoses also depends on the number of calibration images. If, for example, 15 images and therefore 15 poses are used, the length of the tuple RobotPoses is $15 \cdot 7 = 105$ (15 times 7 pose parameters). The first seven parameters thus determine the pose of the manipulator in the first image, and so on.

Algorithm and output parameters The parameter Method determines the type of algorithm used for the hand-eye calibration: With *'linear'* a linear algorithm is chosen, which is fast but in many practical situations not accurate enough. *'nonlinear'* selects a non-linear algorithm, which results in the most accurately calibrated poses and which is the method of choice.

For the calibration of SCARA robots the parameter Method must be set to *'scara_linear'* or *'scara_nonlinear'*, respectively. While the arm of an articulated robot has three rotary joints typically covering 6 degrees of freedom (3 translations and 3 rotations), SCARA robots have two parallel rotary joints and one parallel prismatic joint covering only 4 degrees of freedom (3 translations and 1 rotation). Loosely speaking, an articulated robot is able to tilt its end effector while a SCARA robot is not.

The parameter QualityType switches between different possibilities for assessing the quality of the calibration result returned in Quality. *'error_pose'* stands for the pose error of the complete chain of transformations. To be more precise, a tuple with four elements is returned, where the first element is the root-mean-square error of the translational part, the second element is the root-mean-square error of the rotational part, the third element is the maximum translational error and the fourth element is the maximum rotational error. With *'standard_deviation'*, a tuple with 12 elements containing the standard deviations of the two poses is returned: The first six elements refer to the camera pose and the others to the pose of the calibration points. With *'covariance'*, the full 12x12 covariance matrix of both poses is returned. Like poses, the standard deviations and the covariances are specified in the units [m] and [°]. Note that selecting *'linear'* or *'scara_linear'* for the parameter Method enables only the output of the pose error (*'error_pose'*).

Parameters

▷ **X** (input_control) . . . . . number-array ⇝ *real* / integer
Linear list containing all the x coordinates of the calibration points (in the order of the images).

▷ **Y** (input_control) . . . . . number-array ⇝ *real* / integer
Linear list containing all the y coordinates of the calibration points (in the order of the images).

▷ **Z** (input_control) . . . . . number-array ⇝ *real* / integer
Linear list containing all the z coordinates of the calibration points (in the order of the images).

▷ **Row** (input_control) . . . . . number-array ⇝ *real* / integer
Linear list containing all row coordinates of the calibration points (in the order of the images).

▷ **Col** (input_control) . . . . . number-array ⇝ *real* / integer
Linear list containing all the column coordinates of the calibration points (in the order of the images).

▷ **NumPoints** (input_control) . . . . . integer-array ⇝ *integer*
Number of the calibration points for each image.

▷ **RobotPoses** (input_control) . . . . . pose-array ⇝ *real* / integer
Known 3D pose of the robot for each image (moving camera: robot base in robot tool coordinates; stationary camera: robot tool in robot base coordinates).

▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **Method** (input_control) . . . . . string ⇝ *string*
Method of hand-eye calibration.
Default: 'nonlinear'
List of values: Method ∈ {'linear', 'nonlinear', 'scara_linear', 'scara_nonlinear'}

▷ **QualityType** (input_control) . . . . . string(-array) ⇝ *string*
Type of quality assessment.
Default: 'error_pose'
List of values: QualityType ∈ {'error_pose', 'standard_deviation', 'covariance'}

▷ **CameraPose** (output_control) . . . . . pose ⇝ *real* / integer
Computed relative camera pose: 3D pose of the robot tool (moving camera) or robot base (stationary camera), respectively, in camera coordinates.

▷ **CalibrationPose** (output_control) . . . . . pose ⇝ *real* / integer
Computed 3D pose of the calibration points in robot base coordinates (moving camera) or in robot tool coordinates (stationary camera), respectively.

▷ **Quality** (output_control) . . . . . real(-array) ⇝ *real*
Quality assessment of the result.

Example

```
* Note that, in order to use this code snippet, you must provide
* the camera parameters, the calibration plate description file,
* the calibration images, and the robot poses.
read_cam_par('campar.dat', CameraParam)
CalDescr := 'caltab.descr'
caltab_points(CalDescr, X, Y, Z)
* Process all calibration images.
for i := 0 to NumImages-1 by 1
  read_image(Image, 'calib_'+i$'02d')
  * Find marks on the calibration plate in every image.
  find_caltab(Image, CalPlate, CalDescr, 3, 150, 5)
  find_marks_and_pose(Image, CalPlate, CalDescr, CameraParam, 128, 10, 18, \
                      0.9, 15, 100, RCoordTmp, CCoordTmp, StartPose)
  * Accumulate 2D and 3D coordinates of the marks.
  RCoord := [RCoord, RCoordTmp]
  CCoord := [CCoord, CCoordTmp]
  XCoord := [XCoord, X]
  YCoord := [YCoord, Y]
  ZCoord := [ZCoord, Z]
  NumMarker := [NumMarker, |RCoordTmp|]
  * Read pose of the robot tool in robot base coordinates.
  read_pose('robpose_'+i$'02d'+'.dat', RobPose)
  * Moving camera? Invert pose.
  if (IsMovingCameraConfig == 'true')
    pose_to_hom_mat3d(RobPose, base_H_tool)
    hom_mat3d_invert(base_H_tool, tool_H_base)
    hom_mat3d_to_pose(tool_H_base, RobPose)
  endif
  * Accumulate robot poses.
  RobotPoses := [RobotPoses, RobPose]
endfor
*
* Perform hand-eye calibration.
*
hand_eye_calibration(XCoord, YCoord, ZCoord, RCoord, CCoord, NumMarker, \
                     RobotPoses, CameraParam, 'nonlinear', 'error_pose', \
                     CameraPose, CalibrationPose, Error)
```

Result

The operator `hand_eye_calibration` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

find_marks_and_pose, camera_calibration, calibrate_cameras

Possible Successors

write_pose, convert_pose_type, pose_to_hom_mat3d, disp_caltab, sim_caltab

Alternatives

calibrate_hand_eye

See also

find_caltab, find_marks_and_pose, disp_caltab, sim_caltab, write_cam_par, read_cam_par, create_pose, convert_pose_type, write_pose, read_pose, pose_to_hom_mat3d, hom_mat3d_to_pose, caltab_points, gen_caltab, calibrate_hand_eye

References

K. Daniilidis: "Hand-Eye Calibration Using Dual Quaternions"; International Journal of Robotics Research, Vol. 18, No. 3, pp. 286-298; 1999.

M. Ulrich, C. Steger: "Hand-Eye Calibration of SCARA Robots Using Dual Quaternions"; Pattern Recognition and Image Analysis, Vol. 26, No. 1, pp. 231-239; January 2016.

Module

Calibration

set_calib_data_observ_pose (: : CalibDataID, CameraIdx, CalibObjIdx, CalibObjPoseIdx, ObjInCameraPose :)

Set observed calibration object poses in a calibration data model.

For a calibration data model of type CalibSetup='*hand_eye_moving_cam*', '*hand_eye_stationary_cam*', '*hand_eye_scara_moving_cam*', or '*hand_eye_scara_stationary_cam*' with no calibration object (see create_calib_data), the hand-eye calibration is based on so-called observations of an arbitrary object in the camera coordinate system. In the following this object will be called calibration object. Additionally, the corresponding poses of the robot tool in the robot base coordinate system must be known. With set_calib_data_observ_pose, you store an observation of the calibration object pose in the calibration data model CalibDataID. An observation of the calibration object pose consists of the following data:

CameraIdx: Index of the observing camera

CalibObjIdx: Index of the observed calibration object

CalibObjPoseIdx: Index of the observed pose of the calibration object. You can choose it freely, without following a strict order. If you specify an index that already exists for the calibration object CalibObjIdx, the corresponding observation data is replaced by the new one.

ObjInCameraPose: Pose of the observed calibration object relative to observing camera.

Note that, since the model CalibDataID uses a general sensor and no calibration object (i.e., the model was created by create_calib_data with NumCameras=0 and NumCalibObjects=0), both CameraIdx and CalibObjIdx must be set to 0. If the model uses a camera and an calibration object (i.e., NumCameras=1 and NumCalibObjects=1), then find_calib_object or set_calib_data_observ_points must be used.

The observation pose data can be accessed later by calling get_calib_data_observ_pose using the same values for the arguments CameraIdx, CalibObjIdx, and CalibObjPoseIdx.

Parameters

▷ **CalibDataID** (input_control) ..... calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **CameraIdx** (input_control) ..... number ⇝ *integer*
Index of the observing camera.
Default: 0
Suggested values: CameraIdx ∈ {0, 1, 2}

▷ **CalibObjIdx** (input_control) . . . . . number ⇝ *integer*
Index of the calibration object.
Default: 0
Suggested values: CalibObjIdx ∈ {0, 1, 2}

▷ **CalibObjPoseIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observed calibration object.
Default: 0
Suggested values: CalibObjPoseIdx ∈ {0, 1, 2}
Restriction: CalibObjPoseIdx >= 0

▷ **ObjInCameraPose** (input_control) . . . . . pose ⇝ *real* / integer
Pose of the observed calibration object relative to the observing camera.
Number of elements: 7

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

find_marks_and_pose, set_calib_data_cam_param, set_calib_data_calib_object

Possible Successors

set_calib_data, calibrate_cameras

Alternatives

find_calib_object

Module

Calibration

6.5 Inverse Projection

get_line_of_sight (: : Row, Column, CameraParam : PX, PY, PZ, QX, QY, QZ)

Compute the line of sight corresponding to a point in the image.

get_line_of_sight computes the line of sight corresponding to a pixel (Row, Column) in the image. The line of sight is a (straight) line in the camera coordinate system, which is described by two points (PX,PY,PZ) and (QX,QY,QZ) on the line. The camera is described by the internal camera parameters CameraParam (see Calibration for details). If a pinhole camera is used, the second point lies on the focal plane, i.e., for frame cameras, the output parameter QZ is equivalent to the focal length of the camera, whereas for linescan cameras, QZ also depends on the motion of the camera with respect to the object. The equation of the line of sight is given by

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} \mathtt{PX} \\ \mathtt{PY} \\ \mathtt{PZ} \end{pmatrix} + l \cdot \begin{pmatrix} \mathtt{QX} - \mathtt{PX} \\ \mathtt{QY} - \mathtt{PY} \\ \mathtt{QZ} - \mathtt{PZ} \end{pmatrix}$$

The advantage of representing the line of sight as two points is that it is easier to transform the line in 3D. To do so, all that is necessary is to apply the operator affine_trans_point_3d to the two points.

Parameters

▷ **Row** (input_control) . . . . . real-array ⇝ *real*
Row coordinate of the pixel.

▷ **Column** (input_control) . . . . . real-array ⇝ *real*
Column coordinate of the pixel.

▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **PX** (output_control) . . . . . real-array ⇝ *real*
X coordinate of the first point on the line of sight in the camera coordinate system

▷ **PY** (output_control) . . . . . real-array ⇝ *real*
Y coordinate of the first point on the line of sight in the camera coordinate system

▷ **PZ** (output_control) . . . . . real-array ⇝ *real*
Z coordinate of the first point on the line of sight in the camera coordinate system

▷ **QX** (output_control) . . . . . real-array ⇝ *real*
X coordinate of the second point on the line of sight in the camera coordinate system

▷ **QY** (output_control) . . . . . real-array ⇝ *real*
Y coordinate of the second point on the line of sight in the camera coordinate system

▷ **QZ** (output_control) . . . . . real-array ⇝ *real*
Z coordinate of the second point on the line of sight in the camera coordinate system

Example

```
* Set internal camera parameters.
* Note the, typically, these values are the result of a prior
* calibration.
gen_cam_par_area_scan_division (0.01, 30, 4.65e-006, 4.65e-006, \
                                640, 480, 1280, 960, CameraParam)
* Inverse projection.
get_line_of_sight([50, 100], [100, 200], CameraParam, PX, PY, PZ, QX, QY, QZ)
```

Result

`get_line_of_sight` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`read_cam_par`, `camera_calibration`

Possible Successors

`affine_trans_point_3d`

See also

`camera_calibration`, `disp_caltab`, `read_cam_par`, `project_3d_point`, `affine_trans_point_3d`

Module

Calibration

6.6 Monocular

camera_calibration (: : NX, NY, NZ, NRow, NCol, StartCamParam, NStartPose, EstimateParams : CameraParam, NFinalPose, Errors)

Determine all camera parameters by a simultaneous minimization process.

camera_calibration performs the calibration of a single camera. For this, known 3D model points (with coordinates NX, NY, NZ) are projected into the image and the sum of the squared distances between the projected 3D-coordinates and their corresponding image point coordinates (NRow, NCol) is minimized.

As initial values for the minimization process the external (NStartPose) and internal (StartCamParam) camera parameters are used. Thereby NStartPose is an ordered tuple with all initial values for the external camera parameters given in the form ${}^{ccs}\mathbf{P}_{wcs}$, where *ccs* denotes the camera coordinate system and *wcs* the world coordinate system (see Transformations / Poses and "Solution Guide III-C - 3D Vision"). Individual camera parameters can be explicitly included or excluded from the minimization with EstimateParams. For a detailed description of the available camera models, the different sets of internal camera parameters, and general requirements for the setup, see Calibration.

For a successful calibration, at least one calibration object with accurately known metric properties is needed, e.g., a HALCON calibration plate. Before calling camera_calibration, take a series of images of the calibration object in different orientations and make sure that the whole field of view or measurement volume is covered. The success of the calibration highly depends on the quality of the calibration object and the images. So you might want to exercise special diligence during the acquisition of the calibration images. See the section "How to take a set of suitable images?" in Calibration for further details.

After a successful calibration, camera_calibration returns the optimized internal (CameraParam) and external (NFinalPose ${}^{ccs}\mathbf{P}_{wcs}$) camera parameters of the camera. Additionally, the root mean square error (RMSE) of the back projection of the optimization is returned in Errors (in pixels). This error gives a general indication whether the optimization was successful.

Preparation of the calibration process

How to extract the calibration marks in the images? If a HALCON calibration plate is used, you can use the operator find_calib_object to determine the coordinates of the calibration marks in each image and to compute a rough estimate for the external camera parameters. Using HALCON calibration plates with rectangularly arranged marks (see gen_caltab), a combination of the two operators find_caltab and find_marks_and_pose will have the same effect. In both cases, the hereby obtained values can directly be used as initial values for the external camera parameters (NStartPose).

Obviously, images in which the segmentation of the calibration plate (find_caltab) has failed or the calibration marks have not been determined successfully by find_marks_and_pose or find_calib_object should not be used.

How do you get the required initial values for the calibration? If you use a HALCON calibration plate, the input parameters NX, NY, and NZ are stored in the description file of the calibration plate. You can easily access them by calling the operator caltab_points. Initial values for the internal camera parameters (StartCamParam) can be obtained from the specifications of the used camera. Further information can be found in Calibration. Initial values for the poses of the calibration plate and the coordinates of the calibration marks NRow and NCol can be calculated using the operator find_calib_object. The tuple NStartPose is set by the concatenation of all these poses.

Which camera parameters are estimated? The input parameter EstimateParams is used to select which camera parameters to estimate. Usually, this parameter is set to *'all'*, i.e., all 6 external camera parameters (translation and rotation) and all internal camera parameters are determined. If the internal camera parameters already have been determined (e.g., by a previous call to camera_calibration), it is often desired to only determine the pose of the world coordinate system in camera coordinates (i.e., the external camera parameters). In this case, EstimateParams can be set to *'pose'*. This has the same effect as EstimateParams = *['alpha','beta','gamma','transx','transy','transz']*. Otherwise, EstimateParams contains a tuple of strings that indicates the combination of parameters to estimate. In addition, parameters can be excluded from estimation by using the prefix ~. For example, the values *['pose','~transx']* have the same effect as *['alpha','beta','gamma','transy','transz']*. As a different example, *['all','~focus']* determines all internal and external parameters except the focus. The prefix ~ can be used with all parameter values except *'all'*.

Which limitations exist for the determination of the camera parameters? For additional information about general limitations when determining camera parameters, please see the section "Further Limitations Related to Specific Camera Types" in the chapter Calibration.

What is the order within the individual parameters? The length of the tuple NStartPose depends on the number of calibration images, e.g., using 15 images leads to a length of the tuple NStartPose equal to $15 \cdot 7 = 105$ (15 times the 7 external camera parameters). The first 7 values correspond to the pose of the calibration plate in the first image, the next 7 values to the pose in the second image, etc.

This fixed number of calibration images must be considered within the tuples with the coordinates of the 3D model marks and the extracted 2D marks. If 15 images are used, the length of the tuples NRow and NCol is 15 times the length of the tuples with the coordinates of the 3D model marks (NX, NY, and NZ). If every image consists 49 marks, the length of the tuples NRow and NCol is $15 \cdot 49 = 735$, while the length of the tuples NX, NY, and NZ is 49. The order of the values in NRow and NCol is "image after image", i.e., using 49 marks the first 3D model point corresponds to the 1st, 50th, 99th, 148th, 197th, 246th, etc. extracted 2D mark.

What is the meaning of the output parameters? If the camera calibration process has finished successfully, the output parameters CameraParam and NFinalPose contain the adjusted values for the internal and external camera parameters. The length of the tuple NFinalPose corresponds to the length of the tuple NStartPose.

The representation types of NFinalPose correspond to the representation type of the first tuple of NStartPose (see create_pose). You can convert the representation type by convert_pose_type.

As an additional parameter, the root mean square error (RMSE) (Errors) of the back projection of the optimization is returned. This parameter reflects the accuracy of the calibration. The error value (root mean square error of the position) is measured in pixels. If only a single camera is calibrated, an Error in the order of 0.1 pixel (the typical detection error by extraction of the coordinates of the projected calibration markers) is an indication that the optimization fits the observation data well. If Errors strongly differs from 0.1 pixels, the calibration did not perform well. Reasons for this might be, e.g., a poor image quality, an insufficient number of calibration images, or an inaccurate calibration plate.

Do I have to use a planar calibration object? No. The operator camera_calibration is designed in a way that the input tuples NX, NY, NZ, NRow, and NCol can contain any 3D/2D correspondences. The order of the single parameters is explained in the paragraph "What is the order within the individual parameters?".

Thus, it makes no difference how the required 3D model marks and the corresponding 2D marks are determined. On the one hand, it is possible to use a 3D calibration object, on the other hand, you also can use any characteristic points (e.g., natural landmarks) with known position in the world. By setting EstimateParams to *'pose'*, it is thus possible to compute the pose of an object in camera coordinates! For this, at least three 3D/2D-correspondences are necessary as input. NStartPose can, e.g., be generated directly as shown in the program example for create_pose.

Attention

The minimization process of the calibration depends on the initial values of the internal (StartCamParam) and external (NStartPose) camera parameters. The computed average errors Errors give an impression of the accuracy of the calibration. The errors (deviations in x- and y-coordinates) are measured in pixels.

For line scan cameras, it is possible to set the start value for the internal camera parameter Sy to the value 0.0. In this case, it is not possible to determine the position of the principal point in y-direction. Therefore, EstimateParams must contain the term *'~cy'*. The effective distance of the principle point from the sensor line is then always $p_v = S_y \cdot C_y = 0.0$. Further information can be found in the section "Further Limitations Related to Specific Camera Types" of Calibration.

Parameters

▷ **NX** (input_control) . . . . . number-array ⇝ *real* / integer
Ordered tuple with all x coordinates of the calibration marks (in meters).

▷ **NY** (input_control) . . . . . number-array ⇝ *real* / integer
Ordered tuple with all y coordinates of the calibration marks (in meters).
Number of elements: NY == NX

▷ **NZ** (input_control) . . . . . number-array ⇝ *real* / integer
Ordered tuple with all z coordinates of the calibration marks (in meters).
Number of elements: NZ == NX

▷ **NRow** (input_control) . . . . . number-array ⇝ *real* / integer
Ordered tuple with all row coordinates of the extracted calibration marks (in pixels).

▷ **NCol** (input_control) . . . . . number-array ⇝ *real* / integer
Ordered tuple with all column coordinates of the extracted calibration marks (in pixels).
Number of elements: NCol == NRow

▷ **StartCamParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Initial values for the internal camera parameters.

▷ **NStartPose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
Ordered tuple with all initial values for the external camera parameters.
Number of elements: NStartPose == 7 * NRow / NX

▷ **EstimateParams** (input_control) . . . . . string-array ⇝ *string*
Camera parameters to be estimated.
Default: 'all'
List of values: `EstimateParams` ∈ {'all', 'pose', 'camera', 'alpha', 'beta', 'gamma', 'transx', 'transy', 'transz', 'focus', 'magnification', 'kappa', 'poly', 'k1', 'k2', 'k3', 'poly_tan_2', 'image_plane_dist', 'tilt', 'cx', 'cy', 'sx', 'sy', 'vx', 'vy', 'vz'}

▷ **CameraParam** (output_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **NFinalPose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
Ordered tuple with all external camera parameters.
Number of elements: NFinalPose == 7 * NRow / NX

▷ **Errors** (output_control) . . . . . real(-array) ⇝ *real*
Average error distance in pixels.

Example

```
* Read calibration images.
read_image(Image1, 'calib/grid_space.cal.k.000')
read_image(Image2, 'calib/grid_space.cal.k.001')
read_image(Image3, 'calib/grid_space.cal.k.002')
*  Find calibration pattern.
find_caltab(Image1, CalPlate1, 'caltab_big.descr', 3, 112, 5)
find_caltab(Image2, CalPlate2, 'caltab_big.descr', 3, 112, 5)
find_caltab(Image3, CalPlate3, 'caltab_big.descr', 3, 112, 5)
*  Find calibration marks and start poses.
StartCamPar := ['area_scan_division', 0.008, 0.0, 0.000011, 0.000011, \
                384, 288, 768, 576]
find_marks_and_pose(Image1, CalPlate1, 'caltab_big.descr', StartCamPar, \
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord1, CCoord1, \
                    StartPose1)
find_marks_and_pose(Image2, CalPlate2, 'caltab_big.descr', StartCamPar, \
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord2, CCoord2, \
                    StartPose2)
find_marks_and_pose(Image3, CalPlate3, 'caltab_big.descr', StartCamPar, \
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord3, CCoord3, \
                    StartPose3)
*  Read 3D positions of calibration marks.
caltab_points('caltab_big.descr', NX, NY, NZ)
*  Camera calibration.
camera_calibration(NX, NY, NZ, [RCoord1, RCoord2, RCoord3], \
                   [CCoord1, CCoord2, CCoord3], StartCamPar, \
                   [StartPose1, StartPose2, StartPose3], 'all', \
                   CameraParam, NFinalPose, Errors)
*  Write internal camera parameters to file.
write_cam_par(CameraParam, 'campar.dat')
```

Result

`camera_calibration` returns `2` (`H_MSG_TRUE`) if all parameter values are correct and the desired camera parameters have been determined by the minimization algorithm. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

find_marks_and_pose, caltab_points, read_cam_par

Possible Successors

write_pose, pose_to_hom_mat3d, disp_caltab, sim_caltab

Alternatives

calibrate_cameras

See also

find_caltab, find_marks_and_pose, disp_caltab, sim_caltab, write_cam_par, read_cam_par, create_pose, convert_pose_type, write_pose, read_pose, pose_to_hom_mat3d, hom_mat3d_to_pose, caltab_points, gen_caltab, calibrate_cameras

Module

Calibration

6.7 Multi-View

This chapter describes how to calibrate different multi-view camera setups.

In order to achieve high accuracy for your measuring tasks you need to calibrate your camera setup. In comparison to a single-camera setup, some additional requirements apply to the calibration of a multi-view camera setup. The following paragraphs provide explanations regarding the calibration of multi-view camera setups. For general information on camera calibration please refer to the chapter Calibration.

Preparing the Calibration Input Data for Multi-View Camera Setups

Before the actual calibration can be performed, a calibration data model must be prepared (as described in Calibration). For setups with multiple cameras, these additional aspects should be considered:

- The number of cameras in the setup and the number of used calibration objects can be set when calling create_calib_data.
- When specifying the camera type with set_calib_data_cam_param, note that only cameras of the same type (i.e., area scan or line scan) can be calibrated in a single setup.
- Configure the calibration process, e.g., specify the reference camera, using set_calib_data. You can also specify parameters for the complete setup or just configure parameters of individual cameras as well as calibration object poses in the setup.

Performing the Actual Camera Calibration

The calibration performed by calibrate_cameras depends on the camera types that are involved in the calibration setup. While different camera setups require specific conditions when acquiring images, the basic steps of the calibration procedure for setups including projective and/or telecentric cameras are similar:

1. **Building a chain of observation poses:** In the first step, the operator calibrate_cameras tries to build a valid chain of observation poses, that connects all cameras and calibration object poses to the reference camera. Depending on the setup, the conditions for a valid chain of poses differ. For specific information see the respective paragraphs below.

   If there is a camera that cannot be reached (i.e., it is not observing any calibration object pose that can be connected in the chain), the calibration process is terminated with an error. Otherwise, the algorithm initializes all calibration items' poses by going down this chain.

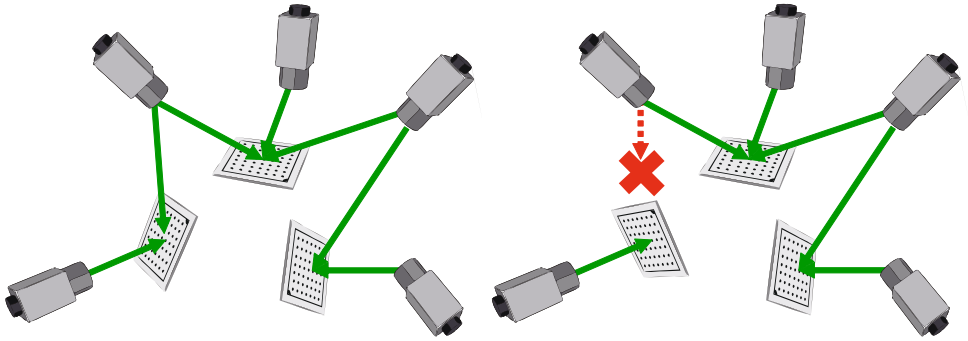
2. **First optimization:** In this step, calibrate_cameras performs the actual optimization for all optimization parameters that were not explicitly excluded from the calibration.

3. **Second optimization:** Based on the so-far calibrated cameras, the algorithm corrects all observations that contain mark contour information (see find_calib_object). Then, the calibration setup is optimized anew for the corrections to take effect. If no contour information was available, this step is skipped.

4. **Compute quality of parameter estimation:** In the last step, `calibrate_cameras` computes the standard deviations and the covariances of the calibrated internal camera parameters.

The following paragraphs give further information about the conditions specific to the camera setups.

Projective area scan cameras For a setup with projective area scan cameras, the calibration is performed in the four steps listed above. The algorithm tries to build a chain of observation poses that connects all cameras and calibration object poses to the reference camera like in the diagram below.



(1) (2)

(1) All cameras can be connected by a chain of observation poses. (2) The leftmost camera is isolated, because the left calibration plate cannot be seen by any other camera.

Possible projective area scan cameras are:

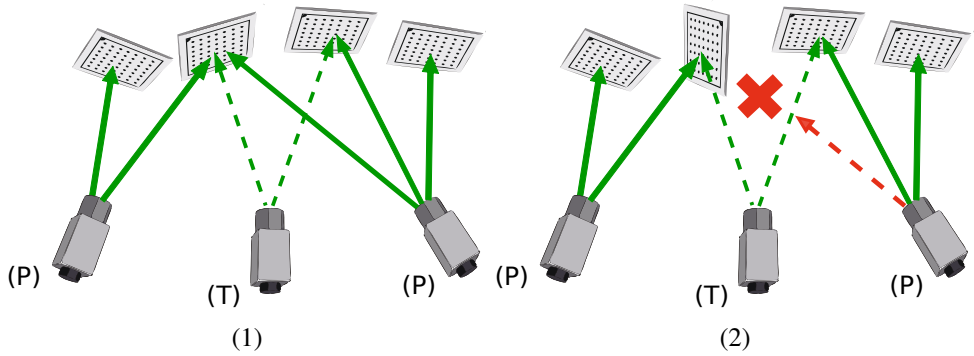
- *'area_scan_division'*
- *'area_scan_polynomial'*
- *'area_scan_tilt_division'*
- *'area_scan_tilt_polynomial'*
- *'area_scan_tilt_image_side_telecentric_division'*
- *'area_scan_tilt_image_side_telecentric_polynomial'*
- *'area_scan_hypercentric_division'*
- *'area_scan_hypercentric_polynomial'*

Telecentric area scan cameras For a setup with telecentric area scan cameras, similar to projective area scan cameras, the same four steps that are listed above are executed. In the first step (building a chain of observation poses that connects all cameras and calibration objects), additional conditions must hold. Since the pose of an object can only be determined up to a translation along the optical axis, each calibration object must be observed by at least two cameras to determine its relative location. Otherwise, its pose is excluded from the calibration. Also, since a planar calibration object appears the same from two different observation angles, the relative pose of the cameras among each other cannot be determined unambiguously. Therefore, there are always two valid alternative relative poses. Both alternatives result in a consistent camera setup which can be used for measuring. Since the ambiguity cannot be resolved, the first of the alternatives is returned. Note that, if the returned pose is not the real pose but the alternative one, then this will result in a mirrored reconstruction.

Possible telecentric area scan cameras are:

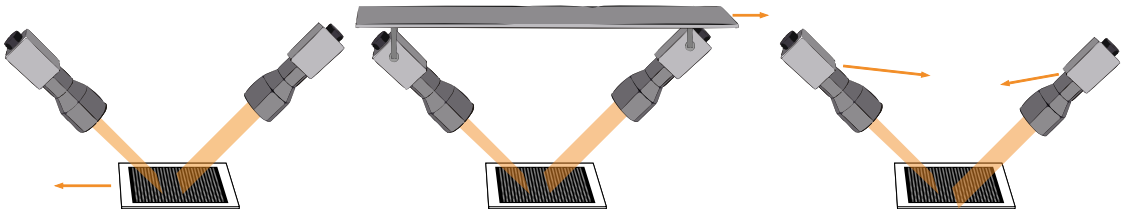
- *'area_scan_telecentric_division'*
- *'area_scan_telecentric_polynomial'*
- *'area_scan_tilt_bilateral_telecentric_division'*
- *'area_scan_tilt_bilateral_telecentric_polynomial'*
- *'area_scan_tilt_object_side_telecentric_division'*
- *'area_scan_tilt_object_side_telecentric_polynomial'*

Projective and telecentric area scan cameras For a mixed setup with projective and telecentric area scan cameras, the algorithm performs the same four steps as enumerated above. Possible ambiguities during the first step (building a chain of observation poses that connects all cameras and calibration objects), as described above for the setup with telecentric cameras, can be resolved as long as there exists a chain of observation poses consisting of all perspective cameras and a sufficient number of calibration objects. Here, sufficient number means that each telecentric camera observes at least two calibration objects of this chain.



Mixed calibration setup with perspective (P) and telecentric (T) area scan cameras. (1) All perspective cameras are connected by a chain of observation poses that only contains perspective cameras. (2) The second calibration plate (from the left) is not observed by the rightmost perspective camera. Therefore, the relative pose between both perspective cameras cannot be determined uniquely.

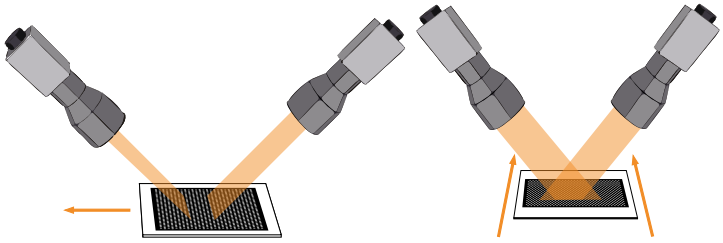
Line scan cameras Setups with telecentric line scan cameras (*'line_scan_telecentric'*) behave identically to setups with telecentric area scan cameras and the same restrictions and ambiguities that are described above apply. For this type of setup, two possible configurations can be distinguished. In the first configuration, all cameras are mounted rigidly and stationary and the object is moved linearly in front of the cameras. Alternatively, all cameras are mounted rigidly with respect to each other and are moved across the object by the same linear actuator. In both cases, all cameras share a common motion vector, which is modeled in the camera coordinate system of the reference camera and is transformed to the camera coordinate systems of all other cameras by the rotation part of the respective camera's pose. This configuration is assumed by default. In the second configuration, the cameras are moved by independent linear actuators in different directions. In this case, each camera has its own independent motion vector. The type of configuration can be selected with `set_calib_data`.



Different configurations of telecentric line scan camera setups can be distinguished: (1) Only one motion vector needs to be computed if the cameras are mounted stationary while the object is moved linearly, (2) or if the cameras are moved across the object while mounted rigidly to each other. (3) Alternatively, if the cameras are moved independently from each other, a motion vector is determined for each camera.

Note that two different stereo setups are common for telecentric line scan cameras. For both setups, a linear, constant motion is assumed for the observed object or the camera system respectively.

- For along-track setups one camera is placed in front, looking in backwards direction, while the second camera is mounted behind, looking forwards, both at an suitable angle in respect to the motion vector.
- The cameras in an across-track setup are all directed perpendicular to the motion vector, while the viewing planes are approximately coplanar. Therefore, the depth of field is rather limited. Precise measurements are only possible in areas where the depth of field of the individual cameras overlap.



(1) (2)

Stereo setups for telecentric line scan cameras: (1) Along-track setup and (2) Across-track setup.

For setups with projective line scan cameras (*'line_scan'*), the following restriction exists: only one camera can be calibrated and only one calibration object per setup can be used.

Finally, for calibration plates with rectangularly arranged marks (see `gen_caltab`) all observations must contain the projection coordinates of all calibration marks of the calibration object. For calibration plates with hexagonally arranged marks (see `create_caltab`) this restriction is not applied. You can find further information about calibration plates and the acquisition of calibration images in the section "Additional information about the calibration process" within the chapter Calibration.

Checking the Success of the Calibration

If more than one camera is calibrated simultaneously, the value of `Error` is more difficult to judge. As a rule of thumb, `Error` should be as small as possible and at least smaller than 1.0, thus indicating that a subpixel precise evaluation of the data is possible with the calibrated parameters. This value might be difficult to reach in particular configurations. For further analysis of the quality of the calibration, refer to the standard deviations and covariances of the estimated parameters.

Getting the Calibration Results

The results of the calibration, i.e., internal camera parameters, camera poses (external camera parameters), calibration objects poses etc., can be queried with `get_calib_data`.

Note that the poses of telecentric cameras can only be determined up to a displacement along the z-axis of the coordinate system of the respective camera (perpendicular to the image plane). Therefore, all camera poses are moved along this axis until they all lie on a common sphere. The center of the sphere is defined by the pose of the first calibration object. The radius of the sphere depends on the calibration setup. If projective and telecentric area scan cameras are calibrated, the radius is the maximum over all distances from the perspective cameras to the first calibration object. Otherwise, if only telecentric area scan cameras are considered, the radius is equal to 1 m.

Further Information

Learn about the calibration of multi-camera setups and many other topics in interactive online courses at our MVTec Academy .

calibrate_cameras (: : CalibDataID : Error)

Determine all camera parameters by a simultaneous minimization process.

The operator `calibrate_cameras` calculates the internal and external camera parameters of a calibration data model specified in `CalibDataID`. The calibration data model describes a setup of one or more cameras and is specified during the creation of the data model. You can find detailed information about the calibration process in the chapter reference Calibration.

The root mean square error (RMSE) of the back projection of the optimization is returned in `Error` (in pixels). The error gives a general indication whether the optimization was successful. You can find more details about the RMSE in the chapter reference mentioned above.

Parameters

▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **Error** (output_control) . . . . . number ⇝ *real*
Back projection root mean square error (RMSE) of the optimization.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_calib_data`, `set_calib_data_cam_param`, `set_calib_data_calib_object`, `set_calib_data_observ_points`, `find_calib_object`, `set_calib_data`, `remove_calib_data_observ`

Possible Successors

`get_calib_data`

References

Carsten Steger: "A Comprehensive and Versatile Camera Model for Cameras with Tilt Lenses"; International Journal of Computer Vision, vol. 123, no. 2, pp. 121-159, 2017.

Carsten Steger, Markus Ulrich, Christian Wiedemann: "Machine Vision Algorithms and Applications"; Wiley-VCH, Weinheim, 2nd Edition, 2018.

Markus Ulrich, Carsten Steger: "A Camera Model for Cameras with Hypercentric Lenses and Some Example Applications"; Machine Vision and Applications, vol. 30, no. 6, pp. 1013-1028, 2019.

Carsten Steger, Markus Ulrich: "A Camera Model for Line-Scan Cameras with Telecentric Lenses"; International Journal of Computer Vision, vol. 129, no. 1, pp. 80-99, 2021.

Carsten Steger, Markus Ulrich: "A Multi-view Camera Model for Line-Scan Cameras with Telecentric Lenses"; Journal of Mathematical Imaging and Vision, vol. 64, no. 2, pp. 105-130, 2022.

Module

Calibration

clear_calib_data (: : CalibDataID :)

Free the memory of a calibration data model.

The operator `clear_calib_data` frees the memory of the calibration data model `CalibDataID`. After calling `clear_calib_data`, the model can no longer be used. The handle `CalibDataID` becomes invalid.

Parameters

▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

Calibration

clear_camera_setup_model (: : CameraSetupModelID :)

Free the memory of a calibration setup model.

The operator clear_camera_setup_model frees the memory of a camera setup model that was created by create_camera_setup_model or read_camera_setup_model or was returned as a result by get_calib_data. After calling clear_camera_setup_model, the model can no longer be used. The handle CameraSetupModelID becomes invalid.

Parameters

▷ **CameraSetupModelID** (input_control) . . . . . camera_setup_model ⇝ *handle*
Handle of the camera setup model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CameraSetupModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

Calibration

create_calib_data (: : CalibSetup, NumCameras,
NumCalibObjects : CalibDataID)

Create a HALCON calibration data model.

The operator create_calib_data creates a generic calibration data model that stores

- the description of a camera calibration setup,
- settings for the calibration process,
- the calibration data, and
- the results of the camera calibration or the hand-eye calibration.

In the parameter CalibSetup, you specify the calibration setup type. Currently, five types are supported. A model of the type *'calibration_object'* is used to calibrate the internal camera parameters and the camera poses of one or more cameras based on the metric information extracted from observations of calibration objects.

A model of type *'hand_eye_moving_cam'*, *'hand_eye_stationary_cam'*, *'hand_eye_scara_moving_cam'*, or *'hand_eye_scara_stationary_cam'* is used to perform a hand-eye calibration based on observations of a calibration object and corresponding poses of a robot tool in the robot base coordinate system. The latter four model types on the one hand distinguish whether the camera or the calibration object is moved by the robot and on the other hand distinguish whether an articulated robot or a SCARA robot is calibrated. The arm of an articulated robot has

three rotary joints typically covering 6 degrees of freedom (3 translations and 3 rotations). SCARA robots have two parallel rotary joints and one parallel prismatic joint covering only 4 degrees of freedom (3 translations and 1 rotation). Loosely speaking, an articulated robot is able to tilt its end effector while a SCARA robot is not.

NumCameras specifies the number of cameras that are calibrated simultaneously in the setup. NumCalibObjects specifies the number of calibration objects observed by the cameras. Please note that for camera calibrations with line scan cameras with perspective lenses only a single calibration object is allowed (NumCalibObjects=1). For hand-eye calibrations, only two setups are currently supported: either one area scan projective camera and one calibration object (NumCameras=1, NumCalibObjects=1) or a general sensor with no calibration object (NumCameras=0, NumCalibObjects=0). **Attention:** The four hand-eye calibration models do not support telecentric cameras.

CalibDataID returns a handle of the new calibration data model. You pass this handle to other operators to collect the description of the camera setup, the calibration settings, and the calibration data. For camera calibrations, you pass it to calibrate_cameras, which performs the actual camera calibration and stores the calibration results in the calibration data model. For a detailed description of the preparation process, please refer to the chapter Calibration. For hand-eye calibrations, you pass it to calibrate_hand_eye, which performs the actual hand-eye calibration and stores the calibration results in the calibration data model. For a detailed description of the preparation process, please refer to the operator calibrate_hand_eye.

Parameters

▷ **CalibSetup** (input_control) . . . . . string ⇝ *string*
Type of the calibration setup.
Default: 'calibration_object'
List of values: CalibSetup ∈ {'calibration_object', 'hand_eye_moving_cam', 'hand_eye_stationary_cam', 'hand_eye_scara_moving_cam', 'hand_eye_scara_stationary_cam'}

▷ **NumCameras** (input_control) . . . . . number ⇝ *integer*
Number of cameras in the calibration setup.
Default: 1
Restriction: NumCameras >= 0

▷ **NumCalibObjects** (input_control) . . . . . number ⇝ *integer*
Number of calibration objects.
Default: 1
Restriction: NumCalibObjects >= 0

▷ **CalibDataID** (output_control) . . . . . calib_data ⇝ *handle*
Handle of the created calibration data model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_calib_data_cam_param, set_calib_data_calib_object

Module

Calibration

create_camera_setup_model (: : NumCameras : CameraSetupModelID)

Create a model for a setup of calibrated cameras.

The operator create_camera_setup_model creates a new camera setup model and returns a handle to it in CameraSetupModelID. The camera setup comprises a fixed number of cameras, which is specified by NumCameras and cannot be changed once the model was created. For each camera, the setup stores its internal parameters, covariances of the internal parameters (optional) and a pose of the camera.

Using set_camera_setup_param, you can change the coordinate system in which the cameras are represented: You can either select a camera and convert all camera poses to be relative to this camera or you can apply a general coordinate transformation, which moves the setup's coordinate system into an arbitrary pose. Changing the coordinate system of the camera setup is particularly useful in cases, where, e.g., you want to represent the cameras in the coordinate system of an object being observed by the cameras. This concept is further demonstrated in the example below.

The internal parameters and pose of a camera are set or modified by set_camera_setup_cam_param. Further camera parameters and general setup parameters can be set by set_camera_setup_param as well. All parameters can be read back by get_camera_setup_param.

A camera setup model can be saved into a file by write_camera_setup_model and read back by read_camera_setup_model.

Parameters

▷ **NumCameras** (input_control) . . . . . integer ⇝ *integer*
Number of cameras in the setup.
Default: 2
Suggested values: NumCameras ∈ {1, 2, 3, 4}
Restriction: NumCameras >= 1

▷ **CameraSetupModelID** (output_control) . . . . . camera_setup_model ⇝ *handle*
Handle to the camera setup model.

Example

```
* Create camera setup of three cameras.
create_camera_setup_model (3, CameraSetupModelID)
gen_cam_par_area_scan_division (0.006, 0, 8.3e-6, 8.3e-6,\
              512, 384, 1024, 768, StartCamPar)
* Camera 0 is located in the origin.
set_camera_setup_cam_param (CameraSetupModelID, 0, [], StartCamPar,\
                            [0, 0, 0, 0, 0, 0, 0])

* Camera 1 is shifted 0.07 m in positive x-direction relative to
* camera 0.
set_camera_setup_cam_param (CameraSetupModelID, 1, [], StartCamPar,\
                            [0.07, 0, 0, 0, 0, 0, 0])

* Camera 2 is shifted 0.1 m in negative y-direction relative to
* camera 0.
set_camera_setup_cam_param (CameraSetupModelID, 2, [], StartCamPar,\
                            [0.0, -0.1, 0, 0, 0, 0, 0])

* There is an object, which is 0.5 away from the origin in
* z-direction, and is facing the origin.
ObjectPose := [0, 0, 0.5, 180, 0, 0, 0]
* Place the setup's origin in the object.
set_camera_setup_param (CameraSetupModelID, 'general', 'coord_transf_pose',\
                        ObjectPose)

* Now the camera poses are given relative to the object.
get_camera_setup_param (CameraSetupModelID, 0, 'pose', CamPose0)
* CamPose0 is equivalent to [0.0, 0.0, 0.5, 180.0, 0.0, 0.0, 0]

get_camera_setup_param (CameraSetupModelID, 1, 'pose', CamPose1)
* CamPose1 is equivalent to [0.07, 0.0, 0.5, 180.0, 0.0, 0.0, 0]

get_camera_setup_param (CameraSetupModelID, 2, 'pose', CamPose2)
* CamPose2 is equivalent to [0.0, 0.1, 0.5, 180.0, 0.0, 0.0, 0]
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_camera_setup_param

Module

Calibration

deserialize_calib_data (: : SerializedItemHandle : CalibDataID)

Deserialize a serialized calibration data model.

deserialize_calib_data deserializes a calibration data model, that was serialized by serialize_calib_data (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized calibration data model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created calibration data model with the handle CalibDataID.

Note that serialize_calib_data does not serialize any calibration results. Yet, calibrate_cameras can be called for a fully configured calibration model immediately after the deserialization. All calibration results are accessible afterwards.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **CalibDataID** (output_control) ..... calib_data ⇝ *handle*
Handle of a calibration data model.

Result

If the parameters are valid, the operator deserialize_calib_data returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_calib_data

Module

Calibration

deserialize_camera_setup_model (
: : SerializedItemHandle : CameraSetupModelID)

Deserialize a serialized camera setup model.

deserialize_camera_setup_model deserializes a camera setup model, that was serialized by serialize_camera_setup_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized camera setup model is defined by the handle

SerializedItemHandle. The deserialized values are stored in an automatically created camera setup model with the handle CameraSetupModelID.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **CameraSetupModelID** (output_control) ..... camera_setup_model ⇝ *handle*
Handle to the camera setup model.

Result

If the parameters are valid, the operator deserialize_camera_setup_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_camera_setup_model

Module

Calibration

get_calib_data (: : CalibDataID, ItemType, ItemIdx, DataName : DataValue)

Query data stored or computed in a calibration data model.

With the operator get_calib_data, you can query data of the calibration data model CalibDataID.

Note that in the following, all 'pose'-related data is given in relation to the coordinate system of the model's *reference camera*, which can be set with set_calib_data and queried with get_calib_data. By default, the first camera (camera index 0) is used as reference camera.

The calibration data model contains various kinds of data. How to query specific data of the calibration data model is described for different categories of data:

- Model-related data (ItemType *'model'*)
- Camera-related data (ItemType *'camera'*)
- Data related to calibration objects (ItemType *'calib_obj'*)
- Data related to calibration object poses (ItemType *'calib_obj_pose'*)
- Hand-eye calibration related data (different values for ItemType)

Before we describe the individual data you can query in detail, we provide you with an overview on which data is available after the individual steps of the calibration processes. When calibrating cameras or a hand-eye system, several operators are called that step by step fill the calibration data model with content. In the following, for each operator a table lists the data that is added to the model. Additionally, you find information about the combinations of the values for ItemType, ItemIdx, and DataName that are needed to query the information with get_calib_data. For the different indices that are used within the tables the following abbreviations (or potential variable names) are used:

- Camera index: CameraIdx
- Calibration object index: CalibObjIdx
- Calibration object pose index: CalibObjPoseIdx

Detailed descriptions of the data that can be queried can then be found in the specific sections that handle the different categories of data individually.

To get detailed information about the calibration process of your camera setup see the chapter Calibration.

Content of Calibration Data Model When Calibrating Cameras

For each operator that extends the calibration model, a table is provided to give an overview on the respective data:

- `create_calib_data`:

Data added to the model	ItemType	ItemIdx	DataName
Type of the calibration data model	*'model'*	*'general'*	*'type'*
Number of cameras	*'model'*	*'general'*	*'num_cameras'*
Number of calibration objects	*'model'*	*'general'*	*'num_calib_objs'*

- `set_calib_data_cam_param`:

Data added to the model	ItemType	ItemIdx	DataName
Camera types	*'camera'*	CameraIdx	*'type'*
Initial internal camera parameters	*'camera'*	CameraIdx	*'init_params'*

- `set_calib_data_calib_object`:

Data added to the model	ItemType	ItemIdx	DataName
Numbers of calibration marks of the calibration objects	*'calib_obj'*	CalibObjIdx	*'num_marks'*
Coordinates of the calibration marks of the calibration objects relative to their calibration object coordinate systems	*'calib_obj'*	CalibObjIdx	*'x'*, *'y'*, *'z'*

For standard HALCON calibration plates, further calibration plate specific information is added to the model, which is not accessible with `get_calib_data` but can be obtained directly from the corresponding calibration plate description files instead (for details about the description files see `create_caltab` for a calibration plate with hexagonally arranged marks and `gen_caltab` for a calibration plate with rectangularly arranged marks).

- `find_calib_object` (for standard HALCON calibration plates):

Data added to the model	accessible with
Observed image coordinates of the calibration marks	`get_calib_data_observ_points`
Observed contours of the calibration marks	`get_calib_data_observ_contours`
Observed poses of the calibration plate relative to the camera coordinate system	`get_calib_data_observ_pose` or `get_calib_data_observ_points`

- `set_calib_data_observ_points` (for other calibration objects than the HALCON calibration plates):

Data added to the model	accessible with
Observed image coordinates of the calibration marks	`get_calib_data_observ_points`

- set_calib_data:

Data added to the model	ItemType	ItemIdx	DataName
Reference camera	*'model'*	*'general'*	*'reference_camera'*
Internal and external camera parameters to calibrate	*'camera'*	*'general'* or CameraIdx	*'calib_settings'*
Internal and external camera parameters to be excluded from the calibration	*'camera'*	*'general'* or CameraIdx	*'excluded_settings'*
For stereo setups with telecentric line scan cameras: Do the cameras have a **common motion vector**?	*'model'*	*'general'*	*'common_motion_vector'*
Calibration object pose settings to be optimized	*'calib_obj_pose'*	*'general'* or [CalibObjIdx, CalibObjPoseIdx]	*'calib_settings'*
Calibration object pose settings to be excluded from the calibration	*'calib_obj_pose'*	*'general'* or [CalibObjIdx, CalibObjPoseIdx]	*'excluded_settings'*

- calibrate_cameras:

Data added to the model	ItemType	ItemIdx	DataName
Camera setup model (needed for multi-view stereo reconstruction)	*'model'*	*'general'*	*'camera_setup_model'*
Optimized internal camera parameters	*'camera'*	CameraIdx	*'params'*
Standard deviations of the optimized internal camera parameters	*'camera'*	CameraIdx	*'params_deviations'*
Covariance matrices of the optimized internal camera parameters	*'camera'*	CameraIdx	*'params_covariances'*
Labels for the internal camera parameters	*'camera'*	CameraIdx	*'params_labels'*
Initial external camera parameters (camera poses)	*'camera'*	CameraIdx	*'init_pose'*
Optimized external camera parameters (camera poses)	*'camera'*	CameraIdx	*'pose'*
Labels for the external camera parameters (camera poses)	*'camera'*	CameraIdx	*'pose_labels'*
Initial calibration object poses	*'calib_obj_pose'*	[CalibObjIdx, CalibObjPoseIdx]	*'init_pose'*
Optimized calibration object poses	*'calib_obj_pose'*	[CalibObjIdx, CalibObjPoseIdx]	*'pose'*
Labels for the calibration object pose parameters	*'calib_obj_pose'*	[CalibObjIdx, CalibObjPoseIdx]	*'pose_labels'*

Content of Calibration Data Model When Performing Hand-Eye Calibration

For each operator that extends the calibration model when performing hand-eye calibration, a table is provided to give an overview on the respective data:

- create_calib_data:
  See the section **'Content of Calibration Data Model When Calibrating Cameras'**.
- set_calib_data:

Data added to the model	ItemType	ItemIdx	DataName
Optimization method	*'model'*	*'general'*	*'optimization_method'*
Poses of the robot tool in robot base coordinates	*'tool'*	CalibObjPoseIdx	*'tool_in_base_pose'*

- set_calib_data_observ_pose (observations obtained by 3D sensors):

Data added to the model	accessible with
Observed calibration object poses	get_calib_data_observ_pose

- set_calib_data_cam_param, set_calib_data_calib_object, and find_calib_object or set_calib_data_observ_points (observations obtained by cameras):
  See the section 'Content of Calibration Data Model When Calibrating Cameras'.

- calibrate_hand_eye:
  Moving camera scenario:

Data added to the model	ItemType	ItemIdx	DataName
Pose of robot tool in camera coordinate system	*'camera'*	*0*	*'tool_in_cam_pose'*
Pose of calibration object in robot base coordinate system	*'calib_obj'*	*0*	*'obj_in_base_pose'*
Standard deviations of the Pose of the robot tool in camera coordinate system	*'camera'*	*0*	*'tool_in_cam_pose_deviations'*
Covariance matrices of the Pose of the robot tool in camera coordinate system	*'camera'*	*0*	*'tool_in_cam_pose_covariances'*
Standard deviations of the Pose of the calibration object in robot base coordinate system	*'calib_obj'*	*0*	*'obj_in_base_pose_deviations'*
Covariance matrices of the Pose of the calibration object in robot base coordinate system	*'calib_obj'*	*0*	*'obj_in_base_pose_covariances'*

Stationary camera scenario:

Data added to the model	ItemType	ItemIdx	DataName
Pose of robot base in camera coordinate system	*'camera'*	*0*	*'base_in_cam_pose'*
Pose of calibration object in robot tool coordinate system	*'calib_obj'*	*0*	*'obj_in_tool_pose'*
Standard deviations of the Pose of the robot base in camera coordinate system	*'camera'*	*0*	*'base_in_cam_pose_deviations'*
Covariance matrices of the Pose of the robot base in camera coordinate system	*'camera'*	*0*	*'base_in_cam_pose_covariances'*
Standard deviations of the Pose of the calibration object in robot tool coordinate system	*'calib_obj'*	*0*	*'obj_in_tool_pose_deviations'*
Covariance matrices of the Pose of the calibration object in robot tool coordinate system	*'calib_obj'*	*0*	*'obj_in_tool_covariances'*

Both hand-eye scenarios:

Data added to the model	ItemType	ItemIdx	DataName
Calibrated poses of the calibration object in camera coordinate system (not available for SCARA robots)	*'calib_obj_pose'*	*[0, CalibObjPoseIdx]*	*'pose'*
Root mean square error (RMSE) of the back projection after the optimization of the camera system	*'model'*	*'general'*	*'camera_calib_error'*
Pose error of the complete chain of transformations	*'model'*	*'general'*	*'hand_eye_calib_error'*
Camera setup model (needed for multi-view stereo reconstruction)	*'model'*	*'general'*	*'camera_setup_model'*

Both hand-eye scenarios, if *'optimization_method'* is set to *'stochastic'*:

Data added to the model	ItemType	ItemIdx	DataName
Root mean square error (RMSE) of the back projection into camera images, via pose chain using corrected tool poses	*'model'*	*'general'*	*'camera_calib_error_corrected_tool'*
Pose error of the complete chain of transformations using corrected tool poses	*'model'*	*'general'*	*'hand_eye_calib_error_corrected_tool'*
Standard deviations of the input poses of the robot tool in robot base coordinates	*'tool'*	*'general'*	*'tool_translation_deviation'*, *'tool_rotation_deviation'*
Corrected poses of the robot tool in robot base coordinates	*'tool'*	CalibObjPoseIdx	*'tool_in_base_pose_corrected'*

The following sections describe the parameters for the specific categories of data in more detail.

Model-Related Data

ItemType=*'model'*: ItemIdx must be set to *'general'*.

Depending on the selection in DataName, the following model-related data is then returned in DataValue:

'type': Type of the calibration data model. Currently, the five types *'calibration_object'*, *'hand_eye_stationary_cam'*, *'hand_eye_moving_cam'*, *'hand_eye_scara_stationary_cam'*, and *'hand_eye_scara_moving_cam'* are supported.

'reference_camera': Index of the reference camera for the calibration model. All poses stored in the calibration data model are specified in the coordinate system of this reference camera.

'num_cameras': Number of cameras in the calibration data model (see create_calib_data).

'num_calib_objs': Number of calibration objects in the calibration data model (see create_calib_data).

'common_motion_vector': For stereo setups with telecentric line scan cameras, a string with a Boolean value (i.e., *'true'* or *'false'*) that determines whether the cameras have a common motion vector.

'camera_setup_model': A handle to a camera setup model containing the poses and the internal parameters for the calibrated cameras from the current calibration setup.

'camera_calib_error': The root mean square error (RMSE) of the back projection of the optimization of the camera system. Typically, this error is queried after a hand-eye calibration (`calibrate_hand_eye`) was performed, where internally the camera system is calibrated without returning the error of the camera calibration. The returned error is identical to the error returned by `calibrate_cameras`, except for *'optimization_method'* set to *'stochastic'*, which refines hand-eye poses and camera parameters simultaneously for articulated robots.

'hand_eye_calib_error': After a successful hand-eye calibration, the pose error of the complete chain of transformations is returned. To be more precise, a tuple with four elements is returned, where the first element is the root-mean-square error of the translational part, the second element is the root-mean-square error of the rotational part, the third element is the maximum translational error and the fourth element is the maximum rotational error. The returned errors are identical to the errors returned by `calibrate_hand_eye`.

'optimization_method': Optimization method that was set for the hand-eye calibration (see `set_calib_data`).

'camera_calib_error_corrected_tool': The root mean square error (RMSE) of the back projection of the calibration mark centers into camera images, via the pose chain using corrected tool poses. By contrast, *'camera_calib_error'* uses the direct back projection of *'calib_obj_pose'*. This parameter is only available if *'optimization_method'* is set to *'stochastic'*.

'hand_eye_calib_error_corrected_tool': After a successful hand_eye calibration, the pose error of the complete chain of transformations using corrected tool poses is returned. By contrast, *'hand_eye_calib_error'* uses the input tool poses. This parameter is only available if *'optimization_method'* is set to *'stochastic'*.

The parameters *'reference_camera'*, *'common_motion_vector'*, and *'optimization_method'* can be set with `set_calib_data`. The other parameters are set during the model creation or are a result of the calibration process and cannot be modified.

Camera-Related Data

`ItemType`=*'camera'*: `ItemIdx` determines, if data is queried for all cameras in general or for a specific camera. With `ItemIdx`=*'general'*, the default value of a parameter for all cameras is returned. In contrast, if you pass a valid camera index instead, i.e., a number between 0 and `NumCameras`-1 (`NumCameras` is specified during model creation with `create_calib_data`), only the parameter value of the specified camera is returned.

By selecting the following parameters in `DataName`, you can query which camera parameters are (or have been) optimized during the calibration performed by `calibrate_cameras`:

'calib_settings': List of the camera parameters that are marked for calibration.

'excluded_settings': List of camera parameters that are excluded from the calibration.

These parameters can be modified by a corresponding call to `set_calib_data`.

The following parameters can only be queried for a specific camera, i.e., you must pass a valid camera index in `ItemIdx`:

'type': The camera type that was set with `set_calib_data_cam_param`.

'init_params': Initial internal camera parameters (set with `set_calib_data_cam_param`).

'params': Optimized internal camera parameters.

'params_deviations': Standard deviations of the optimized camera parameters, as estimated at the end of the camera calibration. Note that if the tuple returned for *'params'* contains n elements, the tuple returned for *'params_deviations'* contains $(n-1)$ elements since the camera parameter tuple contains the camera type in the first element of the tuple, whereas the tuple returned for *'params_deviations'* does not contain the camera type.

'params_covariances': Covariance matrix of the optimized camera parameters, as estimated at the end of the camera calibration. Note that if the tuple returned for *'params'* contains n elements, the tuple returned for *'params_covariances'* contains $(n-1) \times (n-1)$ elements since the camera parameter tuple contains the camera type in the first element of the tuple, whereas the tuple returned for *'params_covariances'* does not contain the camera type.

'params_labels': A convenience list of labels for the entries returned by *'params'*. This list is camera-type specific. Note that this list contains the label *'camera_type'* in its first position. If the first element of the tuple is removed, the list refers to the labels of *'params_deviations'* and the labels of the rows and columns of *'params_covariances'*.

'init_pose': Initial camera pose, relative to the current reference camera. It is computed internally based on observation poses during the calibration process (see Calibration).

'pose': Optimized camera pose, relative to the current reference camera. If one single telecentric camera is calibrated, the translation along the z-axis is set to the value 0.0. If more than one telecentric camera is calibrated, the camera poses are moved in direction of their z-axis until they all lie on a sphere centered at the first observed calibration plate. The radius of the sphere corresponds to the longest distance of a camera to the first observed calibration plate. If this calculated distance is smaller than 1 m, the radius is set to 1 m.

'pose_labels': A convenience list of labels for the entries returned by *'pose'*.

The calibrated camera parameters (*'params'* and *'pose'*) can be queried only after a successful execution of `calibrate_cameras`. The initial internal camera parameters *'init_params'* can be queried after a successful call to `set_calib_data_cam_param`.

Data Related to Calibration Objects

`ItemType`=*'calib_obj'*: `ItemIdx` must be set to a valid calibration object index (number between 0 and `NumCalibObjects`-1). `NumCalibObjects` is specified during the model creation with `create_calib_data`.

The following parameters can be queried with `DataName` and are returned in `DataValue`:

'num_marks': Number of calibration marks of the calibration object.

'x', *'y'*, *'z'*: Coordinates of the calibration marks relative to the calibration object coordinate system.

These parameters can be modified with `set_calib_data_calib_object`.

Data Related to Calibration Object Poses

`ItemType`=*'calib_obj_pose'*: `ItemIdx` determines, if data is queried for all calibration object poses in general or for a specific calibration object pose. With `ItemIdx`=*'general'*, the default value of a parameter for all calibration object poses is returned. In contrast, if you pass a valid calibration object index instead, i.e., a tuple containing a valid index pair `[CalibObjIdx, CalibObjPoseIdx]`, only the parameter value of the specified calibration object pose is returned.

By selecting the following parameters in `DataName`, you can query which calibration object pose parameters are (or have been) optimized during the calibration performed by `calibrate_cameras`:

'calib_settings': List of calibration object pose parameters marked for calibration.

'excluded_settings': List of calibration object pose parameters excluded from calibration.

These parameters can be set with `set_calib_data`.

The following parameters can only be queried for a specific calibration object pose, i.e., you must pass a valid index pair `[CalibObjIdx, CalibObjPoseIdx]` in `ItemIdx`:

'init_pose': Initial calibration object pose. It is computed internally based on observation poses during the calibration process (see Calibration). This pose is relative to the current reference camera.

'pose': Optimized calibration object pose, relative to current reference camera.

'pose_labels': A convenience list of labels for the entries returned by *'pose'*.

These parameters cannot be explicitly modified and can only be queried after `calibrate_cameras` was executed.

Hand-Eye Calibration Related Data

`ItemType`=*'tool'*: The following parameters can be queried with `DataName` and are returned in `DataValue`:

'tool_in_base_pose': Pose of the robot tool in robot base coordinates with Index `ItemIdx`. These poses were previously set using `set_calib_data` and served as input for the hand-eye calibration algorithm.

'tool_in_base_pose_corrected': Corrected pose of the robot tool in robot base coordinates of the input *'tool_in_base_pose'* with Index `ItemIdx`. This parameter is only available if *'optimization_method'* is set to *'stochastic'* and after `calibrate_hand_eye` was executed.

'tool_translation_deviation', *'tool_rotation_deviation'*: Standard deviations of the input poses of the robot tool in robot base coordinates. `ItemIdx` has to be set to *'general'*. This parameter is only available if *'optimization_method'* is set to *'stochastic'* and after `calibrate_hand_eye` was executed.

After performing a successful hand-eye calibration using `calibrate_hand_eye`, the following poses can be queried for a calibration data model of type:

'hand_eye_moving_cam', *'hand_eye_scara_moving_cam'*: For `ItemType`=*'camera'* and `DataName`=*'tool_in_cam_pose'*, the pose of the robot tool in the camera coordinate system is returned in `DataValue`. For `ItemType`=*'calib_obj'* and `DataName`=*'obj_in_base_pose'*, the pose of the calibration object in the robot base coordinate system is returned in `DataValue`.

Note that when calibrating SCARA robots, it is not possible to determine the Z translation of *'obj_in_base_pose'*. To eliminate this ambiguity the Z translation *'obj_in_base_pose'* is internally set to 0.0 and the *'tool_in_cam_pose'* is calculated accordingly. It is necessary to determine the true translation in Z after the calibration (see `calibrate_hand_eye`).

The standard deviations and the covariance matrices of the 6 pose parameters of both poses can be queried with *'tool_in_cam_pose_deviations'*, *'tool_in_cam_pose_covariances'* (`ItemType`=*'camera'*), *'obj_in_base_pose_deviations'*, and *'obj_in_base_pose_covariances'* (`ItemType`=*'calib_obj'*). Like poses, they are specified in the units [m] and [°].

'hand_eye_stationary_cam', *'hand_eye_scara_stationary_cam'*: For `ItemType`=*'camera'* and `DataName`=*'base_in_cam_pose'*, the pose of the robot base in the camera coordinate system is returned in `DataValue`. For `ItemType`=*'calib_obj'* and `DataName`=*'obj_in_tool_pose'*, the pose of the calibration object in the robot tool coordinate system is returned in `DataValue`.

Note that when calibrating SCARA robots, it is not possible to determine the Z translation of *'obj_in_tool_pose'*. To eliminate this ambiguity the Z translation of *'obj_in_tool_pose'* is internally set to 0.0 and the *'base_in_cam_pose'* is calculated accordingly. It is necessary to determine the true translation in Z after the calibration (see `calibrate_hand_eye`).

The standard deviations and the covariance matrices of the 6 pose parameters of both poses can be queried with *'base_in_cam_pose_deviations'*, *'base_in_cam_pose_covariances'* (`ItemType`=*'camera'*), *'obj_in_tool_pose_deviations'*, and *'obj_in_tool_pose_covariances'* (`ItemType`=*'calib_obj'*). Like poses, they are specified in the units [m] and [°].

Parameters

▷ **`CalibDataID`** (input_control) ..... calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **`ItemType`** (input_control) ..... string ⇝ *string*
Type of calibration data item.
Default: 'camera'
List of values: `ItemType` ∈ {'model', 'camera', 'calib_obj', 'calib_obj_pose', 'tool'}

▷ **`ItemIdx`** (input_control) ..... integer(-array) ⇝ *integer* / string
Index of the affected item (depending on the selected `ItemType`).
Default: 0
Suggested values: `ItemIdx` ∈ {0, 1, 2, 'general'}

▷ **`DataName`** (input_control) ..... attribute.name(-array) ⇝ *string*
The name of the inspected data.
Default: 'params'
List of values: `DataName` ∈ {'type', 'reference_camera', 'num_cameras', 'num_calib_objs', 'camera_setup_model', 'camera_calib_error', 'camera_calib_error_corrected_tool', 'hand_eye_calib_error', 'hand_eye_calib_error_corrected_tool', 'optimization_method', 'num_marks', 'x', 'y', 'z', 'params', 'pose', 'init_params', 'init_pose', 'params_deviations', 'params_covariances', 'params_labels', 'pose_labels', 'calib_settings', 'excluded_settings', 'common_motion_vector', 'tool_in_cam_pose', 'obj_in_base_pose', 'base_in_cam_pose', 'obj_in_tool_pose', 'tool_in_base_pose', 'tool_in_cam_pose_deviations', 'obj_in_base_pose_deviations', 'base_in_cam_pose_deviations', 'obj_in_tool_pose_deviations', 'tool_in_cam_pose_covariances', 'obj_in_base_pose_covariances', 'base_in_cam_pose_covariances', 'obj_in_tool_pose_covariances', 'tool_translation_deviation', 'tool_rotation_deviation', 'tool_in_base_pose_corrected'}

▷ **DataValue** (output_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
Requested data.

Example

```
* Get the camera type of camera 0.
get_calib_data (CalibDataID, 'camera', 0, 'type', CameraType)

* Get the optimized (calibrated) pose of pose 1 of the
* calibration object 2.
get_calib_data (CalibDataID, 'calib_obj_pose', [2,1], 'pose', CalobjPose)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

calibrate_cameras, calibrate_hand_eye, create_calib_data, read_calib_data

Module

Calibration

get_calib_data_observ_contours (: Contours : CalibDataID, ContourName, CameraIdx, CalibObjIdx, CalibObjPoseIdx :)

Get contour-based observation data from a calibration data model.

The operator get_calib_data_observ_contours reads contour-based observation data from a calibration data model CalibDataID and returns it in Contours. These contours result from a preceding call of find_calib_object. The parameters CameraIdx, CalibObjIdx, and CalibObjPoseIdx are indices of the observing camera, calibration plate, and calibration object pose. Together, they specify an observation from the calibration model. Note that if an observation exists, but it was stored in the calibration model CalibDataID by set_calib_data_observ_points, no contour-based results can be returned.

By setting ContourName to one of the following values, you can select the specific type of the contour results:

'marks': The contours of the calibration plate marks.

'marks_with_hole': The calibration plate marks which contain a hole. In this case, the output returned in Contours are regions instead of contours.

'caltab': The contour of the calibration plate finder pattern.

'last_caltab': The contour of the calibration plate finder pattern, which has been extracted by the last successful preceding call to find_calib_object. Note that the observation of the successful call to find_calib_object is used and consequently the values in CameraIdx, CalibObjIdx, and CalibObjPoseIdx are ignored.

The mentioned finder pattern depends on the calibration plate:

- Calibration plates with hexagonally arranged marks: Special mark hexagon (i.e., a mark and its six neighbors) where either four or six marks contain a hole, see create_caltab.
- Calibration plates with rectangularly arranged marks: The border of the calibration plate with a triangle in one corner.

Parameters

▷ **Contours** (output_object) . . . . . xld_cont(-array) ⇝ *object*
Contour-based result(s).

▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **ContourName** (input_control) . . . . . string ⇝ *string*
Name of contour objects to be returned.
Default: 'marks'
List of values: ContourName ∈ {'marks', 'caltab', 'last_caltab', 'marks_with_hole'}

▷ **CameraIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observing camera.
Default: 0

▷ **CalibObjIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observed calibration plate.
Default: 0

▷ **CalibObjPoseIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observed calibration object pose.
Default: 0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Calibration

get_calib_data_observ_points (: : CalibDataID, CameraIdx, CalibObjIdx, CalibObjPoseIdx : Row, Column, Index, Pose)

Get point-based observation data from a calibration data model.

The operator get_calib_data_observ_points reads point-based observation data from a calibration data model CalibDataID. This operator reads back observation data stored by set_calib_data_observ_points or find_calib_object. See set_calib_data_observ_points for a detailed description of the arguments.

Please note that if set_calib_data_observ_points is used for the calibration, the returned values of Row and Column are the original values that have been set with set_calib_data_observ_points. Similarly, if find_calib_object is used for the extraction, the values of Row and Column returned by get_calib_data_observ_points coincide with the coordinates of the detected points computed by find_calib_object.

Note that get_calib_data_observ_points returns the pose of an uncalibrated model. To get the pose of a calibrated model, use get_calib_data.

Parameters

▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **CameraIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observing camera.
Default: 0

▷ **CalibObjIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observed calibration object.
Default: 0

▷ **CalibObjPoseIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observed calibration object pose.
Default: 0

▷ **Row** (output_control) . . . . . number-array ⇝ *real* / integer
Row coordinates of the detected points.

▷ **Column** (output_control) . . . . . number-array ⇝ *real* / integer
Column coordinates of the detected points.

▷ **Index** (output_control) . . . . . number-array ⇝ *integer* / real
Correspondence of the detected points to the points of the observed calibration object.

▷ **Pose** (output_control) . . . . . number-array ⇝ *real* / integer
Roughly estimated pose of the observed calibration object relative to the observing camera.
Number of elements: 7

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Calibration

get_camera_setup_param (: : CameraSetupModelID, CameraIdx, GenParamName : GenParamValue)

Get generic camera setup model parameters.

The operator get_camera_setup_param can be used to inspect diverse generic parameters of the camera setup model CameraSetupModelID. Two types of parameters can be queried with this operator:

General parameters:

By setting CameraIdx to *'general'* and GenParamName to one of the following values, general camera setup parameters are returned in GenParamValue:

'num_cameras': Number of cameras described in the model. The number of cameras is fixed with the creation of the camera setup model and cannot be changed after that (see create_camera_setup_model).

'camera_calib_error': The root mean square error (RMSE) of the back projection of the optimization of the camera system. This error is identical with the error returned by calibrate_cameras.

'reference_camera': Returns the index of the camera that has been defined as reference camera within the system. If no reference camera has been specified using set_camera_setup_param, the index 0 is returned. If the coordinate system has been moved by setting a pose with the parameter *'coord_transf_pose'* in set_camera_setup_param, the origin of the coordinate system is not located in any of the available cameras. Therefore, the index -1 is returned.

'coord_transf_pose': Returns the pose in which the coordinate system of the setup has been moved. Please note that after setting a reference camera with set_camera_setup_param, the pose of this camera is returned. Adjusting this coordinate system subsequently using the parameter *'coord_transf_pose'* in set_camera_setup_param yields a pose that corresponds to the location and orientation of the desired coordinate system relative to the current one.

Camera parameters:

By setting CameraIdx to a valid setup camera index (a value between 0 and NumCameras-1) and GenParamName to one of the following values, camera-specific parameters are returned in GenParamValue:

'type': Camera type (see set_camera_setup_cam_param).

'params': A tuple with internal camera parameters. The length of the tuple depends on the camera type.

'params_deviations': A tuple representing the standard deviations of the internal camera parameters. The length of the tuple depends on the camera type.

'params_covariances': A tuple representing the covariance matrix if the internal camera parameters. The length of the tuple depends on the camera type.

'pose': Camera pose relative to the setup's coordinate system (see `create_camera_setup_model` for more details).

Note that the camera needs to be set first by `set_camera_setup_cam_param`, before any of its parameters can be inspected by `get_camera_setup_param`. If `CameraIdx` is an index of an undefined camera, the operator returns an error.

For more information about the calibration process of your camera setup see the chapter Calibration.

Parameters

▷ **`CameraSetupModelID`** (input_control) . . . . . camera_setup_model ⇝ *handle*
Handle to the camera setup model.

▷ **`CameraIdx`** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Index of the camera in the setup.
Default: 0
Suggested values: `CameraIdx` ∈ {0, 1, 2, 'general'}

▷ **`GenParamName`** (input_control) . . . . . attribute.name ⇝ *string*
Names of the generic parameters to be queried.
List of values: `GenParamName` ∈ {'camera_calib_error', 'type', 'params', 'params_deviations', 'params_covariances', 'pose', 'reference_camera', 'coord_transf_pose', 'num_cameras'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
Values of the generic parameters to be queried.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Calibration

`query_calib_data_observ_indices` (: : `CalibDataID`, `ItemType`, `ItemIdx` : `Index1`, `Index2`)

Query information about the relations between cameras, calibration objects, and calibration object poses.

A calibration data model (`CalibDataID`) contains a collection of observations, which are added to the model by `set_calib_data_observ_points`. Each observation is associated to an observing camera, an observed calibration object, and a calibration object pose. With the operator `query_calib_data_observ_indices`, you can query observation indices associated to a camera or an calibration object, depending on the parameter `ItemType`.

For `ItemType`=*'camera'*, you must pass a valid camera index in `ItemIdx`. Then, `Index1` returns a list of calibration object indices and `Index2` returns a list of pose indices. Each pair `[Index1[I],Index2[I]]` represents calibration object pose that are 'observed' by camera `ItemIdx`.

For `ItemType`=*'calib_obj'*, you must specify a valid calibration object index in `ItemIdx`. Then, `Index1` returns a list of camera indices and `Index2` returns a list of corresponding calibration object pose indices. Each pair `[Index1[I],Index2[I]]` denotes that camera `Index1[I]` is observing the `Index2[I]`th pose of calibration object `ItemIdx`.

This operator is particularly suitable for accessing observation data of a calibration data model whose configuration is unknown at the moment of its usage (e.g., if it was just read from a file). As a special case, this operator can be used to get the precise list of poses of one calibration object (see the example).

Parameters

▷ **`CalibDataID`** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **ItemType** (input_control) . . . . . string ⇝ *string*
Kind of referred object.
Default: 'camera'
List of values: ItemType ∈ {'camera', 'calib_obj'}

▷ **ItemIdx** (input_control) . . . . . number ⇝ *integer*
Camera index or calibration object index (depending on the selected ItemType).
Default: 0
Suggested values: ItemIdx ∈ {0, 1, 2}

▷ **Index1** (output_control) . . . . . number-array ⇝ *integer*
List of calibration object indices or list of camera indices (depending on ItemType).

▷ **Index2** (output_control) . . . . . number-array ⇝ *integer*
Calibration object numbers.

Example

```
* Read a calibration model from a file.
read_calib_data ('calib_data.ccd',CalibDataID)

* Get calibration object indices assigned to calibration object 0.
query_calib_data_observ_indices (CalibDataID, 'calib_obj', 0, _, \
                                 CalibObjPoseIndices)
* CalibObjPoseIndices contains the list of pose indices of calibration
* object 0. In order to be stored in the model, each calibration object
* needs to be observed by at least one camera in the setup (a calibration
* object pose that is not observed by any camera cannot be stored in
* the model). Typically, a calibration object pose can be observed by more
* than one camera. Hence, some calibration object pose indices might appear
* repeatedly in CalibObjPoseIndices. We use tuple_sort and tuple_uniq to
* extract a unique list of calibration object pose indices for calibration
* object 0.
tuple_sort (CalibObjPoseIndices, CalibObjPoseIndices)
tuple_uniq (CalibObjPoseIndices, CalibObjPoseIndices)

* Get poses of calibration objects observed by camera 2.
calibrate_cameras (CalibDataID, Error)
query_calib_data_observ_indices (CalibDataID, 'camera', 2, CalibObjIndices,\
                                 CalibObjPoseIndices)
for I := 0 to |CalibObjIndices|-1 by 1
  get_calib_data (CalibDataID, 'calib_obj_pose', \
                  [CalibObjIndices[I], CalibObjPoseIndices[I]], \
                  'pose', CalibObjPose)
endfor
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

Calibration

read_calib_data (: : FileName : CalibDataID)

Restore a calibration data model from a file.

The operator read_calib_data restores a calibration data model from a file specified by its FileName and returns a handle to the restored model in CalibDataID. The model file must have been created by write_calib_data.

Note that write_calib_data does not store any calibration results into the file. Yet, calibrate_cameras can be called for a fully configured calibration model immediately after the reading. All calibration results are accessible afterwards.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
The path and file name of the model file.
File extension: .ccd

▷ **CalibDataID** (output_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

Calibration

read_camera_setup_model (: : FileName : CameraSetupModelID)

Restore a camera setup model from a file.

The operator read_camera_setup_model restores a camera setup model from a file specified by its FileName and returns a handle to the restored model in CameraSetupModelID. The model file must have been created by write_camera_setup_model.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
The path and file name of the model file.
File extension: .csm

▷ **CameraSetupModelID** (output_control) . . . . . camera_setup_model ⇝ *handle*
Handle to the camera setup model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

Calibration

remove_calib_data (: : CalibDataID, ItemType, ItemIdx :)

Remove a data set from a calibration data model.

The operator remove_calib_data, removes data from the calibration data model CalibDataID. Currently, only the hand-eye calibration data set can be altered. With ItemType='tool', you can remove the pose of the robot tool (in robot base coordinates), which was used to obtain the observation of the pose of the calibration object with the same index ItemIdx (corresponds to the parameter CalibObjPoseIdx of any of the operators find_calib_object, set_calib_data_observ_pose, or set_calib_data_observ_pose). Note, that the corresponding observation of the calibration object with the same index ItemIdx that was previously set in the model also has to be removed. Otherwise, the operator calibrate_hand_eye will report an error.

Parameters

▷ **CalibDataID** (input_control) ..... calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **ItemType** (input_control) ..... string ⇝ *string*
Type of the calibration data item.
Default: 'tool'
List of values: ItemType ∈ {'tool'}

▷ **ItemIdx** (input_control) ..... number(-array) ⇝ *integer* / string
Index of the affected item.
Default: 0
Suggested values: ItemIdx ∈ {0, 1, 2}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_calib_data, remove_calib_data_observ

Possible Successors

calibrate_hand_eye

See also

calibrate_cameras

Module

Calibration

remove_calib_data_observ (: : CalibDataID, CameraIdx, CalibObjIdx, CalibObjPoseIdx :)

Remove observation data from a calibration data model.

The operator remove_calib_data_observ removes observations that were set in a calibration data model CalibDataID using find_calib_object, set_calib_data_observ_points, or set_calib_data_observ_pose. The parameters CameraIdx, CalibObjIdx, and CalibObjPoseIdx should specify a valid observation from the calibration model. Note that if the calibration data model CalibDataID is used in calibrate_hand_eye, the corresponding tool pose also has to be deleted using remove_calib_data.

Parameters

▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **CameraIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observing camera.
Default: 0

▷ **CalibObjIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observed calibration object.
Default: 0

▷ **CalibObjPoseIdx** (input_control) . . . . . number ⇝ *integer*
Index of the observed calibration object pose.
Default: 0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`find_calib_object`, `set_calib_data_observ_points`, `set_calib_data_observ_pose`

Possible Successors

`remove_calib_data`, `calibrate_cameras`, `calibrate_hand_eye`

Module

Calibration

serialize_calib_data (: : CalibDataID : SerializedItemHandle)

Serialize a calibration data model.

`serialize_calib_data` serializes the data of a calibration data model (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The same data that is written in a file by `write_calib_data` is converted to a serialized item. The calibration data model is defined by the handle `CalibDataID`. The serialized calibration data model is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_calib_data`.

Note that no calibration results are serialized. You can access them with the operator `get_calib_data`, either as individual items or in form of a camera setup model and store them separately.

Parameters

▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_calib_data` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_calib_data

Module

Calibration

serialize_camera_setup_model (: : CameraSetupModelID : SerializedItemHandle)

Serialize a camera setup model.

serialize_camera_setup_model serializes the data of a camera setup model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_camera_setup_model is converted to a serialized item. The camera setup model is defined by the handle CameraSetupModelID. The serialized camera setup model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_camera_setup_model.

Parameters

▷ **CameraSetupModelID** (input_control) ..... camera_setup_model ⇝ *handle*
Handle to the camera setup model.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_camera_setup_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_camera_setup_model

Module

Calibration

set_calib_data (: : CalibDataID, ItemType, ItemIdx, DataName, DataValue :)

Set data in a calibration data model.

With the operator set_calib_data, you can set data in the calibration data model CalibDataID.

Note that an overview on how the calibration data model is filled with data during the processes of camera calibration and hand-eye calibration is provided by the description of get_calib_data.

The calibration data model can contain various kinds of data. How to set specific data in the calibration data model is described for different categories of data:

- Model-related data (ItemType *'model'*)
- Camera-related data (ItemType *'camera'*)
- Data related to calibration object poses (ItemType *'calib_obj_pose'*)
- Hand-eye calibration related data (ItemType *'tool'*)

The parameter ItemIdx lets you select whether the new value should be set for all items of a type or only for an individual one. The parameters to set are passed in DataName, their values in DataValue.

To get detailed information about the calibration process of your camera setup see the chapter Calibration.

Model-related data

ItemType=*'model'*: ItemIdx must be set to *'general'*.

Depending on the selection in DataName, you can set the following model-related parameters to the value passed in DataValue:

'reference_camera': Set the reference camera for the calibration model to the passed camera index. All poses stored in the calibration data model are specified in the coordinate system of the reference camera (see get_calib_data).

'common_motion_vector': For stereo setups with telecentric line scan cameras, a string with a Boolean value (i.e., *'true'* or *'false'*) that determines whether the cameras have a common motion vector.

'optimization_method': Set the optimization method to be used in the hand-eye calibration process. If DataValue=*'linear'* is set, a linear method is used for the hand-eye calibration. If DataValue=*'nonlinear'* is set, a nonlinear method is used for the hand-eye calibration. If DataValue=*'stochastic'* is set, a method is used which also takes the uncertainty of measured observations into account (see calibrate_hand_eye for more details).

Camera-related data

ItemType=*'camera'*: ItemIdx determines, if data is set for all cameras in general or for a specific camera. With ItemIdx=*'general'*, the new settings are applied to all cameras in the model. If you pass a valid camera index instead, i.e., a number between 0 and NumCameras-1 (NumCameras is specified during model creation with create_calib_data), only the specified camera is affected by the changes.

By selecting the following parameters in DataName, you can **specify which camera parameters shall be optimized during the calibration** performed by calibrate_cameras:

'calib_settings': The camera parameters listed in DataValue are marked for optimization for the affected camera(s) (additionally to the camera parameters that were already marked for optimization). Note that by default, all parameters are marked for the optimization. That is, *'calib_settings'* is mainly suited to add previously excluded parameters again.

'excluded_settings': The camera parameters listed in DataValue are excluded from the optimization for the affected camera(s).

The following camera parameters can be passed in DataValue. See Calibration for affected camera types and further details about the parameters.

Internal camera parameters

'focus': Focal length of the lens.

'magnification': Magnification of the lens.

'kappa': Divisional distortion coefficient kappa.

'k1',*'k2'*,*'k3'*: Polynomial radial distortion parameters.

'poly_tan_2': An alias parameter for all polynomial tangential distortion parameters, i.e., p1 and p2.

'poly': An alias parameter for all polynomial distortion parameters, i.e., k1, k2, k3, p1, and p2.

'image_plane_dist': The distance of the tilted image plane from the perspective projection center.

'tilt': Tilt and rotation of the tilt lens.

'cx',*'cy'*: Coordinates of the camera's principal point.

'principal_point': An alias parameter for *'cx'* and *'cy'*.

'sx',*'sy'*: Sensor element dimensions.

'params': All internal camera parameters.

External camera parameters

'alpha',*'beta'*,*'gamma'*: Rotation part of the camera pose.

'transx',*'transy'*,*'transz'*: Translation part of the camera pose.

'pose': All external camera parameters.

Further camera parameters

'vx',*'vy'*,*'vz'*: Motion vector parameters. Note that for stereo setups with telecentric line scan cameras with a common motion vector (i.e., *'common_motion_vector'* = *'true'*), the motion vector optimization parameters of the reference camera determine which parameters of the common motion vector are optimized. For this kind of setup, it is recommended not to exclude any motion vector parameters from the optimization.

'all': All camera parameters.

By default, all parameters are marked for calibration. As an exception, the camera pose is excluded from the optimization for calibration setups with only one camera (`NumCameras`=1). This setting makes the calibration process equivalent to the one performed by `camera_calibration`.

Data related to calibration object poses

`ItemType`=*'calib_obj_pose'*: `ItemIdx` determines, if data is set for all calibration object poses in general or for a specific calibration object pose. With `ItemIdx`=*'general'* the new settings are applied to all calibration object poses in the model. If you pass a valid calibration object pose index instead, i.e., a tuple containing a valid index pair `[CalibObjIdx, CalibObjPoseIdx]`, you specify a calibration object pose, which is affected by the changes.

By selecting the following parameters in `DataName`, you can **specify which calibration object pose parameters shall be optimized during the calibration** performed by `calibrate_cameras`:

'calib_settings': The calibration object pose settings listed in `DataValue` are marked for optimization for the affected pose(s). Note that by default, all calibration pose parameters are marked for the optimization. That is, *'calib_settings'* is mainly suited to add previously excluded parameters again.

'excluded_settings': The calibration object pose settings listed in `DataValue` are excluded from the optimization for the affected pose(s).

The following calibration pose parameters can be passed in `DataValue`:

'alpha',*'beta'*,*'gamma'*: Rotation part of the calibration object pose.

'transx',*'transy'*,*'transz'*: Translation part of the calibration object pose.

'pose': All calibration object pose parameters.

'all': All calibration objects optimization parameters, i.e., the same as *'pose'*.

By default all parameters are marked for calibration.

The current settings for any model item can be queried with the operator `get_calib_data`.

Hand-eye calibration related data

`ItemType`=*'tool'*: `ItemIdx` must be set to a valid calibration object pose index.

By selecting the following parameter in `DataName`, you can set the pose of the robot tool:

'tool_in_base_pose': Set the pose of the robot tool (in robot base coordinates), which was used to obtain the observation of the pose of the calibration object with the same index `ItemIdx` (corresponds to the parameter `CalibObjPoseIdx` of any of the operators `find_calib_object`, `set_calib_data_observ_pose`, or `set_calib_data_observ_points`).

Parameters

▷ **`CalibDataID`** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **`ItemType`** (input_control) . . . . . string ⇝ *string*
Type of calibration data item.
Default: 'model'
List of values: `ItemType` ∈ {'model', 'camera', 'calib_obj_pose', 'tool'}

▷ **`ItemIdx`** (input_control) . . . . . number(-array) ⇝ *integer* / string
Index of the affected item (depending on the selected `ItemType`).
Default: 'general'
Suggested values: `ItemIdx` ∈ {0, 1, 2, 'general'}

▷ **DataName** (input_control) . . . . . attribute.name ⇝ *string*
Parameter(s) to set.
Default: 'reference_camera'
List of values: DataName ∈ {'reference_camera', 'calib_settings', 'excluded_settings', 'common_motion_vector', 'optimization_method', 'tool_in_base_pose'}

▷ **DataValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer
New value(s).
Default: 0
Suggested values: DataValue ∈ {0, 1, 2, 'all', 'pose', 'params', 'alpha', 'beta', 'gamma', 'transx', 'transy', 'transz', 'focus', 'magnification', 'kappa', 'poly', 'poly_tan_2', 'k1', 'k2', 'k3', 'image_plane_dist', 'tilt', 'principal_point', 'cx', 'cy', 'sx', 'sy', 'vx', 'vy', 'vz', 'true', 'false', 'linear', 'nonlinear', 'stochastic'}

Example

```
* Here, the cell size is known exactly, thus it is excluded from
* the optimization.
set_calib_data (CalibDataID, 'camera', 'general', 'excluded_settings', \
                ['sx','sy'])
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_calib_data_observ_points, find_calib_object

Possible Successors

calibrate_cameras, calibrate_hand_eye

Module

Calibration

set_calib_data_calib_object (: : CalibDataID, CalibObjIdx, CalibObjDescr :)

Define a calibration object in a calibration model.

The operator set_calib_data_calib_object defines the calibration object with the index CalibObjIdx in the camera calibration data model CalibDataID. The index must be between 0 and NumCalibObjects-1 (NumCalibObjects is specified during model creation with create_calib_data and can be queried with get_calib_data).

If a calibration object description with index CalibObjIdx is already defined, then the current object description overwrites it (the description is 'substituted'). Note that all NumCalibObjects calibration objects must be set to perform calibrate_cameras.

The parameter CalibObjDescr can be used in two ways:

as a file name: it specifies a calibration plate description file as created with create_caltab or gen_caltab.

as a numerical tuple: it specifies the 3D coordinates of all points of the calibration object. All X, Y, and Z coordinates, respectively, of all points must be packed sequentially in the tuple in form: [X, Y, Z], i.e., [X1, ..., Xn, Y1, ..., Yn, Z1, ..., Zn], where |X| = |Y| = |Z| and all coordinates are in meters.

To query the calibration objects parameters stored earlier in a calibration data model, use get_calib_data.

To get detailed information about the calibration process of your camera setup see the chapter Calibration.

Parameters

▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **CalibObjIdx** (input_control) . . . . . number ⇝ *integer*
Calibration object index.
Default: 0
Suggested values: CalibObjIdx ∈ {0, 1, 2}

▷ **CalibObjDescr** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
3D point coordinates or a description file name.
List of values: CalibObjDescr ∈ {'calplate.cpd', 'calplate_5mm.cpd', 'calplate_10mm.cpd', 'calplate_20mm.cpd', 'calplate_40mm.cpd', 'calplate_80mm.cpd', 'calplate_160mm.cpd', 'calplate_320mm.cpd', 'calplate_640mm.cpd', 'calplate_1200mm.cpd', 'calplate_20mm_dark_on_light.cpd', 'calplate_40mm_dark_on_light.cpd', 'calplate_80mm_dark_on_light.cpd', 'caltab.descr', 'caltab_650um.descr', 'caltab_2500um.descr', 'caltab_6mm.descr', 'caltab_10mm.descr', 'caltab_30mm.descr', 'caltab_100mm.descr', 'caltab_200mm.descr', 'caltab_800mm.descr', 'caltab_small.descr', 'caltab_big.descr'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_calib_data, set_calib_data_cam_param

Possible Successors

set_calib_data_cam_param, set_calib_data_observ_points, find_calib_object

Module

Calibration

set_calib_data_cam_param (: : CalibDataID, CameraIdx, CameraType, CameraParam :)

Set type and initial parameters of a camera in a calibration data model.

The operator set_calib_data_cam_param sets the initial camera parameters CameraParam for the camera with the index CameraIdx in the calibration data model CalibDataID. The parameter CameraIdx must be between 0 and NumCameras-1 (NumCameras is specified during model creation with create_calib_data and can be queried with get_calib_data). If a camera with CameraIdx was already defined, its parameters are overwritten by the current ones (the camera is substituted). In this case, the selection which camera parameters are marked for optimization is reset and maybe has to be set again. Note that all NumCameras cameras must be set to perform calibrate_cameras. The calibration procedure refines these initial parameters. You can find further information about the calibration process of different camera setups in Calibration.

The parameter CameraType is only provided for backwards compatibility. The information about the camera type is contained in the first element of CameraParam. Therefore, CameraType should be set either to its default value *[]* (the recommended option) or to the same value as the first element of CameraParam. In any other case an error is raised.

An overview of all available camera types and their respective parameters is given in CameraParam.

The camera type can be queried later by calling get_calib_data with the arguments ItemType='camera' and DataName='type'. The initial camera parameters can be queried by calling get_calib_data with arguments ItemType='camera' and DataName='init_params'.

Parameters

▷ **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **CameraIdx** (input_control) . . . . . number-array ⇝ *integer* / string
Camera index.
Default: 0
Suggested values: CameraIdx ∈ {'all', 0, 1, 2}

▷ **CameraType** (input_control) . . . . . string(-array) ⇝ *string*
Type of the camera.
Default: []
List of values: CameraType ∈ {[]}

▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Initial camera internal parameters.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_calib_data, set_calib_data_calib_object

Possible Successors

set_calib_data_calib_object, set_calib_data_observ_points, find_calib_object

Module

Calibration

set_calib_data_observ_points (: : CalibDataID, CameraIdx, CalibObjIdx, CalibObjPoseIdx, Row, Column, Index, Pose :)

Set point-based observation data in a calibration data model.

For a calibration model of type CalibSetup='calibration_object' (see create_calib_data), cameras are calibrated based on so-called observations of calibration objects. With set_calib_data_observ_points, you store such an observation in the calibration data model CalibDataID. An observation consists of the following data:

CameraIdx: index of the observing camera

CalibObjIdx: index of the observed calibration object

CalibObjPoseIdx: index of the observed pose of the calibration object. You can choose it freely, without following a strict order. If you specify an index that already exists for the calibration object CalibObjIdx, the corresponding observation data is replaced by the new one. Of course, the same index can be assigned to poses of different calibration objects.

Row, **Column**, **Index**: Extracted image coordinates and corresponding index of the calibration marks of the calibration object. Row and Column are tuples containing the same number of elements. Index can either contain a tuple (of the same length) or the value *'all'*, indicating that the points [Row, Column] correspond in a one-to-one relation to the calibration marks of the calibration object. If the number of row or column coordinates does not match the number of calibration marks, a corresponding error message is returned.

Pose: A roughly estimated pose of the observed calibration object relative to observing camera.

If you are using the HALCON calibration plate, it is recommended to use find_calib_object instead of set_calib_data_observ_points, since the contour information, which it stores in the calibration data model, enables a more precise calibration procedure with calibrate_cameras.

The observation data can be accessed later by calling get_calib_data_observ_points using the same values for the arguments CameraIdx, CalibObjIdx, and CalibObjPoseIdx.

Parameters

- **CalibDataID** (input_control) . . . . . calib_data ⇝ *handle*
  Handle of a calibration data model.
- **CameraIdx** (input_control) . . . . . number ⇝ *integer*
  Index of the observing camera.
  Default: 0
  Suggested values: CameraIdx ∈ {0, 1, 2}
- **CalibObjIdx** (input_control) . . . . . number ⇝ *integer*
  Index of the calibration object.
  Default: 0
  Suggested values: CalibObjIdx ∈ {0, 1, 2}
- **CalibObjPoseIdx** (input_control) . . . . . number ⇝ *integer*
  Index of the observed calibration object.
  Default: 0
  Suggested values: CalibObjPoseIdx ∈ {0, 1, 2}
  Restriction: CalibObjPoseIdx >= 0
- **Row** (input_control) . . . . . number-array ⇝ *real* / integer
  Row coordinates of the extracted points.
- **Column** (input_control) . . . . . number-array ⇝ *real* / integer
  Column coordinates of the extracted points.
- **Index** (input_control) . . . . . number-array ⇝ *integer* / string
  Correspondence of the extracted points to the calibration marks of the observed calibration object.
  Default: 'all'
  Suggested values: Index ∈ {'all', 0, 1, 2}
- **Pose** (input_control) . . . . . number-array ⇝ *real* / integer
  Roughly estimated pose of the observed calibration object relative to the observing camera.
  Number of elements: 7

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

find_marks_and_pose, set_calib_data_cam_param, set_calib_data_calib_object

Possible Successors

set_calib_data, calibrate_cameras

Alternatives

find_calib_object

Module

Calibration

set_camera_setup_cam_param (: : CameraSetupModelID, CameraIdx, CameraType, CameraParam, CameraPose :)

Define type, parameters, and relative pose of a camera in a camera setup model.

The operator set_camera_setup_cam_param defines the internal parameters and the pose of the camera with CameraIdx in the camera setup model CameraSetupModelID. The parameter CameraIdx must be between 0 and NumCameras-1 (see get_camera_setup_param with argument *'num_cameras'*). If a camera with CameraIdx was already defined, its parameters are overwritten by the current ones (the camera is 'substituted').

The number of values in CameraParam depends on the camera type. See the description of set_calib_data_cam_param for a list of values and Calibration for details on camera types and camera parameters.

The parameter CameraType is only provided for backwards compatibility. The information about the camera type is contained in the first element of CameraParam. Therefore, CameraType should be set either to its default value *[]* (the recommended option) or to the same value as the first element of CameraParam. In any other case an error is raised.

The parameter CameraPose specifies the pose of the camera relative to the setup's coordinate system (see set_camera_setup_param for further explanations on the setup's coordinate system).

All of the parameters set by set_camera_setup_cam_param can be read back by get_camera_setup_param. While the camera type can be changed only with a new call to set_camera_setup_cam_param, all other camera parameters can be modified by set_camera_setup_param. Furthermore, set_camera_setup_param can set additional data to a camera: standard deviations or covariances of the internal camera parameters.

Parameters

▷ **CameraSetupModelID** (input_control) . . . . . camera_setup_model ⇝ *handle*
Handle to the camera setup model.

▷ **CameraIdx** (input_control) . . . . . number-array ⇝ *integer*
Index of the camera in the setup.
Suggested values: CameraIdx ∈ {0, 1, 2}

▷ **CameraType** (input_control) . . . . . string(-array) ⇝ *string*
Type of the camera.
Default: []
List of values: CameraType ∈ {[]}

▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **CameraPose** (input_control) . . . . . number-array ⇝ *real* / integer
Pose of the camera relative to the setup's coordinate system.
Number of elements: 7

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Calibration

set_camera_setup_param (: : CameraSetupModelID, CameraIdx, GenParamName, GenParamValue :)

Set generic camera setup model parameters.

The operator set_camera_setup_param can be used to set diverse generic parameters or transformations for the camera setup model CameraSetupModelID. Two types of parameters can be set with this operator:

Coordinate system of the setup and transformation of camera poses:

By setting CameraIdx to *'general'* and GenParamName to one of the following values, you can perform the following general pose transformation for all cameras:

'reference_camera': When setting GenParamValue to a valid camera index, all camera poses are recomputed relative to the coordinate system of this camera.

'coord_transf_pose': When passing a tuple in HALCON pose format in GenParamValue, the current coordinate system is moved into this pose. The pose in GenParamValue represents the location and orientation of the desired coordinate system relative to the current one. All camera poses are recomputed relative to the new coordinate system.

The recomputed camera poses can be inspected with the operator get_camera_setup_param.

Camera parameters:

By setting CameraIdx to a valid setup camera index (a value between 0 and NumCameras-1) and GenParamName to one of the following values, camera specific parameters can be set with GenParamValue:

'params': A tuple with internal camera parameters.

'params_deviations': A tuple with the standard deviations of the internal camera parameters except for CameraType, Width, and Height, thus |params_deviations|=|params|-3. The internal camera parameters are camera-type dependent. See the description of set_calib_data_cam_param for a list of values and calibrate_cameras for details on camera types and camera parameters.

'params_covariances': A tuple with the covariance matrix of the internal camera parameters. The tuple must represent a square matrix whose both dimensions are identical to the number of standard deviation values, thus |params_covariances|=|params_deviations|^2=(|params|-3)^2, see *'params_deviations'*.

'pose': A tuple representing the pose of the camera in HALCON pose format, relative to camera setup's coordinate system. See the above section for further details.

Note that the camera must already be defined in the model, before any of its parameters can be changed by set_camera_setup_param. If CameraIdx is an index of a undefined camera, the operator returns an error.

All parameters can be read back by get_camera_setup_param.

Parameters

▷ **CameraSetupModelID** (input_control) . . . . . camera_setup_model ⇝ *handle*
Handle to the camera setup model.

▷ **CameraIdx** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Unique index of the camera in the setup.
Default: 0
Suggested values: CameraIdx ∈ {0, 1, 2, 'general'}

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Names of the generic parameters to be set.
List of values: GenParamName ∈ {'params', 'params_deviations', 'params_covariances', 'pose', 'reference_camera', 'coord_transf_pose'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
Values of the generic parameters to be set.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_camera_setup_model, read_camera_setup_model

Module

Calibration

write_calib_data (: : CalibDataID, FileName :)

Store a calibration data model into a file.

The operator write_calib_data stores a calibration data model CalibDataID into a file specified by its file name FileName. The information stored in the file includes:

- initial camera parameters
- calibration object descriptions
- observation data
- model settings: generic and specific optimization parameters for both cameras and calibration object poses.

Note that no calibration results are stored in the file. You can access them with the operator get_calib_data, either as individual items or in form of a camera setup model and store them separately.

The calibration data model can be later read with read_calib_data.

Parameters

▷ **CalibDataID** (input_control) ..... calib_data ⇝ *handle*
Handle of a calibration data model.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
The file name of the model to be saved.
File extension: .ccd

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- CalibDataID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

Calibration

write_camera_setup_model (: : CameraSetupModelID, FileName :)

Store a camera setup model into a file.

The operator write_camera_setup_model stores a camera setup model CameraSetupModelID into a file specified by its file name FileName.

The calibration data model can be later read with read_camera_setup_model.

Parameters

▷ **CameraSetupModelID** (input_control) . . . . . camera_setup_model ⇝ *handle*
Handle to the camera setup model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
The file name of the model to be saved.
File extension: .csm

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Calibration

6.8 Projection

cam_par_pose_to_hom_mat3d (: : CameraParam, Pose : HomMat3D)

Convert internal camera parameters and a 3D pose into a 3×4 projection matrix.

cam_par_pose_to_hom_mat3d converts the internal camera parameters CameraParam and the 3D pose Pose, which represent the external camera parameters, into the 3×4 projection matrix HomMat3D, which can be used to project points from 3D to 2D. The conversion can only be performed if the distortion coefficients in CameraParam are 0. If necessary, change_radial_distortion_cam_par must be used to achieve this. The internal camera parameters and the pose are typically obtained with calibrate_cameras.

Parameters

▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **Pose** (input_control) . . . . . pose ⇝ *real* / integer
3D pose.
Number of elements: 7

▷ **HomMat3D** (output_control) . . . . . hom_mat3d ⇝ *real*
3×4 projection matrix.

Result

cam_par_pose_to_hom_mat3d returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

calibrate_cameras, change_radial_distortion_cam_par

Possible Successors

project_point_hom_mat3d, project_hom_point_hom_mat3d

See also

create_pose, hom_mat3d_to_pose, project_3d_point, get_line_of_sight

Module

Foundation

project_3d_point (: : X, Y, Z, CameraParam : Row, Column)

Project 3D points into (sub-)pixel image coordinates.

project_3d_point projects one or more 3D points (with coordinates X, Y, and Z) into the image plane (in pixels) and returns the result in Row and Column. The coordinates X, Y, and Z are given in the camera coordinate system, i.e., they describe the position of the points relative to the camera.

The internal camera parameters CameraParam describe the projection characteristics of the camera (see Calibration for details).

Parameters

- ▷ **X** (input_control) ..... point3d.x-array ⇝ *real*
  X coordinates of the 3D points to be projected in the camera coordinate system.
- ▷ **Y** (input_control) ..... point3d.y-array ⇝ *real*
  Y coordinates of the 3D points to be projected in the camera coordinate system.
- ▷ **Z** (input_control) ..... point3d.z-array ⇝ *real*
  Z coordinates of the 3D points to be projected in the camera coordinate system.
- ▷ **CameraParam** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameters.
- ▷ **Row** (output_control) ..... point.y-array ⇝ *real*
  Row coordinates of the projected points (in pixels).
- ▷ **Column** (output_control) ..... point.x-array ⇝ *real*
  Column coordinates of the projected points (in pixels).

Example

```
* Set internal camera parameters and pose of the world coordinate
* system in camera coordinates.
* Note that, typically, these values are the result of a prior
* calibration.
gen_cam_par_area_scan_division (0.01, -731, 5.2e-006, 5.2e-006, \
                                654, 519, 1280, 1024, CameraParam)
create_pose (0.1, 0.2, 0.3, 40, 50, 60, \
             'Rp+T', 'gba', 'point', WorldPose)
* Convert pose into transformation matrix.
pose_to_hom_mat3d(WorldPose, HomMat3D)
* Transform 3D points from world into the camera coordinate system.
affine_trans_point_3d(HomMat3D, [3.0, 3.2], [4.5, 4.5], [3.8, 4.2], X, Y, Z)
* Project 3D points into image.
project_3d_point(X, Y, Z, CameraParam, Row, Column)
```

Result

project_3d_point returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_cam_par, affine_trans_point_3d

Possible Successors

gen_region_points, gen_region_polygon, disp_polygon

See also

camera_calibration, disp_caltab, read_cam_par, get_line_of_sight, affine_trans_point_3d, image_points_to_world_plane

Module

Calibration

project_hom_point_hom_mat3d (: : HomMat3D, Px, Py, Pz, Pw : Qx, Qy, Qw)

Project a homogeneous 3D point using a 3×4 projection matrix.

projective_trans_hom_point_3d applies the 3×4 projection matrix HomMat3D to all homogeneous input points (Px,Py,Pz,Pw) and returns an array of homogeneous output points (Qx,Qy,Qw). The transformation is described by the homogeneous transformation matrix given in HomMat3D. This corresponds to the following equation (input and output points as homogeneous vectors):

$$\begin{pmatrix} Qx \\ Qy \\ Qw \end{pmatrix} = HomMat3D \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ Pw \end{pmatrix}$$

To transform the homogeneous coordinates to Euclidean coordinates, they must be divided by Qw:

$$\begin{pmatrix} Ex \\ Ey \end{pmatrix} = \begin{pmatrix} \frac{Qx}{Qw} \\ \frac{Qy}{Qw} \end{pmatrix}$$

This can be achieved directly by calling project_point_hom_mat3d. Thus, project_hom_point_hom_mat3d is primarily useful for transforming points or point sets for which the resulting points might lie on the line at infinity, i.e., points that potentially have $Qw = 0$, for which the above division cannot be performed.

Note that, consistent with the conventions used by the projection in calibrate_cameras, Qx corresponds to the column coordinate of an image and Qy corresponds to the row coordinate.

Parameters

- ▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
  3×4 projection matrix.
- ▷ **Px** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input point (x coordinate).
- ▷ **Py** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input point (y coordinate).
- ▷ **Pz** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input point (z coordinate).
- ▷ **Pw** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input point (w coordinate).
- ▷ **Qx** (output_control) . . . . . real(-array) ⇝ *real*
  Output point (x coordinate).
- ▷ **Qy** (output_control) . . . . . real(-array) ⇝ *real*
  Output point (y coordinate).
- ▷ **Qw** (output_control) . . . . . real(-array) ⇝ *real*
  Output point (w coordinate).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

cam_par_pose_to_hom_mat3d

Alternatives

project_point_hom_mat3d, project_3d_point

Module

Foundation

project_point_hom_mat3d (: : HomMat3D, Px, Py, Pz : Qx, Qy)

Project a 3D point using a 3×4 projection matrix.

project_point_hom_mat3d applies the 3×4 projection matrix HomMat3D to all input points (Px,Py,Pz) and returns an array of output points (Qx,Qy). The transformation is described by the 3×4 projection matrix given in HomMat3D. This corresponds to the following equations (input and output points as homogeneous vectors):

$$\begin{pmatrix} Tx \\ Ty \\ Tw \end{pmatrix} = \texttt{HomMat3D} \cdot \begin{pmatrix} \texttt{Px} \\ \texttt{Py} \\ \texttt{Pz} \\ 1 \end{pmatrix}$$

project_point_hom_mat3d then transforms the homogeneous coordinates to Euclidean coordinates by dividing them by Tw:

$$\begin{pmatrix} \texttt{Qx} \\ \texttt{Qy} \end{pmatrix} = \begin{pmatrix} \frac{Tx}{Tw} \\ \frac{Ty}{Tw} \end{pmatrix}$$

If a point on the line at infinity ($Tw = 0$) is created by the transformation, an error is returned. If this is undesired, project_hom_point_hom_mat3d can be used.

Note that, consistent with the conventions used by the projection in calibrate_cameras, Qx corresponds to the column coordinate of an image and Qy corresponds to the row coordinate.

Parameters

- ▷ **HomMat3D** (input_control) . . . hom_mat3d ⇝ *real*
  3×4 projection matrix.
- ▷ **Px** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (x coordinate).
- ▷ **Py** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (y coordinate).
- ▷ **Pz** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (z coordinate).
- ▷ **Qx** (output_control) . . . real(-array) ⇝ *real*
  Output point (x coordinate).
- ▷ **Qy** (output_control) . . . real(-array) ⇝ *real*
  Output point (y coordinate).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

cam_par_pose_to_hom_mat3d

Alternatives

project_hom_point_hom_mat3d, project_3d_point

Module

Foundation

6.9 Rectification

change_radial_distortion_cam_par (: : Mode, CamParamIn, DistortionCoeffs : CamParamOut)

Determine new camera parameters in accordance to the specified radial distortion.

change_radial_distortion_cam_par modifies the internal camera parameters in accordance to the specified radial distortion coefficients DistortionCoeffs. The operator can only be used for area scan cameras (with any lens type) and for line scan cameras with telecentric lenses. Line scan cameras with perspective lenses are not supported.

With the parameter Mode, one of the following modes can be selected:

'fixed': Only the distortion coefficients are modified, the other internal camera parameters remain unchanged. In general, this leads to a change of the visible part of the scene.

'fullsize': For area scan cameras, the scale factors S_x andS_y and the image center point $(C_x, C_y)^T$ are modified in order to preserve the visible part of the scene. For line scan cameras with telecentric lenses, the scale factor S_x, the image center point $(C_x, C_y)^T$, and the V_y component of the motion vector are changed to achieve the this effect. Thus, all points visible in the original image are also visible in the modified (rectified) image. In general, this leads to undefined pixels in the modified image.

'adaptive': A trade-off between the other modes: The visible part of the scene is slightly reduced to prevent undefined pixels in the modified image. The same parameters as for *'fullsize'* are modified.

'preserve_resolution': As in the mode *'fullsize'*, all points visible in the original image are also visible in the modified (rectified) image. For area scan cameras, the scale factors S_x and S_y and the image center point $(C_x, C_y)^T$ are modified. For line scan cameras with telecentric lenses, the scale factor S_x, the image center point $(C_x, C_y)^T$, and potentially the V_y component of the motion vector are changed to achieve the this effect. In general, this leads to undefined pixels in the modified image. In contrast to the mode *'fullsize'*, additionally the size of the modified image is increased such that the image resolution does not decrease in any part of the image.

In all modes, the distortion coefficients in CamParamOut are set to DistortionCoeffs. For telecentric line scan cameras, the motion vector also influences the percieved distortion. For example, a nonzero V_x component leads to skewed pixels. Furthermore, if $V_y \neq S_x/\text{Magnification}$, the pixels appear to be non-square. Therefore, for telecentric line scan cameras, up to three more components can be passed in addition to κ or $(K_1, K_2, K_3, P_1, P_2)$, respectively, in DistortionCoeffs. These specify the new V_x, V_y, and V_z components of the motion vector.

The transformation of a pixel in the modified image into the image plane using CamParamOut results in the same point as the transformation of a pixel in the original image via CamParamIn.

Parameters

▷ **Mode** (input_control) ..... string ⇝ *string*
Mode
Default: 'adaptive'
Suggested values: Mode ∈ {'fullsize', 'adaptive', 'fixed', 'preserve_resolution'}

▷ **CamParamIn** (input_control) ..... campar ⇝ *real* / integer / string
Internal camera parameters (original).

▷ **DistortionCoeffs** (input_control) ..... real(-array) ⇝ *real* / integer
Desired radial distortions.
Number of elements: DistortionCoeffs == 1 || DistortionCoeffs == 5
Default: 0.0

▷ **CamParamOut** (output_control) ..... campar ⇝ *real* / integer / string
Internal camera parameters (modified).

Result

change_radial_distortion_cam_par returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

camera_calibration, read_cam_par

Possible Successors

change_radial_distortion_image, change_radial_distortion_contours_xld, gen_radial_distortion_map

See also

camera_calibration, read_cam_par, change_radial_distortion_image, change_radial_distortion_contours_xld, change_radial_distortion_points

Module

Calibration

change_radial_distortion_contours_xld (Contours : ContoursRectified : CamParamIn, CamParamOut :)

Change the radial distortion of contours.

change_radial_distortion_contours_xld changes the radial distortion of the input contours Contours in accordance to the internal camera parameters CamParamIn and CamParamOut. Each subpixel of an input contour is transformed into the image plane using CamParamIn and subsequently projected into a subpixel of the corresponding contour in ContoursRectified using CamParamOut.

If CamParamOut was computed via change_radial_distortion_cam_par, the contours ContoursRectified are equivalent to Contours obtained with a lens with a modified radial distortion κ. If $\kappa = 0$ the contours are rectified. A subsequent pose estimation (determination of the external camera parameters) is not affected by this operation.

Please note that change_radial_distortion_contours_xld does not work for line scan cameras with perspective lenses.

Parameters

- ▷ **Contours** (input_object) ..... xld_cont(-array) ⇝ *object*
  Original contours.
- ▷ **ContoursRectified** (output_object) ..... xld_cont(-array) ⇝ *object*
  Resulting contours with modified radial distortion.
- ▷ **CamParamIn** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameter for Contours.
- ▷ **CamParamOut** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameter for ContoursRectified.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

change_radial_distortion_cam_par, gen_contours_skeleton_xld, edges_sub_pix, smooth_contours_xld

Possible Successors

gen_polygons_xld, smooth_contours_xld

See also

change_radial_distortion_cam_par, camera_calibration, read_cam_par, change_radial_distortion_image, change_radial_distortion_points

Module

Calibration

change_radial_distortion_image (Image, Region : ImageRectified : CamParamIn, CamParamOut :)

Change the radial distortion of an image.

change_radial_distortion_image changes the radial distortion of the input image Image in accordance to the internal camera parameters CamParamIn and CamParamOut. Each pixel of the output image that lies within the region Region is transformed into the image plane using CamParamOut and subsequently projected into a subpixel of Image using CamParamIn. The resulting gray value is determined by bilinear interpolation. If the subpixel is outside of Image, the corresponding pixel in ImageRectified is set to 'black' and eliminated from the image domain.

If the gray values of all pixels in the output image shall be calculated, it is sufficient to pass an empty object in Region (which must be previously generated by, for example, using gen_empty_obj). This is especially useful if the size of the output image differs from the size of the input image, and hence, it is not possible to simply pass the region of the input image in Region.

If CamParamOut was computed via change_radial_distortion_cam_par, ImageRectified is equivalent to Image obtained with a lens with a modified radial distortion κ. If $\kappa = 0$ the image is rectified. A subsequent pose estimation (determination of the external camera parameters) is not affected by this operation.

Please note that change_radial_distortion_image does not work for line scan cameras with perspective lenses. Instead, you might want to use image_to_world_plane.

Attention

change_radial_distortion_image can be executed on OpenCL devices if the input image does not exceed the maximum size of image objects of the selected device. As the OpenCL implementation uses single precision arithmetic, the results can differ from the CPU implementation.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Original image.
- ▷ **Region** (input_object) ..... region ⇝ *object*
  Region of interest in ImageRectified.
- ▷ **ImageRectified** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Resulting image with modified radial distortion.
- ▷ **CamParamIn** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameter for Image.
- ▷ **CamParamOut** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameter for Image.

Result

change_radial_distortion_image returns 2 (H_MSG_TRUE) if all parameter values are correct. If the input is empty (no input image is available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on channel level.

Possible Predecessors

change_radial_distortion_cam_par, read_image, grab_image

Possible Successors

edges_image, threshold

See also

change_radial_distortion_cam_par, camera_calibration, read_cam_par, change_radial_distortion_contours_xld, change_radial_distortion_points

Module

Calibration

change_radial_distortion_points (: : Row, Col, CamParamIn, CamParamOut : RowChanged, ColChanged)

Change the radial distortion of pixel coordinates.

change_radial_distortion_points changes the radial distortion of input image coordinates (Row, Col) in accordance to the internal camera parameters CamParamIn and CamParamOut. Each input pixel (Row, Col) is transformed into the image plane using CamParamIn and projected into another image using CamParamOut.

Please note that change_radial_distortion_points does not work for line scan cameras with perspective lenses.

Parameters

- ▷ **Row** (input_control) ..... real-array ⇝ *real*
  Original row component of pixel coordinates.
- ▷ **Col** (input_control) ..... real-array ⇝ *real*
  Original column component of pixel coordinates.
- ▷ **CamParamIn** (input_control) ..... campar ⇝ *real* / integer / string
  The inner camera parameters of the camera used to create the input pixel coordinates.
- ▷ **CamParamOut** (input_control) ..... campar ⇝ *real* / integer / string
  The inner camera parameters of a camera.
- ▷ **RowChanged** (output_control) ..... real-array ⇝ *real*
  Row component of pixel coordinates after changing the radial distortion.
- ▷ **ColChanged** (output_control) ..... real-array ⇝ *real*
  Column component of pixel coordinates after changing the radial distortion.

Result

change_radial_distortion_points returns 2 (H_MSG_TRUE) if all parameter values are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

change_radial_distortion_cam_par, camera_calibration, read_cam_par, change_radial_distortion_contours_xld, change_radial_distortion_image

Module

Calibration

contour_to_world_plane_xld (Contours : ContoursTrans : CameraParam, WorldPose, Scale :)

Transform an XLD contour into the plane z=0 of a world coordinate system.

The operator contour_to_world_plane_xld transforms contour points given in Contours into the plane z=0 in a world coordinate system and returns the 3D contour points in ContoursTrans. The world coordinate system is chosen by passing its 3D pose relative to the camera coordinate system in WorldPose. Hence, latter one is expected in the form ${}^{ccs}\mathbf{P}_{wcs}$, where *ccs* denotes the camera coordinate system and *wcs* the world coordinate system (see Transformations / Poses and "Solution Guide III-C - 3D Vision"). In CameraParam you must pass the internal camera parameters (see Calibration for the sequence of the parameters and the underlying camera model).

In many cases CameraParam and WorldPose are the result of calibrating the camera with the operator calibrate_cameras. See below for an example.

With the parameter Scale you can scale the resulting 3D coordinates. The parameter Scale must be specified as the ratio *desired unit/original unit*. The original unit is determined by the coordinates of the calibration object.

If the original unit is meters (which is the case if you use the standard calibration plate), you can set the desired unit directly by selecting *'m'*, *'cm'*, *'mm'* or *'um'* for the parameter `Scale`.

Internally, the operator first computes the line of sight between the projection center and the image point in the camera coordinate system, taking into account the radial distortions. The line of sight is then transformed into the world coordinate system specified in `WorldPose`. By intersecting the plane z=0 with the line of sight the 3D coordinates of the transformed contour `ContoursTrans` are obtained.

Parameters

- ▷ **`Contours`** (input_object) ..... xld_cont(-array) ⇝ *object*
  Input XLD contours to be transformed in image coordinates.
- ▷ **`ContoursTrans`** (output_object) ..... xld_cont(-array) ⇝ *object*
  Transformed XLD contours in world coordinates.
- ▷ **`CameraParam`** (input_control) ..... number-array ⇝ *real* / integer / string
  Internal camera parameters.
- ▷ **`WorldPose`** (input_control) ..... pose ⇝ *real* / integer
  3D pose of the world coordinate system in camera coordinates.
  Number of elements: 7
- ▷ **`Scale`** (input_control) ..... number ⇝ *string* / integer / real
  Scale or dimension
  Default: 'm'
  Suggested values: `Scale` ∈ {'m', 'cm', 'mm', 'microns', 'um', 1.0, 0.01, 0.001, 1.0e-6, 0.0254, 0.3048, 0.9144}
  Restriction: Scale > 0

Example

```
*  Perform camera calibration (with standard calibration plate).
calibrate_cameras (CalibDataID, Error)
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
* Get reference pose (pose 2 of calibration object 0).
get_calib_data (CalibDataID, 'calib_obj_pose', [0,2], 'pose', WorldPose)
*  Compensate thickness of plate.
set_origin_pose(ObjInCameraPose, 0, 0, 0.0006, WorldPose)
*  Transform contours into world coordinate system (unit mm).
contour_to_world_plane_xld(Contours, ContoursTrans, CamParam, \
                           WorldPose, 'mm')
```

Result

`contour_to_world_plane_xld` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_pose`, `hom_mat3d_to_pose`, `camera_calibration`, `hand_eye_calibration`, `set_origin_pose`

See also

`image_points_to_world_plane`

Module

Calibration

gen_image_to_world_plane_map (: Map : CameraParam, WorldPose, WidthIn, HeightIn, WidthMapped, HeightMapped, Scale, MapType :)

Generate a projection map that describes the mapping between the image plane and the plane z=0 of a world coordinate system.

gen_image_to_world_plane_map generates a projection map Map, which describes the mapping between the image plane and the plane z=0 (plane of measurements) in a world coordinate system. This map can be used to rectify an image with the operator map_image. The rectified image shows neither radial nor perspective distortions; it corresponds to an image acquired by a distortion-free camera that looks perpendicularly onto the plane of measurements. The world coordinate system (*wcs*) is chosen by passing its 3D pose relative to the camera coordinate system (*ccs*) in WorldPose. Thus the pose is expected in the form ${}^{ccs}\mathbf{P}_{wcs}$ (see Transformations / Poses and "Solution Guide III-C - 3D Vision"). In CameraParam you must pass the internal camera parameters (see Calibration for the sequence of the parameters and the underlying camera model).

In many cases CameraParam and WorldPose are the result of calibrating the camera with the operator calibrate_cameras. See below for an example.

The size of the images to be mapped can be specified by the parameters WidthIn and HeightIn. The pixel position of the upper left corner of the output image is determined by the origin of the world coordinate system. The size of the output image can be chosen by the parameters WidthMapped, HeightMapped, and Scale. WidthMapped and HeightMapped must be given in pixels.

The parameter Scale can be used to specify the size of a pixel in the transformed image. There are two ways to use this parameter:

Scale pixels to metric units:

Scale the image such that one pixel in the transformed image corresponds to a metric unit, e.g., setting *'mm'* determines that a pixel in the transformed image corresponds to the area 1mm × 1mm in the plane of measurements. For this, the original unit needs to be meters. This is the case if you use a standard calibration plate.

List of values: *'m'*, *'cm'*, *'mm'*, *'microns'*, *'um'*.

Default: *'m'*.

Control scaling manually:

Scale the image by giving a number that determines the ratio of *original unit length / desired number of pixels*. E.g., if your original unit is meters and you want every pixel of your transformed image to represent 3mm × 3mm of the measuring plane, your scale is calculated Scale = $0.003/1 = 0.003$. If you want to perform a task like shape-based matching on your transformed image, it is useful to scale the image such that its content appears in a size similar to the original image.

Restriction: Scale > 0.

The mapping function is stored in the output image Map. Map has the same size as the resulting images after the mapping. MapType is used to specify the type of the output Map. If *'nearest_neighbor'* is chosen, Map consists of one image containing one channel, in which for each pixel of the resulting image the linearized coordinate of the pixel of the input image is stored that is the nearest neighbor to the transformed coordinates. If *'bilinear'* interpolation is chosen, Map consists of one image containing five channels. In the first channel for each pixel in the resulting image the linearized coordinates of the pixel in the input image is stored that is in the upper left position relative to the transformed coordinates. The four other channels contain the weights of the four neighboring pixels of the transformed coordinates which are used for the bilinear interpolation, in the following order:

2	3
4	5

The second channel, for example, contains the weights of the pixels that lie to the upper left relative to the transformed coordinates. If *'coord_map_sub_pix'* is chosen, Map consists of one vector field image of the semantic type 'vector_field_absolute', in which for each pixel of the resulting image the subpixel precise coordinates in the input image are stored.

If several images have to be mapped using the same camera parameters, gen_image_to_world_plane_map in combination with map_image is much more efficient than the operator image_to_world_plane because the mapping function needs to be computed only once.

If you want to re-use the created map in another program, you can save it as a multi-channel image with the operator `write_image`, using the format *'tiff'*.

Parameters

▷ **Map** (output_object) . . . . . (multichannel-)image ⇝ *object :* int4 / int8 / uint2 / vector_field
Image containing the mapping data.

▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.

▷ **WorldPose** (input_control) . . . . . pose ⇝ *real* / integer
3D pose of the world coordinate system in camera coordinates.
Number of elements: 7

▷ **WidthIn** (input_control) . . . . . extent.x ⇝ *integer*
Width of the images to be transformed.
Restriction: WidthIn >= 1

▷ **HeightIn** (input_control) . . . . . extent.y ⇝ *integer*
Height of the images to be transformed.
Restriction: HeightIn >= 1

▷ **WidthMapped** (input_control) . . . . . extent.x ⇝ *integer*
Width of the resulting mapped images in pixels.
Restriction: WidthMapped >= 1

▷ **HeightMapped** (input_control) . . . . . extent.y ⇝ *integer*
Height of the resulting mapped images in pixels.
Restriction: HeightMapped >= 1

▷ **Scale** (input_control) . . . . . number ⇝ *string* / integer / real
Scale or unit.
Default: 'm'
Suggested values: `Scale` ∈ {'m', 'cm', 'mm', 'microns', 'um', 1.0, 0.01, 0.001, 1.0e-6, 0.0254, 0.3048, 0.9144}
Restriction: Scale > 0

▷ **MapType** (input_control) . . . . . string ⇝ *string*
Type of the mapping.
Default: 'bilinear'
List of values: `MapType` ∈ {'nearest_neighbor', 'bilinear', 'coord_map_sub_pix'}

Example

```
* Calibrate camera.
calibrate_cameras (CalibDataID, Error)
* Obtain camera parameters.
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
* Example values, if no calibration data is available:
CamParam := ['area_scan_division', 0.0087, -1859, 8.65e-006, 8.6e-006, \
             362.5, 291.6, 768, 576]
* Get reference pose (pose 4 of calibration object 0).
get_calib_data (CalibDataID, 'calib_obj_pose',\
                [0,4], 'pose', Pose)
* Example values, if no calibration data is available:
Pose := [-0.11, -0.21, 2.51, 352.73, 346.73, 336.48, 0]
* Compensate thickness of plate.
set_origin_pose (Pose, -1.125, -1.0, 0, PoseNewOrigin)
* Transform the image into the world plane.
read_image (Image, 'calib/calib-3d-coord-04')
gen_image_to_world_plane_map (MapSingle, CamParam, PoseNewOrigin,\
                  CamParam[6], CamParam[7], 900, 800, 0.0025, 'bilinear')
map_image (Image, MapSingle, ImageMapped)
```

Result

`gen_image_to_world_plane_map` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_pose, hom_mat3d_to_pose, camera_calibration, hand_eye_calibration, set_origin_pose

Possible Successors

map_image

Alternatives

image_to_world_plane

See also

map_image, contour_to_world_plane_xld, image_points_to_world_plane

Module

Calibration

gen_radial_distortion_map (: Map : CamParamIn, CamParamOut, MapType :)

Generate a projection map that describes the mapping of images corresponding to a changing radial distortion.

gen_radial_distortion_map computes the mapping of images corresponding to a changing radial distortion in accordance to the internal camera parameters CamParamIn and CamParamOut which can be obtained, e.g., using the operator calibrate_cameras. CamParamIn and CamParamOut contain the old and the new camera parameters including the old and the new radial distortion, respectively (also see Calibration for the sequence of the parameters and the underlying camera model). Each pixel of the potential output image is transformed into the image plane using CamParamOut and subsequently projected into a subpixel position of the potential input image using CamParamIn. Note that gen_radial_distortion_map can only be used with area scan cameras.

The mapping function is stored in the output image Map. The size of Map is given by the camera parameters CamParamOut and therefore defines the size of the resulting mapped images using map_image. The size of the images to be mapped with map_image is determined by the camera parameters CamParamIn. MapType is used to specify the type of the output Map. If *'nearest_neighbor'* is chosen, Map consists of one image containing one channel, in which for each pixel of the resulting image the linearized coordinate of the pixel of the input image is stored that is the nearest neighbor to the transformed coordinates. If *'bilinear'* interpolation is chosen, Map consists of one image containing five channels. In the first channel for each pixel in the resulting image the linearized coordinates of the pixel in the input image is stored that is in the upper left position relative to the transformed coordinates. The four other channels contain the weights of the four neighboring pixels of the transformed coordinates which are used for the bilinear interpolation, in the following order:

2	3
4	5

The second channel, for example, contains the weights of the pixels that lie to the upper left relative to the transformed coordinates. If *'coord_map_sub_pix'* is chosen, Map consists of one vector field image of the semantic type 'vector_field_absolute', in which for each pixel of the resulting image the subpixel precise coordinates in the input image are stored.

If CamParamOut was computed via change_radial_distortion_cam_par, the mapping describes the effect of a lens with a modified radial distortion κ. If $\kappa = 0$, the mapping corresponds to a rectification. A subsequent pose estimation (determination of the external camera parameters) is not affected by this operation.

If several images have to be mapped using the same camera parameters, gen_radial_distortion_map in combination with map_image is much more efficient than the operator change_radial_distortion_image because the transformation must be computed only once.

If you want to re-use the created map in another program, you can save it as a multi-channel image with the operator write_image, using the format *'tiff'*.

Parameters

▷ **Map** (output_object) . . . . . (multichannel-)image ⇝ *object :* int4 / int8 / uint2 / vector_field
Image containing the mapping data.

▷ **CamParamIn** (input_control) . . . . . campar ⇝ *real* / integer / string
Old camera parameters.

▷ **CamParamOut** (input_control) . . . . . campar ⇝ *real* / integer / string
New camera parameters.

▷ **MapType** (input_control) . . . . . string ⇝ *string*
Type of the mapping.
Default: 'bilinear'
List of values: MapType ∈ {'nearest_neighbor', 'bilinear', 'coord_map_sub_pix'}

Result

gen_radial_distortion_map returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

change_radial_distortion_cam_par, camera_calibration, hand_eye_calibration

Possible Successors

map_image

Alternatives

change_radial_distortion_image

See also

change_radial_distortion_contours_xld

Module

Calibration

image_points_to_world_plane (: : CameraParam, WorldPose, Rows, Cols, Scale : X, Y)

Transform image points into the plane z=0 of a world coordinate system.

The operator image_points_to_world_plane transforms image points which are given in Rows and Cols into the plane z=0 in a world coordinate system and returns their 3D coordinates in X and Y. The world coordinate system is chosen by passing its pose relative to the camera coordinate system in WorldPose. Hence, latter one is expected in the form ${}^{ccs}\mathbf{P}_{wcs}$, where *ccs* denotes the camera coordinate system and *wcs* the world coordinate system (see Transformations / Poses and "Solution Guide III-C - 3D Vision"). In CameraParam you must pass the internal camera parameters (see Calibration for the sequence of the parameters and the underlying camera model).

In many cases CameraParam and WorldPose are the result of calibrating the camera with the operator calibrate_cameras. See below for an example.

With the parameter Scale you can scale the resulting 3D coordinates. The parameter Scale must be specified as the ratio *desired unit/original unit*. The original unit is determined by the coordinates of the calibration object. If the original unit is meters (which is the case if you use the standard calibration plate), you can set the desired unit directly by selecting *'m'*, *'cm'*, *'mm'* or *'um'* for the parameter Scale.

Internally, the operator first computes the line of sight between the projection center and the image contour points in the camera coordinate system, taking into account the radial distortions. The line of sight is then transformed

into the world coordinate system specified in WorldPose. By intersecting the plane z=0 with the line of sight the 3D coordinates X and Y are obtained.

It is recommended to use only those image points Rows and Cols, that lie within the calibrated image size. The mathematical model does only work well for image points, that lie within the calibration range.

Parameters

- ▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
  Internal camera parameters.
- ▷ **WorldPose** (input_control) . . . . . pose ⇝ *real* / integer
  3D pose of the world coordinate system in camera coordinates.
  Number of elements: 7
- ▷ **Rows** (input_control) . . . . . coordinates.y-array ⇝ *real* / integer
  Row coordinates of the points to be transformed.
  Default: 100.0
- ▷ **Cols** (input_control) . . . . . coordinates.x-array ⇝ *real* / integer
  Column coordinates of the points to be transformed.
  Default: 100.0
- ▷ **Scale** (input_control) . . . . . number ⇝ *string* / integer / real
  Scale or dimension
  Default: 'm'
  Suggested values: Scale ∈ {'m', 'cm', 'mm', 'microns', 'um', 1.0, 0.01, 0.001, 1.0e-6, 0.0254, 0.3048, 0.9144}
  Restriction: Scale > 0
- ▷ **X** (output_control) . . . . . coordinates.x-array ⇝ *real*
  X coordinates of the points in the world coordinate system.
- ▷ **Y** (output_control) . . . . . coordinates.y-array ⇝ *real*
  Y coordinates of the points in the world coordinate system.

Example

```
*  Perform camera calibration (with standard calibration plate).
calibrate_cameras (CalibDataID, Error)
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
* Get reference pose (pose 2 of calibration object 0).
get_calib_data (CalibDataID, 'calib_obj_pose',\
                [0,2], 'pose', WorldPose)
*  Compensate thickness of plate.
set_origin_pose(ObjInCameraPose, 0, 0, 0.0006, WorldPose)
*  Transform image points into world coordinate system (unit mm).
image_points_to_world_plane(CamParam, WorldPose, PointRows, PointColumns, \
                            'mm', PointXCoord, PointYCoord)
```

Result

image_points_to_world_plane returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_pose, hom_mat3d_to_pose, camera_calibration, hand_eye_calibration, set_origin_pose

See also

contour_to_world_plane_xld, project_3d_point

Module

Calibration

image_to_world_plane (Image : ImageWorld : CameraParam, WorldPose, Width, Height, Scale, Interpolation :)

Rectify an image by transforming it into the plane z=0 of a world coordinate system.

image_to_world_plane rectifies an image Image by transforming it into the plane z=0 (plane of measurements) in a world coordinate system. The resulting rectified image ImageWorld shows neither radial nor perspective distortions; it corresponds to an image acquired by a distortion-free camera that looks perpendicularly onto the plane of measurements. The world coordinate system is chosen by passing its 3D pose relative to the camera coordinate system in WorldPose. Hence, latter one is expected in the form ${}^{ccs}\mathbf{P}_{wcs}$, where *ccs* denotes the camera coordinate system and *wcs* the world coordinate system (see Transformations / Poses and "Solution Guide III-C - 3D Vision"). In CameraParam you must pass the internal camera parameters (see Calibration for the sequence of the parameters and the underlying camera model).

In many cases CameraParam and WorldPose are the result of calibrating the camera with the operator calibrate_cameras. See below for an example.

The pixel position of the upper left corner of the output image ImageWorld is determined by the origin of the world coordinate system. The size of the output image ImageWorld can be chosen by the parameters Width, Height, and Scale. Width and Height must be given in pixels.

The parameter Scale can be used to specify the size of a pixel in the transformed image. There are two ways to use this parameter:

Scale pixels to metric units:

Scale the image such that one pixel in the transformed image corresponds to a metric unit, e.g., setting *'mm'* determines that a pixel in the transformed image corresponds to the area 1mm $\times$ 1mm in the plane of measurements. For this, the original unit needs to be meters. This is the case if you use a standard calibration plate.

List of values: *'m'*, *'cm'*, *'mm'*, *'microns'*, *'um'*.

Default: *'m'*.

Control scaling manually:

Scale the image by giving a number that determines the ratio of *original unit length / desired number of pixels*. E.g., if your original unit is meters and you want every pixel of your transformed image to represent 3mm $\times$ 3mm of the measuring plane, your scale is calculated Scale $= 0.003/1 = 0.003$. If you want to perform a task like shape-based matching on your transformed image, it is useful to scale the image such that its content appears in a size similar to the original image.

Restriction: Scale > *0*.

The parameter Interpolation specifies, whether bilinear interpolation (*'bilinear'*) should be applied between the pixels in the input image or whether the gray value of the nearest neighboring pixel (*'nearest_neighbor'*) should be used.

If several images have to be rectified using the same parameters, gen_image_to_world_plane_map in combination with map_image is much more efficient than the operator image_to_world_plane because the mapping function needs to be computed only once.

Attention

image_to_world_plane can be executed on OpenCL devices if the input image does not exceed the maximum size of image objects of the selected device. There can be slight differences in the output compared to the execution on the CPU.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **ImageWorld** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Transformed image.
- ▷ **CameraParam** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameters.
- ▷ **WorldPose** (input_control) ..... pose ⇝ *real* / integer
  3D pose of the world coordinate system in camera coordinates.
  Number of elements: 7

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the resulting image in pixels.
Restriction: Width >= 1

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the resulting image in pixels.
Restriction: Height >= 1

▷ **Scale** (input_control) . . . . . number ⇝ *string* / integer / real
Scale or unit.
Default: 'm'
Suggested values: Scale ∈ {'m', 'cm', 'mm', 'microns', 'um', 1.0, 0.01, 0.001, 1.0e-6, 0.0254, 0.3048, 0.9144}
Restriction: Scale > 0

▷ **Interpolation** (input_control) . . . . . string ⇝ *string*
Type of interpolation.
Default: 'bilinear'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear'}

Example

```
* Calibrate camera.
calibrate_cameras (CalibDataID, Error)
* Obtain camera parameters.
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
* Example values, if no calibration data is available:
CamParam := ['area_scan_division', 0.0087, -1859, 8.65e-006, 8.6e-006, \
             362.5, 291.6, 768, 576]
* Get reference pose (pose 4 of calibration object 0).
get_calib_data (CalibDataID, 'calib_obj_pose',\
                [0,4], 'pose', Pose)
* Example values, if no calibration data is available:
Pose := [-0.11, -0.21, 2.51, 352.73, 346.73, 336.48, 0]
* Compensate thickness of plate.
set_origin_pose (Pose, -1.125, -1.0, 0, PoseNewOrigin)
* Transform the image into the world plane.
read_image (Image, 'calib/calib-3d-coord-04')
image_to_world_plane (Image, ImageWorld, CamParam, PoseNewOrigin,\
                      900, 800, 0.0025, 'bilinear')
```

Result

image_to_world_plane returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

create_pose, hom_mat3d_to_pose, camera_calibration, hand_eye_calibration, set_origin_pose

Alternatives

gen_image_to_world_plane_map, map_image

See also

contour_to_world_plane_xld, image_points_to_world_plane

Module

Calibration

6.10 Self-Calibration

```
radial_distortion_self_calibration (
    Contours : SelectedContours : Width, Height, InlierThreshold,
    RandSeed, DistortionModel, DistortionCenter,
    PrincipalPointVar : CameraParam )
```

Calibrate the radial distortion.

radial_distortion_self_calibration estimates the distortion parameters and the distortion center of a lens from a set of XLD Contours.

The distortion parameters are returned in CameraParam. Because no other parameters are estimated - particularly not the focal length or the magnification - a telecentric camera model is returned with Magnification 1 and scale factor 1 for S_x and S_y. See Calibration for more information on the different camera models.

Application

Based on the result of radial_distortion_self_calibration, you can remove lens distortions from images by passing the parameter CameraParam, which contains the distortion parameters, to the operators change_radial_distortion_cam_par and change_radial_distortion_image.

Basic principle

The estimation of the distortions is based on the assumption that a significant number of straight lines are visible in the image. Because of lens distortions, these lines will be projected to curved contours. The operator now determines suitable parameters by which the curved contours can be straightened again, thus compensating the lens distortions.

Extract input contours

To get suitable input contours Contours, you can, e.g., use edges_sub_pix or lines_gauss. The contours should be equally distributed and should lie near the image border because there the degree of distortion is at its maximum and therefore the calibration is most stable. To improve speed and robustness, you can try to to obtain long linear or circular segments, e.g., with segment_contours_xld, union_collinear_contours_xld, union_cocircular_contours_xld, or select_shape_xld. If a single image does not contain enough straight contours in the scene, you can use the contours of multiple images (concat_obj).

Set parameters for contour selection

The operator automatically estimates those contours from Contours that are images of straight lines in the scene using the robust RANSAC method. The contours that do not fulfill this condition and hence are not suited for the calibration process are called outliers. The operator can cope with a maximum outlier percentage of 50 percent. A contour is classified as an outlier if the mean deviation of the contour from its associated straight line is, after the distortion correction, higher than a given threshold T.

$$\frac{1}{m}\sum_{j=1}^{m}|d_j| > \texttt{InlierThreshold} \cdot \frac{m}{100} = T$$

The value InlierThreshold describes the mean deviation of a contour from its associated line in pixels for a contour that contains 100 points. The actual threshold T is derived from InlierThreshold by scaling it with the reference length (100) and the number of contour points m. Therefore, similar contours are classified alike. Typical values of InlierThreshold range from 0.05 to 0.5. The higher the value, the more deviation is tolerated. By choosing the value 0, all the contours of Contours are used for the calibration process. The RANSAC contour selection will then be suppressed to enable a manual contour selection. This can be helpful if the outlier percentage is higher than 50 percent.

With the parameter RandSeed, you can control the randomized behavior of the RANSAC algorithm and force it to return reproducible results. The parameter is passed as initial value to the internally used random number generator. If it is set to a positive value, the operator returns identical results for each call with identical parameter values. The value set for the HALCON system variable *'seed_rand'* (see set_system) does not affect the results of radial_distortion_self_calibration.

radial_distortion_self_calibration returns the contours that were chosen for the calibration process in SelectedContours.

Select distortion model

The distortion model used in the calibration can be selected with the parameter DistortionModel. By choosing the division model (DistortionModel = *'division'*), the distortions are modeled by the distortion parameter κ. By choosing the polynomial model (DistortionModel = *'polynomial'*), the distortions are modeled by the radial distortion parameters K_1, K_2, K_3 and the decentering distortion parameters P_1, P_2. See Calibration for details on the different camera models.

Set parameters for the distortion center estimation

The starting value for the estimation of the distortion center $c = (c_x, c_y)$ is the center of the image; the image size is defined by Width and Height.

The distortion parameters (κ, c_x, c_y) or $(K_1, K_2, K_3, P_1, P_2, c_x, c_y)$, respectively, are estimated via the methods *'variable'*, *'adaptive'*, or *'fixed'*, which are specified via the parameter DistortionCenter:

'variable' In the default mode *'variable'*, the distortion center c is estimated with all the other calibration parameters at the same time. Here, many contours should lie equally distributed near the image borders or the distortion should be high. Otherwise, the search for the distortion center could be ill-posed, which results in instability.

'adaptive' With the method *'adaptive'*, the distortion center c is at first fixed in the image center. Then, the outliers are eliminated by using the InlierThreshold. Finally, the calibration process is rerun by estimating (κ, c_x, c_y) or $(K_1, K_2, K_3, P_1, P_2, c_x, c_y)$, respectively, which will be accepted if $c = (c_x, c_y)$ results from a stable calibration and lies near the image center. Otherwise, c will be assumed to lie in the image center. This method should be used if the distortion center can be assumed to lie near the image center and if very few contours are available or the position of other contours is bad (e.g., the contours have the same direction or lie in the same image region).

'fixed' By choosing the method *'fixed'*, the distortion center will be assumed fixed in the image center and only κ or $(K_1, K_2, K_3, P_1, P_2)$, respectively, will be estimated. This method should be used in case of very weak distortions or few contours in bad position.

In order to control the deviation of c from the image center, the parameter PrincipalPointVar can be used in the methods *'adaptive'* and *'variable'*. If the deviation from the image center should be controlled, PrincipalPointVar must lie between 1 and 100. The higher the value, the more the distortion center can deviate from the image center. By choosing the value 0, the principal point is not controlled, i.e., the principal point is determined solely based on the contours. The parameter PrincipalPointVar should be used in cases of weak distortions or similarly oriented contours. Otherwise, a stable solution cannot be guaranteed.

Runtime

The runtime of radial_distortion_self_calibration is shortest for DistortionCenter = *'variable'* and PrincipalPointVar = *0*. The runtime for DistortionCenter = *'variable'* and PrincipalPointVar > *0* increases significantly for smaller values of PrincipalPointVar. The runtimes for DistortionCenter = *'adaptive'* and DistortionCenter = *'fixed'* are also significantly higher than for DistortionCenter = *'variable'* and PrincipalPointVar = *0*.

Attention

Since the polynomial model (DistortionModel = *'polynomial'*) uses more parameters than the division model (DistortionModel = *'division'*) the calibration using the polynomial model can be slightly less stable than the calibration using the division model, which becomes noticeable in the accuracy of the decentering distortion parameters P_1, P_2. To improve the stability, contours of multiple images can be used. Additional stability can be achieved by setting DistortionCenter = *'fixed'*, DistortionCenter = *'adaptive'*, or PrincipalPointVar > *0*, which was already mentioned above.

Parameters

▷ **Contours** (input_object) ..... xld_cont-array ⇝ *object*
Contours that are available for the calibration.

▷ **SelectedContours** (output_object) ..... xld_cont-array ⇝ *object*
Contours that were used for the calibration

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the images from which the contours were extracted.
Default: 640
Suggested values: Width ∈ {640, 768}
Restriction: Width > 0

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the images from which the contours were extracted.
Default: 480
Suggested values: Height ∈ {480, 576}
Restriction: Height > 0

▷ **InlierThreshold** (input_control) . . . . . real ⇝ *real*
Threshold for the classification of outliers.
Default: 0.05
Suggested values: InlierThreshold ∈ {0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1}
Restriction: InlierThreshold >= 0

▷ **RandSeed** (input_control) . . . . . integer ⇝ *integer*
Seed value for the random number generator.
Default: 42

▷ **DistortionModel** (input_control) . . . . . string ⇝ *string*
Determines the distortion model.
Default: 'division'
List of values: DistortionModel ∈ {'division', 'polynomial'}

▷ **DistortionCenter** (input_control) . . . . . string ⇝ *string*
Determines how the distortion center will be estimated.
Default: 'variable'
List of values: DistortionCenter ∈ {'fixed', 'adaptive', 'variable'}

▷ **PrincipalPointVar** (input_control) . . . . . real ⇝ *real*
Controls the deviation of the distortion center from the image center; larger values allow larger deviations from the image center; 0 switches the penalty term off.
Default: 0.0
Suggested values: PrincipalPointVar ∈ {0.0, 5.0, 10.0, 20.0, 50.0, 100.0}
Restriction: PrincipalPointVar >= 0.0 && PrincipalPointVar <= 100.0

▷ **CameraParam** (output_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters.

Example

```
* Assume that GrayImage is one image in gray values with a
* resolution of 640 x 480 and a suitable number of contours.  Then
* the following example performs the calibration using these
* contours and corrects the image with the estimated distortion
* parameters.
edges_sub_pix (GrayImage, Edges, 'canny', 1.0, 20, 40)
segment_contours_xld (Edges, ContoursSplit, 'lines_circles', 5, 8, 4)
radial_distortion_self_calibration (ContoursSplit, SelectedContours, \
                                    640, 480, 0.08, 42, 'division', \
                                    'variable', 0, CameraParam)
get_domain (GrayImage, Domain)
change_radial_distortion_cam_par ('fullsize', CameraParam, 0, CamParamOut)
change_radial_distortion_image (GrayImage, Domain, ImageRectified, \
                                CameraParam, CamParamOut)
```

Result

If the parameters are valid, the operator radial_distortion_self_calibration returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

edges_sub_pix, segment_contours_xld

Possible Successors

change_radial_distortion_cam_par, change_radial_distortion_image

See also

camera_calibration

References

T. Thormählen, H. Broszio: "Automatic line-based estimation of radial lens distortion"; in: Integrated Computer-Aided Engineering; vol. 12; pp. 177-190; 2005.

Module

Calibration

radiometric_self_calibration (Images : : ExposureRatios, Features, FunctionType, Smoothness, PolynomialDegree : InverseResponse)

Perform a radiometric self-calibration of a camera.

radiometric_self_calibration performs a radiometric self-calibration of a camera. For this, at least two images that show the same image contents (scene) must be passed in Images. All images passed in Images must be acquired with different exposures. Typically, the different exposures are obtained by changing the shutter times at the camera. It is not recommended to change the exposure by changing the aperture of the lens since in this case the exposures cannot be determined accurately enough. The ratio of the exposures of consecutive images is passed in ExposureRatios. For example, a value of *0.5* specifies that the second image of an image pair has been acquired with half the exposure of the first image of the pair. The exposure ratio can easily be determined from the shutter times since the exposure is proportional to the shutter time. The exposure ratio must be greater than 0 and smaller than 1. This means that the images must be sorted according to descending exposure. ExposureRatios must contain one element less than the number of images passed in Images. If all exposure ratios are identical, as a simplification a single value can be passed in ExposureRatios.

As described above, the images passed in Images must show identical image contents. Hence, it is typically necessary that neither the camera nor the objects in the scene move. If the camera has rotated around the optical center, the images should be aligned to a reference image (one of the images) using proj_match_points_ransac and projective_trans_image. If the features used for the radiometric calibration are determined from the 2D gray value histogram of consecutive image pairs (Features $= '2d\_histogram'$), it is essential that the images are aligned and that the objects in the scene do not move. For Features $= '1d\_histograms'$, the features used for the radiometric calibration are determined from the 1D gray value histograms of the image pairs. In this mode, the calibration can theoretically be performed if the 1D histograms of the images do not change by the movement of the objects in the images. This can, for example, be the case if an object moves in front of a uniformly textured background. However, it is preferable to use Features $= '2d\_histogram'$ because this mode is more accurate. The mode Features $= '1d\_histograms'$ should only be used if it is impossible to construct the camera set-up such that neither the camera nor the objects in the scene move.

Furthermore, care should be taken to cover the range of gray values without gaps by choosing appropriate image contents. Whether there are gaps in the range of gray values can easily be checked based on the 1D gray value histograms of the images or the 2D gray value histograms of consecutive images. In the 1D gray value histograms (see gray_histo_abs), there should be no areas between the minimum and maximum gray value that have a frequency of 0 or a very small frequency. In the 2D gray value histograms (see histo_2dim), a single connected region having the shape of a "strip" should result from a threshold operation with a lower threshold of 1. If more than one connected component results, a more suitable image content should be chosen. If the image content can be chosen such that the gray value range of the image (e.g., 0-255 for byte images) can be covered with two images with different exposures, and if there are no gaps in the histograms, the two images suffice for the calibration. This, however, is typically not the case, and hence multiple images must be used to cover the entire gray value range. As described above, for this multiple images with different exposures must be taken to cover the entire gray value range as well as possible. For this, normally the first image should be exposed such that the maximum gray value is slightly below the saturation limit of the camera, or such that the image is significantly overexposed. If the first image is overexposed, a significant overexposure is necessary to enable radiometric_self_calibration to detect the overexposed areas reliably. If the camera exhibits an unusual saturation behavior (e.g., a saturation

limit that lies significantly below the maximum gray value) the overexposed areas should be masked out by hand with reduce_domain in the overexposed image.

radiometric_self_calibration returns the inverse gray value response function of the camera in InverseResponse. The inverse response function can be used to create an image with a linear response by using InverseResponse as the LUT in lut_trans. The parameter FunctionType determines which function model is used to model the response function. For FunctionType $= 'discrete'$, the response function is described by a discrete function with the relevant number of gray values (256 for byte images). For FunctionType $= 'polynomial'$, the response is described by a polynomial of degree PolynomialDegree. The computation of the response function is slower for FunctionType $= 'discrete'$. However, since a polynomial tends to oscillate in the areas in which no gray value information can be derived, even if smoothness constraints are imposed as described below, the discrete model should usually be preferred over the polynomial model.

The inverse response function is returned as a tuple of integer values for FunctionType $= 'discrete'$ and FunctionType $= 'polynomial'$. In some applications, it might be desirable to return the inverse response function as floating point values to avoid the numerical error that is introduced by rounding. For example, if the inverse response function must be inverted to obtain the response function of the camera, there is some loss of information if the values are returned as integers. For these applications, FunctionType can be set to *'discrete_real'* or *'polynomial_real'*, in which case the inverse response function will be returned as a tuple of floating point numbers.

The parameter Smoothness defines (in addition to the constraints on the response function that can be derived from the images) constraints on the smoothness of the response function. If, as described above, the gray value range can be covered completely and without gaps, the default value of *1* should not be changed. Otherwise, values > 1 can be used to obtain a stronger smoothing of the response function, while values < 1 lead to a weaker smoothing. The smoothing is particularly important in areas for which no gray value information can be derived from the images, i.e., in gaps in the histograms and for gray values smaller than the minimum gray value of all images or larger than the maximum gray value of all images. In these areas, the smoothness constraints lead to an interpolation or extrapolation of the response function. Because of the nature of the internally derived constraints, FunctionType $= 'discrete'$ favors an exponential function in the undefined areas, whereas FunctionType $= 'polynomial'$ favors a straight line. Please note that the interpolation and extrapolation is always less reliable than to cover the gray value range completely and without gaps. Therefore, in any case it should be attempted first to acquire the images optimally, before the smoothness constraints are used to fill in the remaining gaps. In all cases, the response function should be checked for plausibility after the call to radiometric_self_calibration. In particular, it should be checked whether InverseResponse is monotonic. If this is not the case, a more suitable scene should be used to avoid interpolation, or Smoothness should be set to a larger value. For FunctionType $= 'polynomial'$, it may also be necessary to change PolynomialDegree. If, despite these changes, an implausible response is returned, the saturation behavior of the camera should be checked, e.g., based on the 2D gray value histogram, and the saturated areas should be masked out by hand, as described above.

When the inverse gray value response function of the camera is determined, the absolute energy falling on the camera cannot be determined. This means that InverseResponse can only be determined up to a scale factor. Therefore, an additional constraint is used to fix the unknown scale factor: the maximum gray value that can occur should occur for the maximum input gray value, e.g., InverseResponse$[255] = 255$ for byte images. This constraint usually leads to the most intuitive results. If, however, a multichannel image (typically an RGB image) should be radiometrically calibrated (for this, each channel must be calibrated separately), the above constraint may lead to the result that a different scaling factor is determined for each channel. This may lead to the result that gray tones no longer appear gray after the correction. In this case, a manual white balancing step must be carried out by identifying a homogeneous gray area in the original image, and by deriving appropriate scaling factors from the corrected gray values for two of the three response curves (or, in general, for $n - 1$ of the n channels). Here, the response curve that remains invariant should be chosen such that all scaling factors are < 1. With the scaling factors thus determined, new response functions should be calculated by multiplying each value of a response function with the scaling factor corresponding to that response function.

Parameters

▷ **Images** (input_object) . . . . . singlechannelimage-array ⇝ *object :* byte / uint2
Input images.

▷ **ExposureRatios** (input_control) ..... real(-array) ⇝ *real*
Ratio of the exposure energies of successive image pairs.
Default: 0.5
Suggested values: ExposureRatios ∈ {0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8}
Restriction: ExposureRatios > 0 && ExposureRatios < 1

▷ **Features** (input_control) ..... string ⇝ *string*
Features that are used to compute the inverse response function of the camera.
Default: '2d_histogram'
List of values: Features ∈ {'2d_histogram', '1d_histograms'}

▷ **FunctionType** (input_control) ..... string ⇝ *string*
Type of the inverse response function of the camera.
Default: 'discrete'
List of values: FunctionType ∈ {'discrete', 'polynomial', 'discrete_real', 'polynomial_real'}

▷ **Smoothness** (input_control) ..... real ⇝ *real*
Smoothness of the inverse response function of the camera.
Default: 1.0
Suggested values: Smoothness ∈ {0.3, 0.5, 0.7, 0.8, 1.0, 1.2, 1.5, 2.0, 3.0}
Restriction: Smoothness > 0

▷ **PolynomialDegree** (input_control) ..... integer ⇝ *integer*
Degree of the polynomial if FunctionType = *'polynomial'*.
Default: 5
Suggested values: PolynomialDegree ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Restriction: PolynomialDegree >= 1 && PolynomialDegree <= 20

▷ **InverseResponse** (output_control) ..... number-array ⇝ *integer* / real
Inverse response function of the camera.

Example

```
open_framegrabber ('1394IIDC', 1, 1, 0, 0, 0, 0, 'default', -1, \
                   'default', -1, 'default', 'default', 'default', \
                   -1, -1, AcqHandle)
* Define appropriate shutter times.
Shutters := [1000,750,500,250,125]
Num := |Shutters|
* Grab and accumulate images with the different exposures.  In this
* loop, it must be ensured that the scene remains static.
gen_empty_obj (Images)
for I := 0 to Num-1 by 1
    set_framegrabber_param (AcqHandle, 'shutter', Shutters[I])
    grab_image (Image, AcqHandle)
    concat_obj (Images, Image, Images)
endfor
* Compute the exposure ratios from the shutter times.
ExposureRatios := real(Shutters[1:Num-1])/real(Shutters[0:Num-2])
radiometric_self_calibration (Images, ExposureRatios, '2d_histogram', \
                              'discrete', 1, 5, InverseResponse)
* Note that if the frame grabber supports hardware LUTs, we could
* also call set_framegrabber_lut here instead of lut_trans below.
* This would be more efficient.
while (1)
    grab_image_async (Image, AcqHandle, -1)
    lut_trans (Image, ImageLinear, InverseResponse)
    * Process radiometrically correct image.
    * [...]
endwhile
close_framegrabber (AcqHandle)
```

Result

If the parameters are valid, the operator radiometric_self_calibration returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, grab_image, grab_image_async, set_framegrabber_param, concat_obj, proj_match_points_ransac, proj_match_points_ransac_guided, projective_trans_image

Possible Successors

lut_trans

See also

histo_2dim, gray_histo, gray_histo_abs, reduce_domain

Module

Calibration

stationary_camera_self_calibration (: : NumImages, ImageWidth, ImageHeight, ReferenceImage, MappingSource, MappingDest, HomMatrices2D, Rows1, Cols1, Rows2, Cols2, NumCorrespondences, EstimationMethod, CameraModel, FixedCameraParams : CameraMatrices, Kappa, RotationMatrices, X, Y, Z, Error)

Perform a self-calibration of a stationary projective camera.

stationary_camera_self_calibration performs a self-calibration of a stationary projective camera. Here, stationary means that the camera may only rotate around the optical center and may zoom. Hence, the optical center may not move. Projective means that the camera model is a pinhole camera that can be described by a projective 3D-2D transformation. In particular, radial distortions can only be modeled for cameras with constant parameters. If the lens exhibits significant radial distortions they should be removed, at least approximately, with change_radial_distortion_image.

The camera model being used can be described as follows:

$$\mathbf{x} = \mathtt{P}\mathbf{X} \ .$$

Here, $\mathbf{x}$ is a homogeneous 2D vector, $\mathbf{X}$ a homogeneous 3D vector, and P a homogeneous 3×4 projection matrix. The projection matrix P can be decomposed as follows:

$$\mathtt{P} = \mathtt{K}(\mathtt{R}|\mathbf{t}) \ .$$

Here, R is a 3×3 rotation matrix and $\mathbf{t}$ is an inhomogeneous 3D vector. These two entities describe the position (pose) of the camera in 3D space. This convention is analogous to the convention used in camera_calibration, i.e., for R = I and $\mathbf{t} = 0$ the x axis points to the right, the y axis downwards, and the z axis points forward. K is the calibration matrix of the camera (the camera matrix) which can be described as follows:

$$\mathtt{K} = \begin{pmatrix} af & sf & u \\ 0 & f & v \\ 0 & 0 & 1 \end{pmatrix} \ .$$

Here, f is the focal length of the camera in pixels, a the aspect ratio of the pixels, s is a factor that models the skew of the image axes, and (u, v) is the principal point of the camera in pixels. In this convention, the x axis corresponds to the column axis and the y axis to the row axis.

Since the camera is stationary, it can be assumed that $\mathbf{t} = 0$. With this convention, it is easy to see that the fourth coordinate of the homogeneous 3D vector $\mathbf{X}$ has no influence on the position of the projected 3D point.

Consequently, the fourth coordinate can be set to 0, and it can be seen that $\mathbf{X}$ can be regarded as a point at infinity, and hence represents a direction in 3D. With this convention, the fourth coordinate of $\mathbf{X}$ can be omitted, and hence $\mathbf{X}$ can be regarded as inhomogeneous 3D vector which can only be determined up to scale since it represents a direction. With this, the above projection equation can be written as follows:

$$\mathbf{x} = \mathtt{K}\mathtt{R}\mathbf{X} \ .$$

If two images of the same point are taken with a stationary camera, the following equations hold:

$$\begin{aligned} \mathbf{x}_1 &= \mathtt{K}_1\mathtt{R}_1\mathbf{X} \\ \mathbf{x}_2 &= \mathtt{K}_2\mathtt{R}_2\mathbf{X} \end{aligned}$$

and consequently

$$\mathbf{x}_2 = \mathtt{K}_2\mathtt{R}_2\mathtt{R}_1^{-1}\mathtt{K}_1^{-1}\mathbf{x}_1 = \mathtt{K}_2\mathtt{R}_{12}\mathtt{K}_1^{-1}\mathbf{x}_1 = \mathtt{H}_{12}\mathbf{x}_1 \ .$$

If the camera parameters do not change when taking the two images, $K_1 = K_2$ holds. Because of the above, the two images of the same 3D point are related by a projective 2D transformation. This transformation can be determined with `proj_match_points_ransac`. It needs to be taken into account that the order of the coordinates of the projective 2D transformations in HALCON is the opposite of the above convention. Furthermore, it needs to be taken into account that `proj_match_points_ransac` uses a coordinate system in which the origin of a pixel lies in the upper left corner of the pixel, whereas `stationary_camera_self_calibration` uses a coordinate system that corresponds to the definition used in `camera_calibration`, in which the origin of a pixel lies in the center of the pixel. For projective 2D transformations that are determined with `proj_match_points_ransac` the rows and columns must be exchanged and a translation of $(0.5, 0.5)$ must be applied. Hence, instead of $\mathtt{H}_{12} = \mathtt{K}_2\mathtt{R}_{12}\mathtt{K}_1^{-1}$ the following equations hold in HALCON:

$$\mathtt{H}_{12} = \begin{pmatrix} 0 & 1 & 0.5 \\ 1 & 0 & 0.5 \\ 0 & 0 & 1 \end{pmatrix} \mathtt{K}_2\mathtt{R}_{12}\mathtt{K}_1^{-1} \begin{pmatrix} 0 & 1 & -0.5 \\ 1 & 0 & -0.5 \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$\mathtt{K}_2\mathtt{R}_{12}\mathtt{K}_1^{-1} = \begin{pmatrix} 0 & 1 & -0.5 \\ 1 & 0 & -0.5 \\ 0 & 0 & 1 \end{pmatrix} \mathtt{H}_{12} \begin{pmatrix} 0 & 1 & 0.5 \\ 1 & 0 & 0.5 \\ 0 & 0 & 1 \end{pmatrix} \ .$$

From the above equation, constraints on the camera parameters can be derived in two ways. First, the rotation can be eliminated from the above equation, leading to equations that relate the camera matrices with the projective 2D transformation between the two images. Let $\mathtt{H}_{ij}$ be the projective transformation from image i to image j. Then,

$$\begin{aligned} \mathtt{K}_j\mathtt{K}_j^\top &= \mathtt{H}_{ij}\mathtt{K}_i\mathtt{K}_i^\top\mathtt{H}_{ij}^\top \\ \mathtt{K}_j^{-\top}\mathtt{K}_j^{-1} &= \mathtt{H}_{ij}^{-\top}\mathtt{K}_i^{-\top}\mathtt{K}_i^{-1}\mathtt{H}_{ij}^{-1} \end{aligned}$$

From the second equation, linear constraints on the camera parameters can be derived. This method is used for `EstimationMethod` = *'linear'*. Here, all source images i given by `MappingSource` and all destination images j given by `MappingDest` are used to compute constraints on the camera parameters. After the camera parameters have been determined from these constraints, the rotation of the camera in the respective images can be determined based on the equation $\mathtt{R}_{ij} = \mathtt{K}_j^{-1}\mathtt{H}_{ij}\mathtt{K}_i$ and by constructing a chain of transformations from the reference image `ReferenceImage` to the respective image. From the first equation above, a nonlinear method to determine the camera parameters can be derived by minimizing the following error:

$$E = \sum_{(i,j)\in\{(s,d)\}} \left\| \mathtt{K}_j\mathtt{K}_j^\top - \mathtt{H}_{ij}\mathtt{K}_i\mathtt{K}_i^\top\mathtt{H}_{ij}^\top \right\|_F^2$$

Here, analogously to the linear method, $\{(s,d)\}$ is the set of overlapping images specified by MappingSource and MappingDest. This method is used for EstimationMethod = *'nonlinear'*. To start the minimization, the camera parameters are initialized with the results of the linear method. These two methods are very fast and return acceptable results if the projective 2D transformations H_{ij} are sufficiently accurate. For this, it is essential that the images do not have radial distortions. It can also be seen that in the above two methods the camera parameters are determined independently from the rotation parameters, and consequently the possible constraints are not fully exploited. In particular, it can be seen that it is not enforced that the projections of the same 3D point lie close to each other in all images. Therefore, stationary_camera_self_calibration offers a complete bundle adjustment as a third method (EstimationMethod = *'gold_standard'*). Here, the camera parameters and rotations as well as the directions in 3D corresponding to the image points (denoted by the vectors **X** above), are determined in a single optimization by minimizing the following error:

$$E = \sum_{i=1}^{n} \left(\sum_{j=1}^{m} \| \mathbf{x}_{ij} - \mathrm{K}_i \mathrm{R}_i \mathbf{X}_j \|^2 + \frac{1}{\sigma^2}(u_i^2 + v_i^2) \right)$$

In this equation, only the terms for which the reconstructed direction $\mathbf{X}_j$ is visible in image i are taken into account. The starting values for the parameters in the bundle adjustment are derived from the results of the nonlinear method. Because of the high complexity of the minimization the bundle adjustment requires a significantly longer execution time than the two simpler methods. Nevertheless, because the bundle adjustment results in significantly better results, it should be preferred.

In each of the three methods the camera parameters that should be computed can be specified. The remaining parameters are set to a constant value. Which parameters should be computed is determined with the parameter CameraModel which contains a tuple of values. CameraModel must always contain the value *'focus'* that specifies that the focal length f is computed. If CameraModel contains the value *'principal_point'* the principal point (u, v) of the camera is computed. If not, the principal point is set to (ImageWidth/2, ImageHeight/2). If CameraModel contains the value *'aspect'* the aspect ratio a of the pixels is determined, otherwise it is set to 1. If CameraModel contains the value *'skew'* the skew of the image axes is determined, otherwise it is set to 0. Only the following combinations of the parameters are allowed: *'focus'*, *['focus', 'principal_point']*, *['focus', 'aspect']*, *['focus', 'principal_point', 'aspect']*, and *['focus', 'principal_point', 'aspect', 'skew']*.

Additionally, it is possible to determine the parameter Kappa, which models radial lens distortions, if EstimationMethod = *'gold_standard'* has been selected. In this case, *'kappa'* can also be included in the parameter CameraModel. Kappa corresponds to the radial distortion parameter κ of the division model for lens distortions (see camera_calibration).

When using EstimationMethod = *'gold_standard'* to determine the principal point, it is possible to penalize estimations far away from the image center. This can be done by adding a sigma to the parameter *'principal_point: 0.5'*. If no sigma is given the penalty term in the above equation for calculating the error is omitted.

The parameter FixedCameraParams determines whether the camera parameters can change in each image or whether they should be assumed constant for all images. To calibrate a camera so that it can later be used for measuring with the calibrated camera, only FixedCameraParams = *'true'* is useful. The mode FixedCameraParams = *'false'* is mainly useful to compute spherical mosaics with gen_spherical_mosaic if the camera zoomed or if the focus changed significantly when the mosaic images were taken. If a mosaic with constant camera parameters should be computed, of course FixedCameraParams = *'true'* should be used. It should be noted that for FixedCameraParams = *'false'* the camera calibration problem is determined very badly, especially for long focal lengths. In these cases, often only the focal length can be determined. Therefore, it may be necessary to use CameraModel = *'focus'* or to constrain the position of the principal point by using a small Sigma for the penalty term for the principal point.

The number of images that are used for the calibration is passed in NumImages. Based on the number of images, several constraints for the camera model must be observed. If only two images are used, even under the assumption of constant parameters not all camera parameters can be determined. In this case, the skew of the image axes should be set to 0 by *not* adding *'skew'* to CameraModel. If FixedCameraParams = *'false'* is used, the full set of camera parameters can never be determined, no matter how many images are used. In this case, the skew should be set to 0 as well. Furthermore, it should be noted that the aspect ratio can only be determined accurately if at least one image is rotated around the optical axis (the z axis of the camera coordinate system) with respect to the other images. If this is not the case the computation of the aspect ratio should be suppressed by *not* adding *'aspect'* to CameraModel.

As described above, to calibrate the camera it is necessary that the projective transformation for each overlapping image pair is determined with proj_match_points_ransac. For example, for a 2×2 block of images in the following layout

1	2
3	4

the following projective transformations should be determined, assuming that all images overlap each other: 1↦2, 1↦3, 1↦4, 2↦3, 2↦4 and 3↦4. The indices of the images that determine the respective transformation are given by MappingSource and MappingDest. The indices are start at 1. Consequently, in the above example MappingSource = *[1,1,1,2,2,3]* and MappingDest = *[2,3,4,3,4,4]* must be used. The number of images in the mosaic is given by NumImages. It is used to check whether each image can be reached by a chain of transformations. The index of the reference image is given by ReferenceImage. On output, this image has the identity matrix as its transformation matrix.

The 3×3 projective transformation matrices that correspond to the image pairs are passed in HomMatrices2D. Additionally, the coordinates of the matched point pairs in the image pairs must be passed in Rows1, Cols1, Rows2, and Cols2. They can be determined from the output of proj_match_points_ransac with tuple_select or with the HDevelop function subset. To enable stationary_camera_self_calibration to determine which point pair belongs to which image pair, NumCorrespondences must contain the number of found point matches for each image pair.

The computed camera matrices K_i are returned in CameraMatrices as 3×3 matrices. For FixedCameraParams = *'false'*, NumImages matrices are returned. Since for FixedCameraParams = *'true'* all camera matrices are identical, a single camera matrix is returned in this case. The computed rotations R_i are returned in RotationMatrices as 3×3 matrices. RotationMatrices always contains NumImages matrices.

If EstimationMethod = *'gold_standard'* is used, (X, Y, Z) contains the reconstructed directions $\mathbf{X}_j$. In addition, Error contains the average projection error of the reconstructed directions. This can be used to check whether the optimization has converged to useful values.

If the computed camera parameters are used to project 3D points or 3D directions into the image i the respective camera matrix should be multiplied with the corresponding rotation matrix (with hom_mat2d_compose).

Parameters

- ▷ **NumImages** (input_control) . . . integer ⇝ *integer*
  Number of different images that are used for the calibration.
  Restriction: NumImages >= 2
- ▷ **ImageWidth** (input_control) . . . extent.x ⇝ *integer*
  Width of the images from which the points were extracted.
  Restriction: ImageWidth > 0
- ▷ **ImageHeight** (input_control) . . . extent.y ⇝ *integer*
  Height of the images from which the points were extracted.
  Restriction: ImageHeight > 0
- ▷ **ReferenceImage** (input_control) . . . integer ⇝ *integer*
  Index of the reference image.
- ▷ **MappingSource** (input_control) . . . integer-array ⇝ *integer*
  Indices of the source images of the transformations.
- ▷ **MappingDest** (input_control) . . . integer-array ⇝ *integer*
  Indices of the target images of the transformations.
- ▷ **HomMatrices2D** (input_control) . . . hom_mat2d-array ⇝ *real*
  Array of 3×3 projective transformation matrices.
- ▷ **Rows1** (input_control) . . . point.y-array ⇝ *real* / integer
  Row coordinates of corresponding points in the respective source images.
- ▷ **Cols1** (input_control) . . . point.x-array ⇝ *real* / integer
  Column coordinates of corresponding points in the respective source images.
- ▷ **Rows2** (input_control) . . . point.y-array ⇝ *real* / integer
  Row coordinates of corresponding points in the respective destination images.
- ▷ **Cols2** (input_control) . . . point.x-array ⇝ *real* / integer
  Column coordinates of corresponding points in the respective destination images.
- ▷ **NumCorrespondences** (input_control) . . . integer-array ⇝ *integer*
  Number of point correspondences in the respective image pair.

▷ **EstimationMethod** (input_control) . . . . . string ⇝ *string*
Estimation algorithm for the calibration.
Default: 'gold_standard'
List of values: EstimationMethod ∈ {'linear', 'nonlinear', 'gold_standard'}

▷ **CameraModel** (input_control) . . . . . string-array ⇝ *string*
Camera model to be used.
Default: ['focus','principal_point']
List of values: CameraModel ∈ {'focus', 'aspect', 'skew', 'principal_point', 'kappa'}

▷ **FixedCameraParams** (input_control) . . . . . string ⇝ *string*
Are the camera parameters identical for all images?
Default: 'true'
List of values: FixedCameraParams ∈ {'true', 'false'}

▷ **CameraMatrices** (output_control) . . . . . hom_mat2d-array ⇝ *real*
(Array of) 3×3 projective camera matrices that determine the internal camera parameters.

▷ **Kappa** (output_control) . . . . . real(-array) ⇝ *real*
Radial distortion of the camera.

▷ **RotationMatrices** (output_control) . . . . . hom_mat2d-array ⇝ *real*
Array of 3×3 transformation matrices that determine rotation of the camera in the respective image.

▷ **X** (output_control) . . . . . point3d.x-array ⇝ *real*
X-Component of the direction vector of each point if EstimationMethod = *'gold_standard'* is used.

▷ **Y** (output_control) . . . . . point3d.y-array ⇝ *real*
Y-Component of the direction vector of each point if EstimationMethod = *'gold_standard'* is used.

▷ **Z** (output_control) . . . . . point3d.z-array ⇝ *real*
Z-Component of the direction vector of each point if EstimationMethod = *'gold_standard'* is used.

▷ **Error** (output_control) . . . . . real(-array) ⇝ *real*
Average error per reconstructed point if EstimationMethod = *'gold_standard'* is used.

Example

```
* Assume that Images contains four images in the layout given in the
* above description.  Then the following example performs the camera
* self-calibration using these four images.
From := [1,1,1,2,2,3]
To := [2,3,4,3,4,4]
HomMatrices2D := []
Rows1 := []
Cols1 := []
Rows2 := []
Cols2 := []
NumMatches := []
for J := 0 to |From|-1 by 1
    select_obj (Images, ImageF, From[J])
    select_obj (Images, ImageT, To[J])
    points_foerstner (ImageF, 1, 2, 3, 100, 0.1, 'gauss', 'true', \
                      RowsF, ColsF, _, _, _, _, _, _, _, _)
    points_foerstner (ImageT, 1, 2, 3, 100, 0.1, 'gauss', 'true', \
                      RowsT, ColsT, _, _, _, _, _, _, _, _)
    proj_match_points_ransac (ImageF, ImageT, RowsF, ColsF, RowsT, ColsT, \
                              'ncc', 10, 0, 0, 480, 640, 0, 0.5, \
                              'gold_standard', 2, 42, HomMat2D, \
                              Points1, Points2)
    HomMatrices2D := [HomMatrices2D,HomMat2D]
    Rows1 := [Rows1,subset(RowsF,Points1)]
    Cols1 := [Cols1,subset(ColsF,Points1)]
    Rows2 := [Rows2,subset(RowsT,Points2)]
    Cols2 := [Cols2,subset(ColsT,Points2)]
    NumMatches := [NumMatches,|Points1|]
endfor
```

```
stationary_camera_self_calibration (4, 640, 480, 1, From, To, \
                                    HomMatrices2D, Rows1, Cols1, \
                                    Rows2, Cols2, NumMatches, \
                                    'gold_standard', \
                                    ['focus','principal_point'], \
                                    'true', CameraMatrix, Kappa, \
                                    RotationMatrices, X, Y, Z, Error)
```

Result

If the parameters are valid, the operator `stationary_camera_self_calibration` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`proj_match_points_ransac`, `proj_match_points_ransac_guided`

Possible Successors

`gen_spherical_mosaic`

See also

`gen_projective_mosaic`

References

Lourdes Agapito, E. Hayman, I. Reid: “Self-Calibration of Rotating and Zooming Cameras”; International Journal of Computer Vision; vol. 45, no. 2; pp. 107–127; 2001.

Module

Calibration

Chapter 7

Classification

7.1 Gaussian Mixture Models

add_class_train_data_gmm (: : GMMHandle, ClassTrainDataHandle :)

Add training data to a Gaussian Mixture Model (GMM).

add_class_train_data_gmm adds the training data specified by ClassTrainDataHandle to a Gaussian Mixture Model (GMM) specified by GMMHandle.

Parameters

▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
Handle of a GMM which receives the training data.

▷ **ClassTrainDataHandle** (input_control) ..... class_train_data ⇝ *handle*
Handle of training data for a classifier.

Result

If the parameters are valid, the operator add_class_train_data_gmm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- GMMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_gmm, create_class_train_data

Possible Successors

get_sample_class_gmm

Alternatives

add_sample_class_gmm

See also

create_class_gmm

Module

Foundation

add_sample_class_gmm (: : GMMHandle, Features, ClassID, Randomize :)

Add a training sample to the training data of a Gaussian Mixture Model.

add_sample_class_gmm adds a training sample to the Gaussian Mixture Model (GMM) given by GMMHandle. The training sample is given by Features and ClassID. Features is the feature vector of the sample, and consequently must be a real vector of length NumDim, as specified in create_class_gmm. ClassID is the class of the sample, an integer between 0 and NumClasses-1 (set in create_class_gmm).

In the special case where the feature vectors are of **integer** type, they are lying in the feature space in a grid with step width 1.0. For example, the RGB feature vectors typically used for color classification are triples having integer values between 0 and 255 for each of their components. In fact, there might be even several feature vectors representing the same point. When training a GMM with such data, the training algorithm may tend to align the modeled Gaussians along linearly dependent lines or planes of data that are parallel to the grid dimensions. If the number of Centers returned by train_class_gmm is unusually high, this indicates such a behavior of the algorithm. The parameter Randomize can be used to handle such undesired effects. If Randomize $>$ 0.0, random Gaussian noise with mean 0 and standard deviation Randomize is added to each component of the training data vectors, and the transformed training data is stored in the GMM. For values of Randomize $\leq$ 1.0, the randomized data will look like small clouds around the grid points, which does not improve the properties of the data cloud. For values of Randomize $\gg$ 2.0, the randomization might have a too strong influence on the resulting GMM. For integer feature vectors, a value of Randomize between 1.5 and 2.0 is recommended, which transforms the integer data into homogeneous clouds, without modifying its general form in the feature space. If the data has been created from integer data by scaling, the same problem may occur. Here, Randomize must be scaled with the same scale factor that was used to scale the original data.

Before the GMM can be trained with train_class_gmm, *all* training samples must be added to the GMM with add_sample_class_gmm.

The number of currently stored training samples can be queried with get_sample_num_class_gmm. Stored training samples can be read out again with get_sample_class_gmm.

Normally, it is useful to save the training samples in a file with write_samples_class_gmm to facilitate reusing the samples, and to facilitate that, if necessary, new training samples can be added to the data set, and hence to facilitate that a *newly created* GMM can be trained *anew* with the extended data set.

Parameters

▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
GMM handle.

▷ **Features** (input_control) ..... real-array ⇝ *real*
Feature vector of the training sample to be stored.

▷ **ClassID** (input_control) ..... number ⇝ *integer*
Class of the training sample to be stored.

▷ **Randomize** (input_control) ..... real ⇝ *real*
Standard deviation of the Gaussian noise added to the training data.
Default: 0.0
Suggested values: Randomize ∈ {0.0, 1.5, 2.0}
Restriction: Randomize >= 0.0

Result

If the parameters are valid, the operator add_sample_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- GMMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_gmm

Possible Successors

train_class_gmm, write_samples_class_gmm

Alternatives

read_samples_class_gmm, add_samples_image_class_gmm

See also

clear_samples_class_gmm, get_sample_num_class_gmm, get_sample_class_gmm

Module

Foundation

classify_class_gmm (: : GMMHandle, Features, Num : ClassID, ClassProb, Density, KSigmaProb)

Calculate the class of a feature vector by a Gaussian Mixture Model.

classify_class_gmm computes the best Num classes of the feature vector Features with the Gaussian Mixture Model (GMM) GMMHandle and returns the classes in ClassID and the corresponding probabilities of the classes in ClassProb. Before calling classify_class_gmm, the GMM must be trained with train_class_gmm.

classify_class_gmm corresponds to a call to evaluate_class_gmm and an additional step that extracts the best Num classes. As described with evaluate_class_gmm, the output values of the GMM can be interpreted as probabilities of the occurrence of the respective classes. However, here the posterior probability ClassProb is further normalized as ClassProb $= p(i|x)/p(x)$, where $p(i|x)$ and $p(x)$ are specified with evaluate_class_gmm. In most cases it should be sufficient to use Num $= 1$ in order to decide whether the probability of the best class is high enough. In some applications it may be interesting to also take the second best class into account (Num $= 2$), particularly if it can be expected that the classes show a significant degree of overlap.

Density and KSigmaProb are explained with evaluate_class_gmm.

Parameters

- ▷ **GMMHandle** (input_control) . . . class_gmm ⇝ *handle*
  GMM handle.
- ▷ **Features** (input_control) . . . real-array ⇝ *real*
  Feature vector.
- ▷ **Num** (input_control) . . . integer ⇝ *integer*
  Number of best classes to determine.
  Default: 1
  Suggested values: Num ∈ {1, 2, 3, 4, 5}
- ▷ **ClassID** (output_control) . . . integer(-array) ⇝ *integer*
  Result of classifying the feature vector with the GMM.
- ▷ **ClassProb** (output_control) . . . real-array ⇝ *real*
  A-posteriori probability of the classes.
- ▷ **Density** (output_control) . . . real-array ⇝ *real*
  Probability density of the feature vector.
- ▷ **KSigmaProb** (output_control) . . . real-array ⇝ *real*
  Normalized k-sigma-probability for the feature vector.

Result

If the parameters are valid, the operator classify_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_gmm, read_class_gmm

Alternatives

evaluate_class_gmm

See also

create_class_gmm

References

Christopher M. Bishop: "Neural Networks for Pattern Recognition"; Oxford University Press, Oxford; 1995.

Mario A.T. Figueiredo: "Unsupervised Learning of Finite Mixture Models"; IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 24, No. 3; March 2002.

Module

Foundation

clear_class_gmm (: : GMMHandle :)

Clear a Gaussian Mixture Model.

clear_class_gmm clears the Gaussian Mixture Model (GMM) given by GMMHandle and frees all memory required for the GMM. After calling clear_class_gmm, the GMM can no longer be used. The handle GMMHandle becomes invalid.

Parameters

▷ **GMMHandle** (input_control) ..... class_gmm(-array) ⇝ *handle*
GMM handle.

Result

If the parameters are valid, the operator clear_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- GMMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

classify_class_gmm, evaluate_class_gmm

See also

create_class_gmm, read_class_gmm, write_class_gmm, train_class_gmm

Module

Foundation

clear_samples_class_gmm (: : GMMHandle :)

Clear the training data of a Gaussian Mixture Model.

clear_samples_class_gmm clears all training samples that have been stored in the Gaussian Mixture Model (GMM) GMMHandle. clear_samples_class_gmm should only be used if the GMM is trained in the same process that uses the GMM for evaluation with evaluate_class_gmm or for classification with classify_class_gmm. In this case, the memory required for the training samples can be freed with clear_samples_class_gmm, and hence memory can be saved. In the normal usage, in which the GMM is trained offline and written to a file with write_class_gmm, it is typically unnecessary to call clear_samples_class_gmm because write_class_gmm does not save the training samples, and hence the online process, which reads the GMM with read_class_gmm, requires no memory for the training samples.

Parameters

▷ **GMMHandle** (input_control) ..... class_gmm(-array) ⇝ *handle*
GMM handle.

Result

If the parameters are valid, the operator clear_samples_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- GMMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

train_class_gmm, write_samples_class_gmm

See also

create_class_gmm, clear_class_gmm, add_sample_class_gmm, read_samples_class_gmm

Module

Foundation

create_class_gmm (: : NumDim, NumClasses, NumCenters, CovarType, Preprocessing, NumComponents, RandSeed : GMMHandle)

Create a Gaussian Mixture Model for classification

create_class_gmm creates a Gaussian Mixture Model (GMM) for classification. NumDim specifies the number of dimensions of the feature space, NumClasses specifies the number of classes. A GMM consists of NumCenters *Gaussian centers* per class. NumCenters can not only be the exact number of centers to be used, but, depending on the number of parameters, can specify upper and lower bounds for the number of centers:

exactly one parameter: The parameter determines the exact number of centers to be used for all classes.

exactly two parameters: The first parameter determines the minimum number of centers, the second determines the maximum number of centers for all classes.

exactly $2 \cdot NumClasses$ **parameters:** Alternatingly every first parameter determines the minimum number of centers per class and every second parameters determines the maximum number of centers per class.

When upper and lower bounds are specified, the optimum number of centers will be determined with the help of the Minimum Message Length Criterion (MML). In general, we recommend to start the training with (too) many centers as maximum and the expected number of centers as minimum.

Each center is described by the parameters center $\mathbf{m}_j$, covariance matrix $\mathbf{C}_j$, and mixing coefficient P_j. These parameters are calculated from the training data by means of the Expectation Maximization (EM) algorithm. A GMM

can approximate an arbitrary probability density, provided that enough centers are being used. The covariance matrices $\mathbf{C}_j$ have the dimensions NumDim · NumDim (NumComponents · NumComponents if preprocessing is used) and are symmetric. Further constraints can be given by CovarType:

For CovarType = *'spherical'*, $\mathbf{C}_j$ is a scalar multiple of the identity matrix $\mathbf{C}_j = s_j^2\mathbf{I}$. The center density function $p(\mathbf{x}|j)$ is

$$p(\mathbf{x}|j) = \frac{1}{(2\pi s_j^2)^{d/2}} \exp(-\frac{\|\mathbf{x}-\mathbf{m}_j\|^2}{2s_j^2})$$

For CovarType = *'diag'*, $\mathbf{C}_j$ is a diagonal matrix

$\mathbf{C}_j = diag(s_{j,1}^2, ..., s_{j,d}^2)$. The center density function $p(\mathbf{x}|j)$ is

$$p(\mathbf{x}|j) = \frac{1}{(2\pi \prod_{i=1}^{d} s_{j,i}^2)^{d/2}} \exp(-\sum_{i=1}^{d} \frac{(x_i - m_{j,i})^2}{2s_{j,i}^2})$$

For CovarType = *'full'*, $\mathbf{C}_j$ is a positive definite matrix. The center density function $p(\mathbf{x}|j)$ is

$$p(\mathbf{x}|j) = \frac{1}{(2\pi)^{d/2}|\mathbf{C}_j|^{\frac{1}{2}}} \exp(-\frac{1}{2}(\mathbf{x}-\mathbf{m}_j)^T\mathbf{C}^{-1}(\mathbf{x}-\mathbf{m}_j))$$

The complexity of the calculations increases from CovarType = *'spherical'* over CovarType = *'diag'* to CovarType = *'full'*. At the same time the flexibility of the centers increases. In general, *'spherical'* therefore needs higher values for NumCenters than *'full'*.

The procedure to use GMM is as follows: First, a GMM is created by create_class_gmm. Then, training vectors are added by add_sample_class_gmm, afterwards they can be written to disk with write_samples_class_gmm. With train_class_gmm the classifier center parameters (defined above) are determined. Furthermore, they can be saved with write_class_gmm for later classifications.

From the mixing probabilities P_j and the center density function $p(\mathbf{x}|j)$, the probability density function $p(\mathbf{x})$ can be calculated by:

$$p(x) = \sum_{j=1}^{n_{comp}} P(j)p(\mathbf{x}|j)$$

The probability density function $p(\mathbf{x})$ can be evaluated with evaluate_class_gmm for a feature vector $\mathbf{x}$. classify_class_gmm sorts the $p(\mathbf{x})$ and therefore discovers the most probable class of the feature vector.

The parameters Preprocessing and NumComponents can be used to preprocess the training data and reduce its dimensions. These parameters are explained in the description of the operator create_class_mlp.

create_class_gmm initializes the coordinates of the centers with random numbers. To ensure that the results of training the classifier with train_class_gmm are reproducible, the seed value of the random number generator is passed in RandSeed.

Parameters

▷ **NumDim** (input_control) ..... integer ⇝ *integer*
Number of dimensions of the feature space.
Default: 3
Suggested values: NumDim ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Restriction: NumDim >= 1

▷ **NumClasses** (input_control) ..... integer ⇝ *integer*
Number of classes of the GMM.
Default: 5
Suggested values: NumClasses ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Restriction: NumClasses >= 1

▷ **NumCenters** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of centers per class.
Default: 1
Suggested values: NumCenters ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30}
Restriction: NumClasses >= 1

▷ **CovarType** (input_control) . . . . . string ⇝ *string*
Type of the covariance matrices.
Default: 'spherical'
List of values: CovarType ∈ {'spherical', 'diag', 'full'}

▷ **Preprocessing** (input_control) . . . . . string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
Default: 'normalization'
List of values: Preprocessing ∈ {'none', 'normalization', 'principal_components', 'canonical_variates'}

▷ **NumComponents** (input_control) . . . . . integer ⇝ *integer*
Preprocessing parameter: Number of transformed features (ignored for Preprocessing = *'none'* and Preprocessing = *'normalization'*).
Default: 10
Suggested values: NumComponents ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Restriction: NumComponents >= 1

▷ **RandSeed** (input_control) . . . . . integer ⇝ *integer*
Seed value of the random number generator that is used to initialize the GMM with random values.
Default: 42

▷ **GMMHandle** (output_control) . . . . . class_gmm ⇝ *handle*
GMM handle.

Example

```
* Classification with Gaussian Mixture Models
create_class_gmm (NumDim , NumClasses, [1,5], 'full', 'none',\
                  NumComponents, 42, GMMHandle)
* Add the training data
for J := 0 to NumData-1 by 1
    * Features := [...]
    * ClassID := [...]
    add_sample_class_gmm (GMMHandle, Features, ClassID, Randomize)
endfor
* Train the GMM
train_class_gmm (GMMHandle, 100, 0.001, 'training', 0.0001, Centers, Iter)
* Classify unknown data in 'Features'
classify_class_gmm (GMMHandle, Features, 1, ID, Prob, Density, KSigmaProb)
```

Result

If the parameters are valid, the operator create_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

add_sample_class_gmm, add_samples_image_class_gmm

Alternatives

create_class_mlp, create_class_svm

See also

clear_class_gmm, train_class_gmm, classify_class_gmm, evaluate_class_gmm, classify_image_class_gmm

References

Christopher M. Bishop: "Neural Networks for Pattern Recognition"; Oxford University Press, Oxford; 1995.

Mario A.T. Figueiredo: "Unsupervised Learning of Finite Mixture Models"; IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 24, No. 3; March 2002.

Module

Foundation

deserialize_class_gmm (: : SerializedItemHandle : GMMHandle)

Deserialize a serialized Gaussian Mixture Model.

deserialize_class_gmm deserializes a Gaussian Mixture Model (GMM) (including its training samples), that was serialized by serialize_class_gmm (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized Gaussian Mixture Model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created Gaussian Mixture Model with the handle GMMHandle.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **GMMHandle** (output_control) ..... class_gmm ⇝ *handle*
GMM handle.

Result

If the parameters are valid, the operator deserialize_class_gmm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_class_gmm

Possible Successors

classify_class_gmm, evaluate_class_gmm, create_class_lut_gmm

See also

create_class_gmm, write_class_gmm, serialize_class_gmm

Module

Foundation

evaluate_class_gmm (: : GMMHandle, Features : ClassProb, Density, KSigmaProb)

Evaluate a feature vector by a Gaussian Mixture Model.

evaluate_class_gmm computes three different probability values for a feature vector Features with the Gaussian Mixture Model (GMM) GMMHandle.

The a-posteriori probability of class i for the sample Features(x) is computed as

$$p(i|\mathbf{x}) = \sum_{j=1}^{n_{comp}} P(j)p(\mathbf{x}|j)$$

and returned for each class in ClassProb. The formulas for the calculation of the center density function $p(\mathbf{x}|j)$ are described with create_class_gmm.

The probability density of the feature vector is computed as a sum of the posterior class probabilities

$$p(\mathbf{x}) = \sum_{i=1}^{n_{classes}} Pr(i)p(i|\mathbf{x})$$

and is returned in Density. Here, $Pr(i)$ are the prior classes probabilities as computed by train_class_gmm. Density can be used for novelty detection, i.e., to reject feature vectors that do not belong to any of the trained classes. However, since Density depends on the scaling of the feature vectors and since Density is a probability density, and consequently does not need to lie between 0 and 1, the novelty detection can typically be performed more easily with KSigmaProb (see below).

A *k-sigma error ellipsoid* is defined as a locus of points for which

$$(\mathbf{x} - \mu)^T C^{-1} (\mathbf{x} - \mu) = k^2$$

In the one dimensional case this is the interval $[\mu - k\sigma, \mu + k\sigma]$. For any 1D Gaussian distribution, it is true that approximately 68% of the occurrences of the random variable are within this range for $k = 1$, approximately 95% for $k = 2$, approximately 99% for $k = 3$, etc. This probability is called k-sigma probability and is denoted by $P[k]$. $P[k]$ can be computed numerically for univariate as well as for multivariate Gaussian distributions, where it should be noted that for the same values of k, $P^{(N)}[k] > P^{(N+1)}[k]$ (here N and (N+1) denote dimensions). For Gaussian mixture models the k-sigma probability is computed as:

$$P_{GMM}[\mathbf{x}] = \sum_{j=1}^{n_{comp}} P(j) P_j[k_j]$$

where

$$k_j^2 = (\mathbf{x} - \mu_j)^T C_j^{-1} (\mathbf{x} - \mu_j)$$

. $P_{GMM}[k]$ are weighted with the class priors and then normalized. The maximum value of all classes is returned in KSigmaProb, such that

$$\text{KSigmaProb} = \frac{1}{\Pr_{max}} \max\left(Pr(i) P_{GMM}[\mathbf{x}]\right)$$

KSigmaProb can be used for novelty detection, as it indicates how well a feature vector fits into the distribution of the class it is assigned to. Typically, feature vectors having values below 0.0001 should be rejected. Note that the rejection threshold defined by the parameter RejectionThreshold in classify_image_class_gmm refers to the KSigmaProb values.

Before calling evaluate_class_gmm, the GMM must be trained with train_class_gmm.

The position of the maximum value of ClassProb is usually interpreted as the class of the feature vector and the corresponding value as the probability of the class. In this case, classify_class_gmm should be used instead of evaluate_class_gmm, because classify_class_gmm directly returns the class and corresponding probability.

Parameters

- ▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
  GMM handle.
- ▷ **Features** (input_control) ..... real-array ⇝ *real*
  Feature vector.
- ▷ **ClassProb** (output_control) ..... real-array ⇝ *real*
  A-posteriori probability of the classes.
- ▷ **Density** (output_control) ..... real ⇝ *real*
  Probability density of the feature vector.

▷ **KSigmaProb** (output_control) . . . . . real ⇝ *real*
Normalized k-sigma-probability for the feature vector.

Result

If the parameters are valid, the operator `evaluate_class_gmm` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`train_class_gmm`, `read_class_gmm`

Alternatives

`classify_class_gmm`

See also

`create_class_gmm`

References

Christopher M. Bishop: "Neural Networks for Pattern Recognition"; Oxford University Press, Oxford; 1995.

Mario A.T. Figueiredo: "Unsupervised Learning of Finite Mixture Models"; IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 24, No. 3; March 2002.

Module

Foundation

get_class_train_data_gmm (: : GMMHandle : ClassTrainDataHandle)

Get the training data of a Gaussian Mixture Model (GMM).

`get_class_train_data_gmm` gets the training data of a Gaussian Mixture Model (GMM) and returns it in `ClassTrainDataHandle`.

Parameters

▷ **GMMHandle** (input_control) . . . . . class_gmm ⇝ *handle*
Handle of a GMM that contains training data.

▷ **ClassTrainDataHandle** (output_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data of the classifier.

Result

If the parameters are valid, the operator `get_class_train_data_gmm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`add_sample_class_gmm`, `read_samples_class_gmm`

Possible Successors

`add_class_train_data_svm`, `add_class_train_data_svm`, `add_class_train_data_knn`

See also

`create_class_train_data`

Module

Foundation

get_params_class_gmm (: : GMMHandle : NumDim, NumClasses, MinCenters, MaxCenters, CovarType)

Return the parameters of a Gaussian Mixture Model.

get_params_class_gmm returns the parameters of a Gaussian Mixture Model (GMM) that were specified when the GMM was created with create_class_gmm. This is particularly useful if the GMM was read with read_class_gmm. The output of get_params_class_gmm can, for example, be used to check whether the feature vectors and/or the target data to be used have appropriate dimensions to be used with GMM. For a description of the parameters, see create_class_gmm.

Parameters

- ▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
  GMM handle.
- ▷ **NumDim** (output_control) ..... integer ⇝ *integer*
  Number of dimensions of the feature space.
- ▷ **NumClasses** (output_control) ..... integer ⇝ *integer*
  Number of classes of the GMM.
- ▷ **MinCenters** (output_control) ..... integer-array ⇝ *integer*
  Minimum number of centers per GMM class.
- ▷ **MaxCenters** (output_control) ..... integer-array ⇝ *integer*
  Maximum number of centers per GMM class.
- ▷ **CovarType** (output_control) ..... string ⇝ *string*
  Type of the covariance matrices.

Result

If the parameters are valid, the operator get_params_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_class_gmm, read_class_gmm

Possible Successors

add_sample_class_gmm, train_class_gmm

See also

evaluate_class_gmm, classify_class_gmm

Module

Foundation

get_prep_info_class_gmm (: : GMMHandle, Preprocessing : InformationCont, CumInformationCont)

Compute the information content of the preprocessed feature vectors of a GMM.

get_prep_info_class_gmm computes the information content of the training vectors that have been transformed with the preprocessing given by Preprocessing. Preprocessing can be set to *'principal_components'* or *'canonical_variates'*. The preprocessing methods are described with create_class_mlp. The information content is derived from the variations of the transformed components of the feature vector, i.e., it

is computed solely based on the training data, independent of any error rate on the training data. The information content is computed for all relevant components of the transformed feature vectors (NumComponents for *'principal_components'* and *'canonical_variates'*, see create_class_gmm), and is returned in InformationCont as a number between 0 and 1. To convert the information content into a percentage, it simply needs to be multiplied by 100. The cumulative information content of the first n components is returned in the n-th component of CumInformationCont, i.e., CumInformationCont contains the sums of the first n elements of InformationCont. To use get_prep_info_class_gmm, a sufficient number of samples must be added to the GMM given by GMMHandle by using add_sample_class_gmm or read_samples_class_gmm.

InformationCont and CumInformationCont can be used to decide how many components of the transformed feature vectors contain relevant information. An often used criterion is to require that the transformed data must represent x% (e.g., 90%) of the data. This can be decided easily from the first value of CumInformationCont that lies above x%. The number thus obtained can be used as the value for NumComponents in a new call to create_class_gmm. The call to get_prep_info_class_gmm already requires the creation of a GMM, and hence the setting of NumComponents in create_class_gmm to an initial value. However, if get_prep_info_class_gmm is called, it is typically not known how many components are relevant, and hence how to set NumComponents in this call. Therefore, the following two-step approach should typically be used to select NumComponents: In a first step, a GMM with the maximum number for NumComponents is created (NumComponents for *'principal_components'* and *'canonical_variates'*). Then, the training samples are added to the GMM and are saved in a file using write_samples_class_gmm. Subsequently, get_prep_info_class_gmm is used to determine the information content of the components, and with this NumComponents. After this, a new GMM with the desired number of components is created, and the training samples are read with read_samples_class_gmm. Finally, the GMM is trained with train_class_gmm.

Parameters

▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
GMM handle.

▷ **Preprocessing** (input_control) ..... string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
Default: 'principal_components'
List of values: Preprocessing ∈ {'principal_components', 'canonical_variates'}

▷ **InformationCont** (output_control) ..... real-array ⇝ *real*
Relative information content of the transformed feature vectors.

▷ **CumInformationCont** (output_control) ..... real-array ⇝ *real*
Cumulative information content of the transformed feature vectors.

Example

```
* Create the initial GMM
create_class_gmm (NumDim, NumClasses, NumCenters, 'full',\
                  'principal_components', NumComponents, 42, GMMHandle)
* Generate and add the training data
for J := 0 to NumData-1 by 1
    * Generate training features and classes
    * Data = [...]
    * ClassID = [...]
    add_sample_class_gmm (GMMHandle, Data, ClassID, Randomize)
endfor
write_samples_class_gmm (GMMHandle, 'samples.gtf')
* Compute the information content of the transformed features
get_prep_info_class_gmm (GMMHandle, 'principal_components',\
                         InformationCont, CumInformationCont)
* Determine Comp by inspecting InformationCont and CumInformationCont
* NumComponents = [...]
* Create the actual GMM
create_class_gmm (NumDim, NumClasses, NumCenters, 'full',\
                  'principal_components', NumComponents, 42, GMMHandle)
* Train the GMM
read_samples_class_gmm (GMMHandle, 'samples.gtf')
```

```
train_class_gmm (GMMHandle, 200, 0.0001, 0.0001, Regularize, Centers, Iter)
write_class_gmm (GMMHandle, 'classifier.gmm')
```

Result

If the parameters are valid, the operator `get_prep_info_class_gmm` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

`get_prep_info_class_gmm` may return the error 9211 (Matrix is not positive definite) if `Preprocessing` = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`add_sample_class_gmm`, `read_samples_class_gmm`

Possible Successors

`clear_class_gmm`, `create_class_gmm`

References

Christopher M. Bishop: "Neural Networks for Pattern Recognition"; Oxford University Press, Oxford; 1995.

Andrew Webb: "Statistical Pattern Recognition"; Arnold, London; 1999.

Module

Foundation

get_sample_class_gmm (: : GMMHandle, NumSample : Features, ClassID)

Return a training sample from the training data of a Gaussian Mixture Models (GMM).

`get_sample_class_gmm` reads out a training sample from the Gaussian Mixture Model (GMM) given by `GMMHandle` that was stored with `add_sample_class_gmm` or `add_samples_image_class_gmm`. The index of the sample is specified with `NumSample`. The index is counted from 0, i.e., `NumSample` must be a number between 0 and `NumSamples` − 1, where `NumSamples` can be determined with `get_sample_num_class_gmm`. The training sample is returned in `Features` and `ClassID`. `Features` is a feature vector of length `NumDim`, while `ClassID` is its class (see `add_sample_class_gmm` and `create_class_gmm`).

`get_sample_class_gmm` can, for example, be used to reclassify the training data with `classify_class_gmm` in order to determine which training samples, if any, are classified incorrectly.

Parameters

▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
GMM handle.

▷ **NumSample** (input_control) ..... integer ⇝ *integer*
Index of the stored training sample.

▷ **Features** (output_control) ..... real-array ⇝ *real*
Feature vector of the training sample.

▷ **ClassID** (output_control) ..... number ⇝ *integer*
Class of the training sample.

Example

```
create_class_gmm (2, 2, [1,10], 'spherical', 'none', 2, 42, GMMHandle)
read_samples_class_gmm (GMMHandle, 'samples.gsf')
train_class_gmm (GMMHandle, 100, 1e-4, 'training', 1e-4, Centers, Iter)
```

```
* Reclassify the training samples
get_sample_num_class_gmm (GMMHandle, NumSamples)
for I := 0 to NumSamples-1 by 1
  get_sample_class_gmm (GMMHandle, I, Features, Class)
  classify_class_gmm (GMMHandle, Features, 2, ClassID, ClassProb,\
                      Density, KSigmaProb)
  if (not (Class == ClassProb[0]))
    * classified incorrectly
  endif
endfor
```

Result

If the parameters are valid, the operator `get_sample_class_gmm` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`add_sample_class_gmm`, `add_samples_image_class_gmm`, `read_samples_class_gmm`, `get_sample_num_class_gmm`

Possible Successors

`classify_class_gmm`, `evaluate_class_gmm`

See also

`create_class_gmm`

Module

Foundation

`get_sample_num_class_gmm` (: : GMMHandle : NumSamples)

Return the number of training samples stored in the training data of a Gaussian Mixture Model (GMM).

`get_sample_num_class_gmm` returns in `NumSamples` the number of training samples that are stored in the Gaussian Mixture Model (GMM) given by `GMMHandle`. `get_sample_num_class_gmm` should be called before the individual training samples are read out with `get_sample_class_gmm`, e.g., for the purpose of reclassifying the training data (see `get_sample_class_gmm`).

Parameters

▷ **`GMMHandle`** (input_control) ..... class_gmm ⇝ *handle*
GMM handle.

▷ **`NumSamples`** (output_control) ..... integer ⇝ *integer*
Number of stored training samples.

Result

If the parameters are valid, the operator `get_sample_num_class_gmm` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_gmm, add_samples_image_class_gmm, read_samples_class_gmm

Possible Successors

get_sample_class_gmm

See also

create_class_gmm

Module

Foundation

read_class_gmm (: : FileName : GMMHandle)

Read a Gaussian Mixture Model from a file.

read_class_gmm reads a Gaussian Mixture Model (GMM) that has been stored with write_class_gmm. Since the training of an GMM can consume a relatively long time, the GMM is typically trained in an offline process and written to a file with write_class_gmm. In the online process the GMM is read with read_class_gmm and subsequently used for evaluation with evaluate_class_gmm or for classification with classify_class_gmm. The default HALCON file extension for the GMM classifier is 'ggc'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .ggc

▷ **GMMHandle** (output_control) . . . . . class_gmm ⇝ *handle*
GMM handle.

Result

If the parameters are valid, the operator read_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

classify_class_gmm, evaluate_class_gmm, create_class_lut_gmm

See also

create_class_gmm, write_class_gmm

Module

Foundation

read_samples_class_gmm (: : GMMHandle, FileName :)

Read the training data of a Gaussian Mixture Model from a file.

read_samples_class_gmm reads training samples from the file given by FileName and adds them to the training samples that have already been stored in the Gaussian Mixture Model (GMM) given by GMMHandle. The GMM must be created with create_class_gmm before calling read_samples_class_gmm. As described with train_class_gmm and write_samples_class_gmm, read_samples_class_gmm, add_sample_class_gmm, and write_samples_class_gmm can be used to build up a database of training samples, and hence to improve the performance of the GMM by retraining the GMM with extended data sets.

It should be noted that the training samples must have the correct dimensionality. The feature vectors stored in FileName must have the lengths NumDim that was specified with create_class_gmm, and enough classes must have been created in create_class_gmm. If this is not the case, an error message is returned.

It is possible to read files of samples that were written with write_samples_class_svm or write_samples_class_mlp.

Parameters

▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
GMM handle.

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
File name.

Result

If the parameters are valid, the operator read_samples_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- GMMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_gmm

Possible Successors

train_class_gmm

Alternatives

add_sample_class_gmm

See also

write_samples_class_gmm, write_samples_class_mlp, clear_samples_class_gmm

Module

Foundation

select_feature_set_gmm (: : ClassTrainDataHandle, SelectionMethod, GenParamName, GenParamValue : GMMHandle, SelectedFeatureIndices, Score)

Selects an optimal combination from a set of features to classify the provided data.

select_feature_set_gmm selects an optimal subset from a set of features to solve a given classification problem. The classification problem has to be specified with annotated training data in ClassTrainDataHandle and will be classified by a Gaussian Mixture Model. Details of the properties of this classifier can be found in create_class_gmm.

The result of the operator is a trained classifier that is returned in GMMHandle. Additionally, the list of indices or names of the selected features is returned in SelectedFeatureIndices. To use this classifier, calculate for new input data all features mentioned in SelectedFeatureIndices and pass them to the classifier.

A possible application of this operator can be a comparison of different parameter sets for certain feature extraction techniques. Another application is to search for a feature that is discriminating between different classes.

To define the features that should be selected from ClassTrainDataHandle, the dimensions of the feature vectors in ClassTrainDataHandle can be grouped into subfeatures by calling

set_feature_lengths_class_train_data. A subfeature can contain several subsequent elements of a feature vector. select_feature_set_gmm decides for each of these subfeatures, if it is better to use it for the classification or leave it out.

The indices of the selected subfeatures are returned in SelectedFeatureIndices. If names were set in set_feature_lengths_class_train_data, these names are returned instead of the indices. If set_feature_lengths_class_train_data was not called for ClassTrainDataHandle before, each element of the feature vector is considered as a subfeature.

The selection method SelectionMethod is either a greedy search *'greedy'* (iteratively add the feature with highest gain) or the dynamically oscillating search *'greedy_oscillating'* (add the feature with highest gain and test then if any of the already added features can be left out without great loss). The method *'greedy'* is generally preferable, since it is faster. Only in cases when the subfeatures are low-dimensional or redundant, the method *'greedy_oscillating'* should be chosen.

The optimization criterion is the classification rate of a two-fold cross-validation of the training data. The best achieved value is returned in Score.

The following generic parameters can be set in GenParamName and GenParamValue:

'min_centers': Minimal number of clusters to represent a class in the training data.
Suggested values: *'1'*, *'2'*
Default: *'1'*

'max_center': Maximal number of clusters to represent a class in the training data.
Suggested values: *'1'*, *'5'*, *'10'*
Default: *'1'*

'covar_type': Type of the covariance to represent the size of a cluster.
List of values: *'spherical'*, *'diag'*, *'full'*
Default: *'spherical'*

'random_seed': Random seed.
Default: *'42'*

'threshold': Training threshold.
Default: *'0.001'*

'regularize': Regularization value.
Default: *'0.0001'*

'randomize': Randomize the input vector.
Default: *'0'*

'class_priors': Mode to determine the a-priori probabilities of the classes.
List of values: *'training'*, *'uniform'*
Default: *'training'*

A more exact description of those parameters can be found in create_class_gmm and train_class_gmm.

Attention

This operator may take considerable time, depending on the size of the data set in the training file, and the number of features.

Please note, that this operator should not be called, if only a small set of training data is available. Due to the risk of overfitting the operator select_feature_set_gmm may deliver a classifier with a very high score. However, the classifier may perform poorly when tested.

Parameters

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data.

▷ **SelectionMethod** (input_control) . . . . . string ⇝ *string*
Method to perform the selection.
Default: 'greedy'
List of values: SelectionMethod ∈ {'greedy', 'greedy_oscillating'}

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Names of generic parameters to configure the classifier.
Default: []
List of values: GenParamName ∈ {'min_centers', 'max_center', 'covar_type', 'random_seed', 'threshold', 'regularize', 'randomize', 'class_priors'}

▷ **GenParamValue** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Values of generic parameters to configure the classifier.
Default: []
Suggested values: GenParamValue ∈ {1, 2, 3, 'spherical', 'diag', 'full', 42, 0.001, 0.0001, 0}

▷ **GMMHandle** (output_control) . . . . . class_gmm ⇝ *handle*
A trained GMM classifier using only the selected features.

▷ **SelectedFeatureIndices** (output_control) . . . . . string-array ⇝ *string*
The selected feature set, contains indices or names.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
The achieved score using two-fold cross-validation.

Example

```
* Find out which of the two features distinguishes two Classes
NameFeature1 := 'Good Feature'
NameFeature2 := 'Bad Feature'
LengthFeature1 := 3
LengthFeature2 := 2
* Create training data
create_class_train_data (LengthFeature1+LengthFeature2,\
  ClassTrainDataHandle)
* Define the features which are in the training data
set_feature_lengths_class_train_data (ClassTrainDataHandle, [LengthFeature1,\
  LengthFeature2], [NameFeature1, NameFeature2])
* Add training data
*                                                          |Feat1|  |Feat2|
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  2,1  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  2,1  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  3,4  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  3,4  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [0,0,1,  5,6  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,3,2,  5,6  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [0,0,1,  5,6  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,3,2,  5,6  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [0,0,1,  5,6  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,3,2,  5,6  ], 1)
* Add more data
* ...
* Select the better feature with a GMM
select_feature_set_gmm (ClassTrainDataHandle, 'greedy', [], [], GMMHandle,\
  SelectedFeatureGMM, Score)
* Use the classifier
* ...
```

Result

If the parameters are valid, the operator select_feature_set_gmm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`create_class_train_data`, `add_sample_class_train_data`, `set_feature_lengths_class_train_data`

Possible Successors

`classify_class_gmm`

Alternatives

`select_feature_set_mlp`, `select_feature_set_knn`, `select_feature_set_svm`

See also

`create_class_gmm`, `gray_features`, `region_features`

Module

Foundation

`serialize_class_gmm` (: : GMMHandle : SerializedItemHandle)

Serialize a Gaussian Mixture Model (GMM).

`serialize_class_gmm` serializes a Gaussian Mixture Model (GMM) and its stored training samples (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The same data that is written in a file by `write_class_gmm` and `write_samples_class_gmm` is converted to a serialized item. The Gaussian Mixture Model is defined by the handle `GMMHandle`. The serialized Gaussian Mixture Model is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_class_gmm`.

Parameters

▷ **`GMMHandle`** (input_control) ..... class_gmm ⇝ *handle*
GMM handle.

▷ **`SerializedItemHandle`** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_class_gmm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`train_class_gmm`

Possible Successors

`clear_class_gmm`, `fwrite_serialized_item`, `send_serialized_item`, `deserialize_class_gmm`

See also

`create_class_gmm`, `read_class_gmm`, `write_samples_class_gmm`, `deserialize_class_gmm`

Module

Foundation

`train_class_gmm` (: : GMMHandle, MaxIter, Threshold, ClassPriors, Regularize : Centers, Iter)

Train a Gaussian Mixture Model.

train_class_gmm trains the Gaussian Mixture Model (GMM) referenced by GMMHandle. Before the GMM can be trained, all training samples to be used for the training must be stored in the GMM using add_sample_class_gmm, add_samples_image_class_gmm, or read_samples_class_gmm. After the training, new training samples can be added to the GMM and the GMM can be trained again.

During the training, the error that results from the GMM applied to the training vectors will be minimized with the expectation maximization (EM) algorithm.

MaxIter specifies the maximum number of iterations per class for the EM algorithm. In practice, values between *20* and *200* should be sufficient for most problems. Threshold specifies a threshold for the relative changes of the error. If the relative change in error exceeds the threshold after MaxIter iterations, the algorithm will be canceled for this class. Because the algorithm starts with the maximum specified number of centers (parameter NumCenters in create_class_gmm), in case of a premature termination the number of centers and the error for this class will not be optimal. In this case, a new training with different parameters (e.g., another value for RandSeed in create_class_gmm) can be tried.

ClassPriors specifies the method of calculation of the class priors in GMM. If *'training'* is specified, the priors of the classes are taken from the proportion of the corresponding sample data during training. If *'uniform'* is specified, the priors are set equal to 1/NumClasses for all classes.

Regularize is used to regularize (nearly) *singular* covariance matrices during the training. A covariance matrix might *collapse* to singularity if it is trained with linearly dependent data. To avoid this, a small value specified by Regularize is added to each main diagonal element of the covariance matrix, which prevents this element from becoming smaller than Regularize. A recommended value for Regularize is *0.0001*. If Regularize is set to *0.0*, no regularization is performed.

The centers are initially randomly distributed. In individual cases, relatively high errors will result from the algorithm because the initial random values determined by RandSeed in create_class_gmm lead to local minima. In this case, a new GMM with a different value for RandSeed should be generated to test whether a significantly smaller error can be obtained.

It should be noted that, depending on the number of centers, the type of covariance matrix, and the number of training samples, the training can take from a few seconds to several hours.

On output, train_class_gmm returns in Centers the number of centers per class that have been found to be optimal by the EM algorithm. These values can be used as a reference in NumCenters (in create_class_gmm) for future GMMs. If the number of centers found by training a new GMM on integer training data is unexpectedly high, this might be corrected by adding a Randomize noise to the training data in add_sample_class_gmm. Iter contains the number of performed iterations per class. If a value in Iter equals MaxIter, the training algorithm has been terminated prematurely (see above).

Parameters

▷ **GMMHandle** (input_control) . . . . . class_gmm ⇝ *handle*
GMM handle.

▷ **MaxIter** (input_control) . . . . . integer ⇝ *integer*
Maximum number of iterations of the expectation maximization algorithm
Default: 100
Suggested values: MaxIter ∈ {10, 20, 30, 50, 100, 200}

▷ **Threshold** (input_control) . . . . . real ⇝ *real*
Threshold for relative change of the error for the expectation maximization algorithm to terminate.
Default: 0.001
Suggested values: Threshold ∈ {0.001, 0.0001}
Restriction: Threshold >= 0.0 && Threshold <= 1.0

▷ **ClassPriors** (input_control) . . . . . string ⇝ *string*
Mode to determine the a-priori probabilities of the classes
Default: 'training'
List of values: ClassPriors ∈ {'training', 'uniform'}

▷ **Regularize** (input_control) . . . . . real ⇝ *real*
Regularization value for preventing covariance matrix singularity.
Default: 0.0001
Restriction: Regularize >= 0.0 && Regularize < 1.0

▷ **Centers** (output_control) . . . . . integer-array ⇝ *integer*
Number of found centers per class

▷ **Iter** (output_control) . . . . . integer-array ⇝ *integer*
Number of executed iterations per class

Example

```
create_class_gmm (NumDim, NumClasses, [1,5], 'full', 'none', 0, 42,\
                  GMMHandle)
* Add the training data
read_samples_class_gmm (GMMHandle, 'samples.gsf')
* Train the GMM
train_class_gmm (GMMHandle, 100, 1e-4, 'training', 1e-4, Centers, Iter)
* Write the Gaussian Mixture Model to file
write_class_gmm (GMMHandle, 'gmmclassifier.gmm')
```

Result

If the parameters are valid, the operator `train_class_gmm` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- GMMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`add_sample_class_gmm`, `read_samples_class_gmm`

Possible Successors

`evaluate_class_gmm`, `classify_class_gmm`, `write_class_gmm`, `create_class_lut_gmm`

Alternatives

`read_class_gmm`

See also

`create_class_gmm`

References

Christopher M. Bishop: “Neural Networks for Pattern Recognition”; Oxford University Press, Oxford; 1995.

Mario A.T. Figueiredo: “Unsupervised Learning of Finite Mixture Models”; IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 24, No. 3; March 2002.

Module

Foundation

write_class_gmm (: : GMMHandle, FileName :)

Write a Gaussian Mixture Model to a file.

`write_class_gmm` writes the Gaussian Mixture Model (GMM) `GMMHandle` to the file given by `FileName`. The default HALCON file extension for the GMM classifier is ’ggc’. `write_class_gmm` is typically called after the GMM has been trained with `train_class_gmm`. The GMM can be read with `read_class_gmm`. `write_class_gmm` does *not* write any training samples that possibly have been stored in the GMM. For this purpose, `write_samples_class_gmm` should be used.

Parameters

▷ **GMMHandle** (input_control) . . . . . class_gmm ⇝ *handle*
GMM handle.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: .ggc

Result

If the parameters are valid, the operator write_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_gmm

Possible Successors

clear_class_gmm

See also

create_class_gmm, read_class_gmm, write_samples_class_gmm

Module

Foundation

write_samples_class_gmm (: : GMMHandle, FileName :)

Write the training data of a Gaussian Mixture Model to a file.

write_samples_class_gmm writes the training samples stored in the Gaussian Mixture Model (GMM) GMMHandle to the file given by FileName. write_samples_class_gmm can be used to build up a database of training samples, and hence to improve the performance of the GMM by training it with an extended data set (see train_class_gmm).

The file FileName is overwritten by write_samples_class_gmm. Nevertheless, extending the database of training samples is easy because read_samples_class_gmm and add_sample_class_gmm add the training samples to the training samples that are already stored in memory with the GMM.

The created file can be read with read_samples_class_mlp if the classifier of a multilayer perceptron (MLP) should be used. The class of a training sample in the GMM corresponds to a component of the target vector in the MLP being 1.0.

Parameters

▷ **GMMHandle** (input_control) . . . . . class_gmm ⇝ *handle*
GMM handle.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.

Result

If the parameters are valid, the operator write_samples_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_gmm

Possible Successors

clear_samples_class_gmm

See also

create_class_gmm, read_samples_class_gmm, read_samples_class_mlp, write_samples_class_mlp

Module

Foundation

7.2 K-Nearest Neighbors

add_class_train_data_knn (: : KNNHandle, ClassTrainDataHandle :)

Add training data to a k-nearest neighbors (k-NN) classifier.

add_class_train_data_knn adds the training data specified by ClassTrainDataHandle to a k-nearest neighbors (k-NN) classifier specified by KNNHandle.

Parameters

- ▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
  Handle of a k-NN which receives the training data.
- ▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
  Training data for a classifier.

Result

If the parameters are valid, the operator add_class_train_data_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- KNNHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_knn, create_class_train_data

Possible Successors

get_sample_class_knn

Alternatives

add_sample_class_knn

See also

create_class_knn

Module

Foundation

add_sample_class_knn (: : KNNHandle, Features, ClassID :)

Add a sample to a k-nearest neighbors (k-NN) classifier.

add_sample_class_knn adds a feature vector to a k-nearest neighbors (k-NN) data structure. The length of a feature vector was specified in create_class_knn by NumDim. A handle to a k-NN data structure has to be specified in KNNHandle.

The feature vectors are collected in Features. The length of the input vector must be a multiple of NumDim. Each feature vector needs a class which can be given by ClassID, if only one was specified, the class is used for all vectors. The class is a natural number greater or equal to 0. If only one class is used, the class has to be 0. In case the operator classify_image_class_knn will be used, all numbers starting from 0 to the number of classes-1 should be used, since otherwise an empty region will be generated for each unused number.

It is allowed to add samples to an already trained k-NN classificator. The new data is only integrated after another call to train_class_knn.

If the k-NN classifier has been trained with automatic feature normalization enabled, the supplied features Features are interpreted as unnormalized and are normalized as it was defined by the last call to train_class_knn. Please see train_class_knn for more information on normalization.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **Features** (input_control) . . . . . number(-array) ⇝ *real*
List of features to add.

▷ **ClassID** (input_control) . . . . . integer(-array) ⇝ *integer*
Class IDs of the features.

Result

If the parameters are valid, the operator add_sample_class_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- KNNHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

train_class_knn, read_class_knn

See also

create_class_knn, read_class_knn

References

Marius Muja, David G. Lowe: “Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration”; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

classify_class_knn (: : KNNHandle, Features : Result, Rating)

Search for the next neighbors for a given feature vector.

classify_class_knn searches for the next *'k'* neighbors of the feature vector given in Features. The distance which is used to determine the next neighbor is the L2-norm of the given vector and the training samples.

The value of *'k'* can be set via set_params_class_knn. The results can either be the determined class of the feature vector or the indices of the nearest neighbors. The selection of the result behavior can be made by set_params_class_knn via the generic parameters *'method'* and *'max_num_classes'*:

'classes_distance': returns the nearest samples for each of maximally *'max_num_classes'* different classes, if they have a representative in the nearest *'k'* neighbors. The results in `Result` are classes sorted by their minimal distance in `Rating`. There is no efficient way to determine in a k-NN-tree the nearest neighbor for exactly *'max_num_classes'* classes.

'classes_frequency': counts the occurrences of certain classes among the nearest *'k'* neighbors and returns the occurring classes in `Result` sorted by their relative frequency that is returned in `Rating`. Again, maximally *'max_num_classes'* values are returned.

'classes_weighted_frequencies': counts the occurrences of certain classes among the nearest *'k'* neighbors and returns the occurring classes in `Result` sorted by their relative frequency weighted with the average distance that is returned in `Rating`. Again, maximally *'max_num_classes'* values are returned.

'neighbors_distance': returns the indices of the nearest *'k'* neighbors in `Result` and the distances in `Rating`.

The default behavior is *'classes_distance'* and returns the classes and distances.

Parameters

▷ **`KNNHandle`** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **`Features`** (input_control) . . . . . number-array ⇝ *real*
Features that should be classified.

▷ **`Result`** (output_control) . . . . . number-array ⇝ *integer*
The classification result, either class IDs or sample indices.

▷ **`Rating`** (output_control) . . . . . number-array ⇝ *real*
A rating for the results. This value contains either a distance, a frequency or a weighted frequency.

Result

If the parameters are valid, the operator `classify_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`train_class_knn`, `read_class_knn`, `set_params_class_knn`

Possible Successors

`clear_class_knn`

See also

`create_class_knn`, `read_class_knn`

References

Marius Muja, David G. Lowe: “Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration”; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

`clear_class_knn` (: : KNNHandle :)

Clear a k-NN classifier.

`clear_class_knn` clears the k-NN classifiers given in `KNNHandle`. After calling `clear_class_knn`, `KNNHandle` becomes invalid.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

Result

If the parameters are valid, the operator `clear_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- KNNHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`train_class_knn`, `read_class_knn`

See also

`create_class_knn`

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

create_class_knn (: : NumDim : KNNHandle)

Create a k-nearest neighbors (k-NN) classifier.

`create_class_knn` creates a k-nearest neighbors (k-NN) data structure. This can be either used to classify data or to approximately locate nearest neighbors in a `NumDim`-dimensional space.

Most of the operators described in Classification/K-Nearest-Neighbor use the resulting handle `KNNHandle`.

The k-NN classifies by searching approximately the nearest neighbors and returning their classes as result. With the used approximation, the search time is logarithmically to the number of samples and dimensions.

The dimension of the feature vectors is the only parameter that necessarily has to be set in `NumDim`.

Parameters

▷ **NumDim** (input_control) . . . . . number-array ⇝ *integer*
Number of dimensions of the feature.
Default: 10

▷ **KNNHandle** (output_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

Result

If the parameters are valid, the operator `create_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

add_sample_class_knn, train_class_knn

Alternatives

create_class_svm, create_class_mlp

See also

select_feature_set_knn, read_class_knn

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

deserialize_class_knn (: : SerializedItemHandle : KNNHandle)

Deserialize a serialized k-NN classifier.

deserialize_class_knn deserializes a k-NN classifier (k-NN) (including its training samples), that was serialized by serialize_class_knn (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized k-NN classifier is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created k-NN classifier with the handle KNNHandle.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **KNNHandle** (output_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

Result

If the parameters are valid, the operator deserialize_class_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_class_knn

Possible Successors

classify_class_knn

Alternatives

serialize_class_knn

See also

create_class_knn

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

get_class_train_data_knn (: : KNNHandle : ClassTrainDataHandle)

Get the training data of a k-nearest neighbors (k-NN) classifier.

get_class_train_data_knn gets the training data of a k-nearest neighbors (k-NN) classifier and returns it in ClassTrainDataHandle.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier that contains training data.

▷ **ClassTrainDataHandle** (output_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data of the classifier.

Result

If the parameters are valid, the operator get_class_train_data_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

add_sample_class_knn

Possible Successors

add_class_train_data_svm, add_class_train_data_gmm, add_class_train_data_knn

See also

create_class_train_data

Module

Foundation

get_params_class_knn (: : KNNHandle,
GenParamName : GenParamValue)

Get parameters of a k-NN classification.

get_params_class_knn gets parameters of the k-NN referred by KNNHandle. The possible entries in GenParamName are:

'method': Retrieve the currently selected method for determining the result of classify_class_knn. The result can be *'classes_distance'*, *'classes_frequency'*, *'classes_weighted_frequencies'* or *'neighbors_distance'*.

'k': The number of nearest neighbors that is considered to determine the results.

'max_num_classes': The maximum number of classes that are returned. This parameter is ignored in case the method *'neighbors_distance'* is selected.

'num_checks': Defines the maximum number of runs through the trees.

'epsilon': A parameter to lower the accuracy in the tree to gain speed.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of the parameters that can be read from the k-NN classifier.
Default: ['method','k']
List of values: GenParamName ∈ {'method', 'num_checks', 'epsilon', 'k'}

▷ **GenParamValue** (output_control) . . . . . number-array ⇝ *integer / real / string*
Values of the selected parameters.

Result

If the parameters are valid, the operator `get_params_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`train_class_knn`, `read_class_knn`

Possible Successors

`classify_class_knn`

See also

`create_class_knn`, `read_class_knn`

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

get_sample_class_knn (: : KNNHandle, IndexSample : Features, ClassID)

Return a training sample from the training data of a k-nearest neighbors (k-NN) classifier.

`get_sample_class_knn` reads a training sample from the k-nearest neighbors (k-NN) classifier given by `KNNHandle` that was added with `add_sample_class_knn` or `read_class_knn`. The index of the sample is specified with `IndexSample`. The index is counted from 0, i.e., `IndexSample` must be a number between 0 and `NumSamples` −1, where `NumSamples` can be determined with `get_sample_num_class_knn`. The training sample is returned in `Features` and `ClassID`. `Features` is a feature vector of length `NumDim` (see `create_class_knn`), while `ClassID` is the class label, which is a number between 0 and the number of classes.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **IndexSample** (input_control) . . . . . integer ⇝ *integer*
Index of the training sample.

▷ **Features** (output_control) . . . . . real-array ⇝ *real*
Feature vector of the training sample.

▷ **ClassID** (output_control) . . . . . integer-array ⇝ *integer*
Class of the training sample.

Result

If the parameters are valid the operator `get_sample_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_train_data

See also

create_class_knn

Module

Foundation

get_sample_num_class_knn (: : KNNHandle : NumSamples)

Return the number of training samples stored in the training data of a k-nearest neighbors (k-NN) classifier.

get_sample_num_class_knn returns in NumSamples the number of training samples that are stored in the k-nearest neighbors (k-NN) classifier given by KNNHandle. get_sample_num_class_knn should be called before the individual training samples are accessed with get_sample_class_knn.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **NumSamples** (output_control) . . . . . integer ⇝ *integer*
Number of stored training samples.

Result

If KNNHandle is valid, the operator get_sample_num_class_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_knn

Possible Successors

get_sample_class_knn

See also

create_class_knn

Module

Foundation

read_class_knn (: : FileName : KNNHandle)

Read the k-NN classifier from a file.

read_class_knn reads the saved classifier from the file FileName (see write_class_knn). The values of the current classifier are overwritten. The default HALCON file extension for the k-NN classifier is 'gnc'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name of the classifier.
File extension: .gnc

▷ **KNNHandle** (output_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

Result

read_class_knn returns 2 (H_MSG_TRUE). An exception is raised if it was not possible to open the file FileName or the file has the wrong format.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

classify_class_knn

See also

create_class_knn

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

select_feature_set_knn (: : ClassTrainDataHandle, SelectionMethod, GenParamName, GenParamValue : KNNHandle, SelectedFeatureIndices, Score)

Selects an optimal subset from a set of features to solve a certain classification problem.

select_feature_set_knn selects an optimal subset from a set of features to solve a certain classification problem. The classification problem has to be specified with annotated training data in ClassTrainDataHandle and will be classified by a a k-nearest neighbors classifier. Details of the properties of this classifier can be found in create_class_knn.

The result of the operator is a trained classifier that is returned in KNNHandle. Additionally, the list of indices or names of the selected features is returned in SelectedFeatureIndices. To use this classifier, calculate for new input data all features mentioned in SelectedFeatureIndices and pass them to the classifier.

A possible application of this operator can be a comparison of different parameter sets for certain feature extraction techniques. Another application is to search for a property that is discriminating between different classes of parts or classes of errors.

To define the features that should be selected from ClassTrainDataHandle, the dimensions of the feature vectors in ClassTrainDataHandle can be grouped into subfeatures by calling set_feature_lengths_class_train_data. A subfeature can contain several subsequent elements of a feature vector. The operator decides for each of these subfeatures, if it is better to use it for the classification or leave it out.

The indices of the selected subfeatures are returned in SelectedFeatureIndices. If names were set in set_feature_lengths_class_train_data, these names are returned instead of the indices. If set_feature_lengths_class_train_data was not called for ClassTrainDataHandle before, each element of the feature vector is considered as a subfeature.

The selection method SelectionMethod is either a greedy search *'greedy'* (iteratively add the feature with highest gain) or the dynamically oscillating search *'greedy_oscillating'* (add the feature with highest gain and test then if any of the already added features can be left out without great loss). The method *'greedy'* is generally preferable, since it is faster. Only in cases when the subfeatures are low-dimensional or redundant, the method *'greedy_oscillating'* should be chosen.

The optimization criterion is the classification rate of a two-fold cross-validation of the training data. The best achieved value is returned in Score.

The k-NN classifier can be parameterized using the following values in GenParamName and GenParamValue:

'num_neighbors': The number of minimally evaluated nodes, increase this value for high dimensional data.
Suggested values: *'1'*, *'2'*, *'5'*, *'10'*
Default: *'1'*

'num_trees': Number of search trees in the k-NN classifier
Suggested values: *'1'*, *'4'*, *'10'*
Default: *'4'*

Attention

This operator may take considerable time, depending on the size of the data set in the training file, and the number of features.

Please note, that this operator should not be called, if only a small set of training data is available. Due to the risk of overfitting the operator `select_feature_set_knn` may deliver a classifier with a very high score. However, the classifier may perform poorly when tested.

Parameters

- ▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
  Handle of the training data.
- ▷ **SelectionMethod** (input_control) . . . . . string ⇝ *string*
  Method to perform the selection.
  Default: 'greedy'
  List of values: `SelectionMethod` ∈ {'greedy', 'greedy_oscillating'}
- ▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
  Names of generic parameters to configure the selection process and the classifier.
  Default: []
  List of values: `GenParamName` ∈ {'num_neighbors', 'num_trees'}
- ▷ **GenParamValue** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
  Values of generic parameters to configure the selection process and the classifier.
  Default: []
  Suggested values: `GenParamValue` ∈ {1, 2, 3}
- ▷ **KNNHandle** (output_control) . . . . . class_knn ⇝ *handle*
  A trained k-NN classifier using only the selected features.
- ▷ **SelectedFeatureIndices** (output_control) . . . . . string-array ⇝ *string*
  The selected feature set, contains indices or names.
- ▷ **Score** (output_control) . . . . . real-array ⇝ *real*
  The achieved score using two-fold cross-validation.

Example

```
* Find out which of the two features distinguishes two Classes
NameFeature1 := 'Good Feature'
NameFeature2 := 'Bad Feature'
LengthFeature1 := 3
LengthFeature2 := 2
* Create training data
create_class_train_data (LengthFeature1+LengthFeature2,\
  ClassTrainDataHandle)
* Define the features which are in the training data
set_feature_lengths_class_train_data (ClassTrainDataHandle, [LengthFeature1,\
  LengthFeature2], [NameFeature1, NameFeature2])
* Add training data
*                                                        |Feat1| |Feat2|
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  2,1  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  2,1  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  3,4  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  3,4  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [0,0,1,  5,6  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,3,2,  5,6  ], 1)
* Add more data
```

```
* ...
* Select the better feature with the k-NN classifier
select_feature_set_knn (ClassTrainDataHandle, 'greedy', [], [], KNNHandle,\
  SelectedFeatureKNN, Score)
* Use the classifier
* ...
```

Result

If the parameters are valid, the operator `select_feature_set_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`create_class_train_data`, `add_sample_class_train_data`, `set_feature_lengths_class_train_data`

Possible Successors

`classify_class_knn`

Alternatives

`select_feature_set_mlp`, `select_feature_set_svm`, `select_feature_set_gmm`

See also

`select_feature_set_trainf_knn`, `gray_features`, `region_features`

Module

Foundation

serialize_class_knn (: : KNNHandle : SerializedItemHandle)

Serialize a k-NN classifier.

`serialize_class_knn` serializes a k-NN classifier (k-NN) and its stored training samples (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The same data that is written in a file by `write_class_knn` is converted to a serialized item. The k-NN classifier is defined by the handle `KNNHandle`. The serialized k-NN classifier is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_class_knn`.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_knn, read_class_knn

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_class_knn

See also

create_class_knn, read_class_knn, deserialize_class_knn

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

set_params_class_knn (: : KNNHandle, GenParamName, GenParamValue :)

Set parameters for k-NN classification.

set_params_class_knn sets parameters for the classification of the k-nearest neighbors (k-NN) classifier KNNHandle. It controls the behavior of classify_class_knn.

The value of *'k'* can be set via GenParamName and GenParamValue. Increasing *'k'* also increases the accuracy of the resulting neighbors and increases the run time.

The results can either be the determined class of the feature vector or the indices of the nearest neighbors. The result behavior can be selected with set_params_class_knn via the generic parameters *'method'* and *'max_num_classes'*:

'classes_distance': returns the nearest samples for each of maximally *'max_num_classes'* different classes, if they have a representative in the nearest *'k'* neighbors. The results are classes sorted by their minimal distance. There is no efficient way to determine in a k-NN-tree the nearest neighbor for exactly *'max_num_classes'* classes.

'classes_frequency': counts the occurrences of certain classes among the nearest *'k'* neighbors and returns the occurrent classes sorted by their relative frequency that is returned, too. Again, maximally *'max_num_classes'* values are returned.

'classes_weighted_frequencies': counts the occurrences of certain classes among the nearest *'k'* neighbors and returns the occurrent classes sorted by their relative frequency weighted with the average distance that is returned, too. Again, maximally *'max_num_classes'* values are returned.

'neighbors_distance': returns the indices of the nearest *'k'* neighbors and the distances.

The default behavior is *'classes_distance'*.

The option *'num_checks'* allows to set the number of maximal runs through the trees. The parameter has to be positive and the default value is *32*. The higher this value is, the more accurate the results will be. As a trade-off, the running time will also be higher. Setting this parameter to *0* triggers an exact search.

The option *'epsilon'* allows to set a stop criteria if the value is increased from the default value 0.0. The higher the value is set, the less accurate results of the estimated neighbors can be expected, while it might speed up the search.

Parameters

▷ **KNNHandle** (input_control) ..... class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **GenParamName** (input_control) ..... string-array ⇝ *string*
Names of the generic parameters that can be adjusted for the k-NN classifier.
Default: ['method','k','max_num_classes']
List of values: GenParamName ∈ {'method', 'num_checks', 'epsilon', 'k', 'max_num_classes'}

▷ **GenParamValue** (input_control) ..... number-array ⇝ *integer* / real / string
Values of the generic parameters that can be adjusted for the k-NN classifier.
Default: ['classes_distance',5,1]
Suggested values: GenParamValue ∈ {'classes_distance', 'classes_frequency', 'classes_weighted_frequencies', 'neighbors_distance', 32, 0.0, 0.02, 0, 1, 2, 3, 4, 5, 6}

Result

If the parameters are valid, the operator `set_params_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- KNNHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`train_class_knn`, `read_class_knn`

Possible Successors

`classify_class_knn`

See also

`create_class_knn`, `read_class_knn`, `get_params_class_knn`

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

train_class_knn (: : KNNHandle, GenParamName, GenParamValue :)

Creates the search trees for a k-NN classifier.

`train_class_knn` creates the search trees for a k-NN classifier.

It is possible to set the number of trees via the parameters `GenParamName` and `GenParamValue` by *'num_trees'*. The default value for the number of search trees is *4*. A higher number of trees improves the accuracy of the search, but increases the run time.

It is possible to add more samples after training using the operator `add_sample_class_knn`. The added data affects the classification only, if `train_class_knn` is called again.

Automatic feature normalization can be activated by setting *'normalization'* in `GenParamName` and *'true'* in `GenParamValue`. The feature vectors are normalized by normalizing each dimension separately. For each dimension, the mean and standard deviation is calculated over the training samples. Every feature vector is normalized by subtracting the mean and dividing by the standard deviation of the individual dimension. This results in a normalization, where each dimension has zero mean and unit variance. If the standard deviation happens to be zero, only the mean is subtracted. Please note however, that a feature dimension with no standard deviation does not change the classification result and should be removed. Automatic feature normalization will change the stored training data, but the original data can be restored at any time by calling `train_class_knn` with *'normalization'* set to *'false'*. If normalization is used, the operator `classify_class_knn` interprets the input data as unnormalized and performs normalization internally as it has been defined in the last call to `train_class_knn`.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of the generic parameters that can be adjusted for the k-NN classifier creation.
Default: []
List of values: `GenParamName` ∈ {'num_trees', 'normalization'}

▷ **GenParamValue** (input_control) . . . . . number-array ⇝ *integer / string / real*
Values of the generic parameters that can be adjusted for the k-NN classifier creation.
Default: []
Suggested values: GenParamValue ∈ {4, 'false', 'true'}

Result

If the parameters are valid, the operator train_class_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- KNNHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

add_sample_class_knn, read_class_knn

Alternatives

select_feature_set_knn

See also

create_class_knn, read_class_knn

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

write_class_knn (: : KNNHandle, FileName :)

Save the k-NN classifier in a file.

write_class_knn writes the k-NN classifier KNNHandle to the file given by FileName. The classifier can be read again with read_class_knn. Since the samples are an intrinsic component of a k-NN-classifier, the operator write_class_knn saves them within the class file. In contrast to other classifiers like SVM, there is no operator for saving the samples separately. The samples can be retrieved from a k-NN-classifier using get_sample_class_knn. The default HALCON file extension for the k-NN classifier is 'gnc'.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the file in which the classifier will be written.
File extension: .gnc

Result

write_class_knn returns 2 (H_MSG_TRUE). An exception is raised if it was not possible to open file FileName.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

train_class_knn, read_class_knn

See also

create_class_knn, read_class_knn

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

7.3 Look-Up Table

clear_class_lut (: : ClassLUTHandle :)

Clear a look-up table classifier.

clear_class_lut clears the look-up table (LUT) given by ClassLUTHandle and frees all memory required for the LUT. After calling clear_class_lut, the LUT classifier can no longer be used. The handle ClassLUTHandle becomes invalid.

Parameters

▷ **ClassLUTHandle** (input_control) ..... class_lut(-array) ⇝ *handle*
Handle of the LUT classifier.

Result

If ClassLUTHandle is valid, the operator clear_class_lut returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

classify_image_class_lut

See also

create_class_lut_mlp, create_class_lut_svm, create_class_lut_gmm

Module

Foundation

create_class_lut_gmm (: : GMMHandle, GenParamName,
GenParamValue : ClassLUTHandle)

Create a look-up table using a gaussian mixture model to classify byte images.

create_class_lut_gmm generates a look-up table (LUT) ClassLUTHandle using the data of a trained gaussian mixture model (GMM) GMMHandle to classify multi-channel byte images. By using this GMM-based LUT classifier the operator classify_image_class_gmm of the subsequent classification can be replaced by the operator classify_image_class_lut. The classification gets a major speed-up, because the estimation of the class in every image point is no longer necessary since every possible response of the GMM is stored in the LUT. For the generation of the LUT, the parameters NumDim, Preprocessing, and NumComponents defined in the earlier called operator create_class_gmm are important. In NumDim, the number of image channels the images must have to be classified is defined. By using the Preprocessing (see create_class_gmm)

the number of image channels can be transformed to NumComponents. NumComponents defines the length of the feature vector, which the classifier classify_class_gmm handles internally. Because of performance and disk space, the LUT is restricted to be maximal *3*-dimensional. Since it replaces the operator classify_class_gmm, NumComponents $\leq$ *3* must hold. If there is no preprocessing that reduces the number of image channels (NumDim = NumComponents), all possible pixel values, which can occur in a byte image, are classified with classify_class_gmm. The returned classes are stored in the LUT. If there is a preprocessing that reduces the number of image channels (NumDim > NumComponents), the preprocessing parameters of the GMM are stored in a separate structure of the LUT. To create the LUT, all transformed pixel values are classified with classify_class_gmm. The returned classes are stored in the LUT. Because of the discretization of the LUT, the accuracy of the LUT classifier could become lower than the accuracy of classify_image_class_gmm. With *'bit_depth'* and *'class_selection'* the accuracy of the classification, the required storage, and the runtime needed to create the LUT can be controlled.

The following parameters of the GMM-based LUT classifier can be set with GenParamName and GenParamValue:

'bit_depth': Number of bits used from the pixels. It controls the storage requirement of the LUT classifier and is bounded by the bit depth of the image (*'bit_depth'* $\leq$ *8*). If the bit depth of the LUT is smaller (*'bit_depth'* $<$ *8*), the classes of multiple pixel combinations will be mapped to the same LUT entry, which can result in a lower accuracy for the classification. One of these clusters contains $2^{NumComponents \cdot (8-bit\_depth)}$ pixel combinations, where NumComponents denotes the dimension of the LUT, which is specified in create_class_gmm. For example, for *'bit_depth'* = *7*, NumComponents = *3*, the classes of 8 pixel combinations are mapped in the same LUT entry. The LUT requires at most $2^{NumComponents \cdot bit\_depth+2}$ bytes of storage. For example, for NumComponents = *3*, *'bit_depth'* = *8* and NumClasses $<$ *16* (specified in create_class_gmm), the LUT requires 8 MB of storage with internal storage optimization. If NumClasses = *1*, the LUT requires only 2 MB of storage by using the full bit depth of the LUT. The runtime for the classification in classify_image_class_lut becomes minimal if the LUT fits into the cache. **Suggested values:** *6*,*7*,*8*

Default: *8*

Restriction: *'bit_depth'* $\geq$ *1*, *'bit_depth'* $\leq$ *8*.

'class_selection': Method for the class selection for the LUT. Can be modified to control the accuracy and the runtime needed to create the LUT classifier. The value in *'class_selection'* is ignored if the bit depth of the LUT is maximal, thus *'bit_depth'* = *8* holds. If the bit depth of the LUT is smaller (*'bit_depth'* $<$ *8*), the classes of multiple pixel combinations will be mapped to the same LUT entry. One of these clusters contains $2^{NumComponents \cdot (8-bit\_depth)}$ pixel combinations, where NumComponents denotes the dimension of the LUT, which is specified in create_class_gmm. By choosing *'class_selection'* = *'best'*, the class that appears most often in the cluster is stored in the LUT. For *'class_selection'* = *'fast'*, only one pixel of the cluster, i.e., the pixel with the smallest value (component-wise), is classified. The returned class is stored in the LUT. In this case, the accuracy of the subsequent classification could become lower. On the other hand, the runtime needed to create the LUT can be reduced, which is proportional to the maximal needed storage of the LUT, which is defined with $2^{NumComponents \cdot bit\_depth+2}$.

List of values: *'fast'*, *'best'*

Default: *'fast'*

'rejection_threshold': Threshold for the rejection of uncertain classified points of the GMM. The parameter represents a threshold on the K-sigma probability measure returned by the classification (see classify_class_gmm and evaluate_class_gmm). All pixels having a probability below *'rejection_threshold'* are not assigned to any class.

Default: *0.0001*

Restriction: *'rejection_threshold'* $\geq$ *0*, *'rejection_threshold'* $\leq$ *1*.

Parameters

▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
GMM handle.

▷ **GenParamName** (input_control) ..... attribute.name-array ⇝ *string*
Names of the generic parameters that can be adjusted for the LUT classifier creation.
Default: []
Suggested values: GenParamName ∈ {'bit_depth', 'class_selection', 'rejection_threshold'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / integer / real
Values of the generic parameters that can be adjusted for the LUT classifier creation.
Default: []
Suggested values: GenParamValue ∈ {8, 7, 6, 'fast', 'best'}

▷ **ClassLUTHandle** (output_control) . . . . . class_lut ⇝ *handle*
Handle of the LUT classifier.

Result

If the parameters are valid, the operator create_class_lut_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

train_class_gmm, read_class_gmm

Possible Successors

classify_image_class_lut

Alternatives

create_class_lut_knn, create_class_lut_mlp, create_class_lut_svm

See also

classify_image_class_lut, clear_class_lut

Module

Foundation

create_class_lut_knn (: : KNNHandle, GenParamName, GenParamValue : ClassLUTHandle)

Create a look-up table using a k-nearest neighbors classifier (k-NN) to classify byte images.

create_class_lut_knn generates a look-up table (LUT) ClassLUTHandle using the data of a trained k-nearest neighbors classifier (k-NN) KNNHandle to classify multi-channel byte images. By using this k-NN-based LUT classifier, the operator classify_image_class_knn of the subsequent classification can be replaced by the operator classify_image_class_lut. The classification is speed up considerably, because the estimation of the class in every image point is no longer necessary since every possible response of the k-NN is stored in the LUT. For the generation of the LUT, the parameter NumDim of called operator create_class_knn is important. The number of image channels the images must have to be classified is defined in NumDim.

To create the LUT, all pixel values are classified with classify_class_knn. The returned classes are stored in the LUT. Because of the discretization of the LUT, the accuracy of the LUT classifier could become lower than the accuracy of classify_image_class_knn.

With *'bit_depth'* the accuracy of the classification, the required storage, and the runtime needed to create the LUT can be controlled.

The following parameters of the k-NN-based LUT classifier can be set with GenParamName and GenParamValue:

'bit_depth': Number of bits used from the pixels. It controls the storage requirement of the LUT classifier and is bounded by the bit depth of the image (*'bit_depth'* $\leq$ *8*). If the bit depth of the LUT is smaller (*'bit_depth'* $<$ *8*), the classes of multiple pixel combinations will be mapped to the same LUT entry, which can result in a lower accuracy for the classification. One of these clusters contains $2^{NumDim \cdot (8-bit\_depth)}$ pixel combinations, where NumDim denotes the dimension of the LUT, which is specified in create_class_knn. For example, for *'bit_depth'* $=$ *7*, NumDim $=$ *3*, the classes of 8 pixel combinations are mapped in the same LUT entry. The LUT requires at most

$2^{NumDim \cdot bit\_depth+2}$ bytes of storage. For example, for NumDim = *3*, *'bit_depth'* = *8* and number of classes is smaller than *16*, the LUT requires 8 MB of storage with internal storage optimization. The runtime for the classification in classify_image_class_lut becomes minimal if the LUT fits into the cache.

Suggested values: *6*,*7*,*8*

Default: *8*

Restriction: *'bit_depth'* $\geq$ *1*, *'bit_depth'* $\leq$ *8*.

'rejection_threshold': Threshold for the rejection of uncertain classified points of the k-NN. The parameter represents a threshold on the distance returned by the classification (see classify_class_knn). All pixels having a distance over *'rejection_threshold'* are not assigned to any class.

Default: *5*

Restriction: *'rejection_threshold'* $\geq$ *0*.

Parameters

▷ **KNNHandle** (input_control) . . . . . class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters that can be adjusted for the LUT classifier creation.
Default: []
Suggested values: GenParamName ∈ {'bit_depth', 'rejection_threshold'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / integer / real
Values of the generic parameters that can be adjusted for the LUT classifier creation.
Default: []
Suggested values: GenParamValue ∈ {8, 7, 6, 0.5, 5, 10, 50}

▷ **ClassLUTHandle** (output_control) . . . . . class_lut ⇝ *handle*
Handle of the LUT classifier.

Result

If the parameters are valid, the operator create_class_lut_knn returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

train_class_knn, read_class_knn

Possible Successors

classify_image_class_lut

Alternatives

create_class_lut_svm, create_class_lut_gmm, create_class_lut_mlp

See also

classify_image_class_lut, clear_class_lut

Module

Foundation

create_class_lut_mlp (: : MLPHandle, GenParamName, GenParamValue : ClassLUTHandle)

Create a look-up table using a multi-layer perceptron to classify byte images.

create_class_lut_mlp generates a look-up table (LUT) ClassLUTHandle using the data of a trained multi-layer perceptron (MLP) MLPHandle to classify multi-channel byte images. By using this MLP-based LUT

classifier the operator classify_image_class_mlp of the subsequent classification can be replaced by the operator classify_image_class_lut. The classification gets a major speed-up, because the estimation of the class in every image point is no longer necessary since every possible response of the MLP is stored in the LUT. For the generation of the LUT, the parameters NumInput, Preprocessing, and NumComponents defined in the earlier called operator create_class_mlp are important. In NumInput, the number of image channels the images must have to be classified is defined. By using the Preprocessing (see create_class_mlp) the number of image channels can be transformed to NumComponents. NumComponents defines the length of the feature vector, which the classifier classify_class_mlp handles internally. Because of performance and disk space, the LUT is restricted to be maximal *3*-dimensional. Since it replaces the operator classify_class_mlp, NumComponents $\leq$ *3* must hold. If there is no preprocessing that reduces the number of image channels (NumInput = NumComponents), all possible pixel values, which can occur in a byte image, are classified with classify_class_mlp. The returned classes are stored in the LUT. If there is a preprocessing that reduces the number of image channels (NumInput > NumComponents), the preprocessing parameters of the MLP are stored in a separate structure of the LUT. To create the LUT, all transformed pixel values are classified with classify_class_mlp. The returned classes are stored in the LUT. Because of the discretization of the LUT, the accuracy of the LUT classifier could become lower than the accuracy of classify_image_class_mlp. With *'bit_depth'* and *'class_selection'* the accuracy of the classification, the required storage, and the runtime needed to create the LUT can be controlled.

The following parameters of the MLP-based LUT classifier can be set with GenParamName and GenParamValue:

'bit_depth': Number of bits used from the pixels. It controls the storage requirement of the LUT classifier and is bounded by the bit depth of the image (*'bit_depth'* $\leq$ *8*). If the bit depth of the LUT is smaller (*'bit_depth'* $<$ *8*), the classes of multiple pixel combinations will be mapped to the same LUT entry, which can result in a lower accuracy for the classification. One of these clusters contains $2^{NumComponents \cdot (8-bit\_depth)}$ pixel combinations, where NumComponents denotes the dimension of the LUT, which is specified in create_class_mlp. For example, for *'bit_depth'* = *7*, NumComponents = *3*, the classes of *8* pixel combinations are mapped in the same LUT entry. The LUT requires at most $2^{NumComponents \cdot bit\_depth+2}$ bytes of storage. For example, for NumComponents = *3*, *'bit_depth'* = *8* and NumOutput $<$ *16* (specified in create_class_mlp), the LUT requires 8 MB of storage with internal storage optimization. If NumOutput = *1*, the LUT requires only 2 MB of storage by using the full bit depth of the LUT. The runtime for the classification in classify_image_class_lut becomes minimal if the LUT fits into the cache.

Suggested values: *6*,*7*,*8* **Default:** *8*

Restriction: *'bit_depth'* $\geq$ *1*, *'bit_depth'* $\leq$ *8*.

'class_selection': Method for the class selection for the LUT. Can be modified to control the accuracy and the runtime needed to create the LUT classifier. The value in *'class_selection'* is ignored if the bit depth of the LUT is maximal, thus *'bit_depth'* = *8* holds. If the bit depth of the LUT is smaller (*'bit_depth'* $<$ *8*), the classes of multiple pixel combinations will be mapped to the same LUT entry. One of these clusters contains $2^{NumComponents \cdot (8-bit\_depth)}$ pixel combinations, where NumComponents denotes the dimension of the LUT, which is specified in create_class_mlp. By choosing *'class_selection'* = *'best'*, the class that appears most often in the cluster is stored in the LUT. For *'class_selection'* = *'fast'*, only one pixel of the cluster, i.e., the pixel with the smallest value (component-wise), is classified. The returned class is stored in the LUT. In this case, the accuracy of the subsequent classification could become lower. On the other hand, the runtime needed to create the LUT can be reduced, which is proportional to the maximal needed storage of the LUT, which is defined with $2^{NumComponents \cdot bit\_depth+2}$.

List of values: *'fast'*, *'best'*

Default: *'fast'*,

'rejection_threshold': Threshold for the rejection of uncertain classified points of the MLP. The parameter represents a threshold on the probability measure returned by the classification (see classify_class_mlp and evaluate_class_mlp). All pixels having a probability below *'rejection_threshold'* are not assigned to any class.

Default: *0.5*

Restriction: *'rejection_threshold'* $\geq$ *0*, *'rejection_threshold'* $\leq$ *1*.

Parameters

▷ **MLPHandle** (input_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters that can be adjusted for the LUT classifier creation.
Default: []
Suggested values: GenParamName ∈ {'bit_depth', 'class_selection', 'rejection_threshold'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / integer / real
Values of the generic parameters that can be adjusted for the LUT classifier creation.
Default: []
Suggested values: GenParamValue ∈ {8, 7, 6, 'fast', 'best'}

▷ **ClassLUTHandle** (output_control) . . . . . class_lut ⇝ *handle*
Handle of the LUT classifier.

Result

If the parameters are valid, the operator create_class_lut_mlp returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

train_class_mlp, read_class_mlp

Possible Successors

classify_image_class_lut

Alternatives

create_class_lut_gmm, create_class_lut_knn, create_class_lut_svm

See also

classify_image_class_lut, clear_class_lut

Module

Foundation

create_class_lut_svm (: : SVMHandle, GenParamName, GenParamValue : ClassLUTHandle)

Create a look-up table using a Support-Vector-Machine to classify byte images.

create_class_lut_svm generates a look-up table (LUT) ClassLUTHandle using the data of a trained Support-Vector-Machine (SVM) SVMHandle to classify multi-channel byte images. By using this SVM-based LUT classifier the operator classify_image_class_svm of the subsequent classification can be replaced by the operator classify_image_class_lut. The classification gets a major speed-up, because the estimation of the class in every image point is no longer necessary since every possible response of the SVM is stored in the LUT. For the generation of the LUT, the parameters NumFeatures, Preprocessing, and NumComponents defined in the earlier called operator create_class_svm are important. In NumFeatures, the number of image channels the images must have to be classified is defined. By using the Preprocessing (see create_class_svm) the number of image channels can be transformed to NumComponents. NumComponents defines the length of the feature vector, which the classifier classify_class_svm handles internally. Because of performance and disk space, the LUT is restricted to be maximal *3*-dimensional. Since it replaces the operator classify_class_svm, NumComponents $\leq$ *3* must hold. If there is no preprocessing that reduces the number of image channels (NumFeatures = NumComponents), all possible pixel values, which can occur in a byte image, are classified with

classify_class_svm. The returned classes are stored in the LUT. If there is a preprocessing that reduces the number of image channels (NumFeatures > NumComponents), the preprocessing parameters of the SVM are stored in a separate structure of the LUT. To create the LUT, all transformed pixel values are classified with classify_class_svm. The returned classes are stored in the LUT. Because of the discretization of the LUT, the accuracy of the LUT classifier could become lower than the accuracy of classify_image_class_svm. With *'bit_depth'* and *'class_selection'* the accuracy of the classification, the required storage, and the runtime needed to create the LUT can be controlled.

The following parameters of the SVM-based LUT classifier can be set with GenParamName and GenParamValue:

'bit_depth': Number of bits used from the pixels. It controls the storage requirement of the LUT classifier and is bounded by the bit depth of the image (*'bit_depth'* $\leq$ *8*). If the bit depth of the LUT is smaller (*'bit_depth'* $<$ *8*), the classes of multiple pixel combinations will be mapped to the same LUT entry, which can result in a lower accuracy for the classification. One of these clusters contains $2^{NumComponents \cdot (8-bit\_depth)}$ pixel combinations, where NumComponents denotes the dimension of the LUT, which is specified in create_class_svm. For example, for *'bit_depth'* = *7*, NumComponents = *3*, the classes of 8 pixel combinations are mapped in the same LUT entry. The LUT requires at most $2^{NumComponents \cdot bit\_depth+2}$ bytes of storage. For example, for NumComponents = *3*, *'bit_depth'* = *8* and NumClasses $<$ *16* (specified in create_class_svm), the LUT requires 8 MB of storage with internal storage optimization. If NumClasses = *1*, the LUT requires only 2 MB of storage by using the full bit depth of the LUT. The runtime for the classification in classify_image_class_lut becomes minimal if the LUT fits into the cache.

Suggested values: *6*,*7*,*8*

Default: *8*

Restriction: *'bit_depth'* $\geq$ *1*, *'bit_depth'* $\leq$ *8*.

'class_selection': Method for the class selection for the LUT. Can be modified to control the accuracy and the runtime needed to create the LUT classifier. The value in *'class_selection'* is ignored if the bit depth of the LUT is maximal, thus *'bit_depth'* = *8* holds. If the bit depth of the LUT is smaller (*'bit_depth'* $<$ *8*), the classes of multiple pixel combinations will be mapped to the same LUT entry. One of these clusters contains $2^{NumComponents \cdot (8-bit\_depth)}$ pixel combinations, where NumComponents denotes the dimension of the LUT, which is specified in create_class_svm. By choosing *'class_selection'* = *'best'*, the class that appears most often in the cluster is stored in the LUT. For *'class_selection'* = *'fast'*, only one pixel of the cluster, i.e., the pixel with the smallest value (component-wise), is classified. The returned class is stored in the LUT. In this case, the accuracy of the subsequent classification could become lower. On the other hand, the runtime needed to create the LUT can be reduced, which is proportional to the maximal needed storage of the LUT, which is defined with $2^{NumComponents \cdot bit\_depth+2}$.

List of values: *'fast'*, *'best'*

Default: *'fast'*

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **GenParamName** (input_control) ..... attribute.name-array ⇝ *string*
Names of the generic parameters that can be adjusted for the LUT classifier creation.
Default: []
Suggested values: GenParamName ∈ {'bit_depth', 'class_selection'}

▷ **GenParamValue** (input_control) ..... attribute.value-array ⇝ *string* / integer
Values of the generic parameters that can be adjusted for the LUT classifier creation.
Default: []
Suggested values: GenParamValue ∈ {8, 7, 6, 'fast', 'best'}

▷ **ClassLUTHandle** (output_control) ..... class_lut ⇝ *handle*
Handle of the LUT classifier.

Result

If the parameters are valid, the operator create_class_lut_svm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

train_class_svm, read_class_svm

Possible Successors

classify_image_class_lut

Alternatives

create_class_lut_gmm, create_class_lut_knn, create_class_lut_mlp

See also

classify_image_class_lut, clear_class_lut

Module

Foundation

7.4 Misc

add_sample_class_train_data (: : ClassTrainDataHandle, Order, Features, ClassID :)

Add a training sample to training data.

add_sample_class_train_data adds a training sample to the training data given by ClassTrainDataHandle. The training sample is given by Features and ClassID. Features is the feature vector of the sample, and consequently must be a real vector of length NumDim, as specified in create_class_train_data. ClassID is the class of the sample. More than one training sample can be added at once. In this case the parameter Order defines in which order the elements of the feature vectors are passed in Features. If it is set to *'row'*, the first training sample comes first, the second comes second, and so on. If it is set to *'column'*, the first dimension of all feature vectors comes first, and then the second dimension of all feature vectors, and so on. The third possible mode for Order is *'feature_column'*. This mode expects features which were grouped before with set_feature_lengths_class_train_data to come completely and row-wise before the second feature, and so on.

Parameters

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data.

▷ **Order** (input_control) . . . . . string ⇝ *string*
The order of the feature vector.
Default: 'row'
List of values: Order ∈ {'row', 'column', 'feature_column'}

▷ **Features** (input_control) . . . . . number-array ⇝ *real*
Feature vector of the training sample.

▷ **ClassID** (input_control) . . . . . integer-array ⇝ *integer*
Class of the training sample.

Result

If the parameters are valid, the operator add_sample_class_train_data returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ClassTrainDataHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_class_train_data`

Possible Successors

`add_class_train_data_svm`, `add_class_train_data_knn`, `add_class_train_data_gmm`, `add_class_train_data_mlp`

See also

`create_class_train_data`

Module

Foundation

clear_class_train_data (: : ClassTrainDataHandle :)

Clears training data for classifiers.

`clear_class_train_data` clears the training data given in `ClassTrainDataHandle`. After calling `clear_class_train_data`, `ClassTrainDataHandle` becomes invalid.

Parameters

▷ **ClassTrainDataHandle** (input_control) ..... class_train_data ⇝ *handle*
Handle of training data for a classifier.

Result

If the parameters are valid, the operator `clear_class_train_data` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ClassTrainDataHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_class_train_data`

See also

`create_class_train_data`

Module

Foundation

create_class_train_data (: : NumDim : ClassTrainDataHandle)

Create a handle for training data for classifiers.

`create_class_train_data` creates a handle for training data for classifiers. The handle is returned in `ClassTrainDataHandle`. The dimension of the feature vectors is specified with `NumDim`. Only feature vectors of this length can be added to the handle.

Parameters

▷ **NumDim** (input_control) . . . . . number ⇝ *integer*
Number of dimensions of the feature vector.
Default: 10

▷ **ClassTrainDataHandle** (output_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data.

Example

```
* Find out which of the two features distinguishes two Classes
NameFeature1 := 'Good Feature'
NameFeature2 := 'Bad Feature'
LengthFeature1 := 3
LengthFeature2 := 2
* Create training data
create_class_train_data (LengthFeature1+LengthFeature2,\
  ClassTrainDataHandle)
* Define the features which are in the training data
set_feature_lengths_class_train_data (ClassTrainDataHandle, [LengthFeature1,\
  LengthFeature2], [NameFeature1, NameFeature2])
* Add training data
*                                                        |Feat1| |Feat2|
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  2,1  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  2,1  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  3,4  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  3,4  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [0,0,1,  5,6  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,3,2,  5,6  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [0,0,1,  5,6  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,3,2,  5,6  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [0,0,1,  5,6  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,3,2,  5,6  ], 1)
* Add more data
* ...
* Select the better feature with the classifier of your choice
select_feature_set_knn (ClassTrainDataHandle, 'greedy', [], [], KNNHandle,\
  SelectedFeature, Score)
select_feature_set_svm (ClassTrainDataHandle, 'greedy', [], [], SVMHandle,\
  SelectedFeature, Score)
select_feature_set_mlp (ClassTrainDataHandle, 'greedy', [], [], MLPHandle,\
  SelectedFeature, Score)
select_feature_set_gmm (ClassTrainDataHandle, 'greedy', [], [], GMMHandle,\
  SelectedFeature, Score)
* Use the classifier
* ...
```

Result

If the parameters are valid, the operator `create_class_train_data` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

add_sample_class_knn, train_class_knn

Alternatives

create_class_svm, create_class_mlp

See also

select_feature_set_knn, read_class_knn

Module

Foundation

deserialize_class_train_data (: : SerializedItemHandle : ClassTrainDataHandle)

Deserialize serialized training data for classifiers.

deserialize_class_train_data deserializes training data for classifiers, that were serialized by serialize_class_train_data (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized training data is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created training data block with the handle ClassTrainDataHandle.

Parameters

- ▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
  Handle of the serialized item.
- ▷ **ClassTrainDataHandle** (output_control) ..... class_train_data ⇝ *handle*
  Handle of the training data.

Result

If the parameters are valid, the operator deserialize_class_train_data returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

serialize_class_train_data

Possible Successors

fwrite_serialized_item

See also

create_class_train_data

Module

Foundation

get_sample_class_train_data (: : ClassTrainDataHandle, IndexSample : Features, ClassID)

Return a training sample from training data.

get_sample_class_train_data reads a training sample from the training data given by ClassTrainDataHandle that was added, e.g., with add_sample_class_train_data. The index of the sample is specified with IndexSample. The index is counted from 0. That means that IndexSample must be a number between 0 and NumSamples − 1, where NumSamples can be determined with get_sample_num_class_train_data. The training sample is returned in Features

and ClassID. Features is a feature vector of length NumDim (see create_class_train_data) and ClassID is the class of the feature vector.

Parameters

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Handle of training data for a classifier.

▷ **IndexSample** (input_control) . . . . . integer ⇝ *integer*
Number of stored training sample.

▷ **Features** (output_control) . . . . . real-array ⇝ *real*
Feature vector of the training sample.

▷ **ClassID** (output_control) . . . . . integer ⇝ *integer*
Class of the training sample.

Result

If the parameters are valid, the operator get_sample_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_train_data

See also

create_class_train_data

Module

Foundation

get_sample_num_class_train_data (: : ClassTrainDataHandle : NumSamples)

Return the number of training samples stored in the training data.

get_sample_num_class_train_data returns in NumSamples the number of training samples which are stored in the training data specified by ClassTrainDataHandle.

Parameters

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Handle of training data.

▷ **NumSamples** (output_control) . . . . . integer ⇝ *integer*
Number of stored training samples.

Result

If ClassTrainDataHandle is valid, the operator get_sample_num_class_train_data returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_train_data

Possible Successors

get_sample_class_train_data

See also

create_class_train_data

Module

Foundation

read_class_train_data (: : FileName : ClassTrainDataHandle)

Read the training data for classifiers from a file.

read_class_train_data reads the saved training data for classifiers from the file FileName (see write_class_train_data). The default HALCON file extension for training data for a classifier is 'ctd'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name of the training data.
File extension: .ctd

▷ **ClassTrainDataHandle** (output_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data.

Result

read_class_train_data returns 2 (H_MSG_TRUE). An exception is raised if it was not possible to open the file FileName or the file has the wrong format.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

See also

create_class_train_data, write_class_train_data

Module

Foundation

select_sub_feature_class_train_data (: : ClassTrainDataHandle, SubFeatureIndices : SelectedClassTrainDataHandle)

Select certain features from training data to create training data containing less features.

select_sub_feature_class_train_data selects certain features from the training data in ClassTrainDataHandle and returns the subset in SelectedClassTrainDataHandle. The features that should be selected can be chosen by SubFeatureIndices. If set_feature_lengths_class_train_data was not called before, the indices refer to the columns. If set_feature_lengths_class_train_data was called before, the grouping defined there is relevant for the meaning of the indices. The entry n in the list selects then the n-th feature group. If set_feature_lengths_class_train_data was called with names for the feature groups, those names can be used instead of the indices.

Parameters

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data.

▷ **SubFeatureIndices** (input_control) . . . . . number-array ⇝ *integer* / string
Indices or names to select the subfeatures or columns.

▷ **SelectedClassTrainDataHandle** (output_control) . . . . . class_train_data ⇝ *handle*
Handle of the reduced training data.

Example

```
* Find out which of the two features distinguishes two Classes
NameFeature1 := 'Good Feature'
NameFeature2 := 'Bad Feature'
LengthFeature1 := 3
LengthFeature2 := 2
* Create training data
create_class_train_data (LengthFeature1+LengthFeature2,\
                         ClassTrainDataHandle)
* Define the features which are in the training data
set_feature_lengths_class_train_data (ClassTrainDataHandle, [LengthFeature1,\
                           LengthFeature2], [NameFeature1, NameFeature2])
* Add training data
*                                                          |Feat1| |Feat2|
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  2,1  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  2,1  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  3,4  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  3,4  ], 1)
* Add more data
* ...
* Select one of the features
select_sub_feature_class_train_data (ClassTrainDataHandle, NameFeature1, \
                                     SelectedClassTrainDataHandle)
* Add training data to a classifier
create_class_knn (LengthFeature1, KNNHandle)
add_class_train_data_knn (KNNHandle, SelectedClassTrainDataHandle)
train_class_knn (KNNHandle, [], [])
* Use the classifier
* ...
```

Result

If the parameters are valid, the operator `select_sub_feature_class_train_data` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_class_train_data`, `add_sample_class_train_data`, `set_feature_lengths_class_train_data`

Possible Successors

`add_class_train_data_gmm`, `add_class_train_data_mlp`, `add_class_train_data_svm`, `add_class_train_data_knn`

Module

Foundation

serialize_class_train_data (: : ClassTrainDataHandle : SerializedItemHandle)

Serialize training data for classifiers.

`serialize_class_train_data` serializes training data for classifiers and its stored training samples (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The same data that

is written in a file by write_class_train_data is converted to a serialized item. The training data is defined by the handle ClassTrainDataHandle. The serialized training data is returned by the handle SerializedItemHandle and can be deserialized by deserialize_class_train_data.

Parameters

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_class_train_data returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

deserialize_class_train_data

See also

create_class_train_data, read_class_train_data

Module

Foundation

set_feature_lengths_class_train_data (: : ClassTrainDataHandle, SubFeatureLength, Names :)

Define subfeatures in training data.

set_feature_lengths_class_train_data defines subfeatures in the training data in ClassTrainDataHandle. The subfeatures are defined in SubFeatureLength by a set of lengths that groups the previously added columns subsequently into subfeatures. It is not possible to group columns which are not subsequent. The sum over all entries in SubFeatureLength must be equal to the number of dimensions set in create_class_train_data with the parameter NumDim. Optionally, names for all subsets can be defined in Names.

An exemplary situation in which this operator is helpful is described here: Two different data sources are available. Both data sources provide a vector of a certain length. The first data source provides data of length n and the second of length m. In order to automatically decide which of the data sources is more valuable for a certain classification problem, training data can be created that contains both data sources. E.g., if create_class_train_data was called with NumDim $= n + m = w$, then set_feature_lengths_class_train_data can be called with *[n,m]* in SubFeatureLength and *[Name1, Name2]* in Names to describe this situation for a later usage of operators like select_feature_set_knn or select_feature_set_svm. Then the classification problem has to be specified via calls of add_sample_class_train_data, by giving a vector of the first data source and a vector of the second data source as the combined feature vector of length w. The result of the call of select_feature_set_knn would then be either *[Name1]* if the first is more relevant, *[Name2]* if the second is more relevant or *[Name1, Name2]* if both are necessary.

Parameters

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data that should be partitioned into subfeatures.

▷ **SubFeatureLength** (input_control) . . . . . number-array ⇝ *integer*
Length of the subfeatures.

▷ **Names** (input_control) . . . . . string-array ⇝ *string*
Names of the subfeatures.

Example

```
* Find out which of the two features distinguishes two Classes
NameFeature1 := 'Good Feature'
NameFeature2 := 'Bad Feature'
LengthFeature1 := 3
LengthFeature2 := 2
* Create training data
create_class_train_data (LengthFeature1+LengthFeature2,\
  ClassTrainDataHandle)
* Define the features which are in the training data
set_feature_lengths_class_train_data (ClassTrainDataHandle, [LengthFeature1,\
  LengthFeature2], [NameFeature1, NameFeature2])
* Add training data
*                                                          |Feat1| |Feat2|
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  2,1  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  2,1  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  3,4  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  3,4  ], 1)
* Add more data
* ...
* Select the better feature
select_feature_set_knn (ClassTrainDataHandle, 'greedy', [], [], KNNHandle,\
  SelectedFeature, Score)
classify_class_knn (KNNHandle, [1,1,1], Result, Rating)
classify_class_knn (KNNHandle, [2,2,2], Result, Rating)
* Use the classifier
* ...
```

Result

If the parameters are valid, the operator `set_feature_lengths_class_train_data` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ClassTrainDataHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_class_train_data`, `add_sample_class_train_data`

Possible Successors

`select_feature_set_knn`, `select_feature_set_svm`, `select_feature_set_mlp`, `select_feature_set_gmm`

Module

Foundation

`write_class_train_data` (: : `ClassTrainDataHandle`, `FileName` :)

Save the training data for classifiers in a file.

write_class_train_data writes the training data for classifiers ClassTrainDataHandle to the file given by FileName. The classifier can be read again with read_class_train_data. The default HALCON file extension for the training data is 'ctd'.

Parameters

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the file in which the training data will be written.
File extension: .ctd

Result

write_class_train_data returns 2 (H_MSG_TRUE). An exception is raised if it was not possible to open file FileName.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

create_class_train_data, read_class_train_data

Module

Foundation

7.5 Neural Nets

```
add_class_train_data_mlp ( : : MLPHandle,
    ClassTrainDataHandle : )
```

Add training data to a multilayer perceptron (MLP).

add_class_train_data_mlp adds the training data specified by ClassTrainDataHandle to a multilayer perceptron (MLP) specified by MLPHandle.

Parameters

▷ **MLPHandle** (input_control) . . . . . class_mlp ⇝ *handle*
MLP handle which receives the training data.

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Training data for a classifier.

Result

If the parameters are valid, the operator add_class_train_data_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MLPHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_mlp, create_class_train_data

Possible Successors

get_sample_class_mlp

Alternatives

add_sample_class_mlp

See also

create_class_mlp

Module

Foundation

add_sample_class_mlp (: : MLPHandle, Features, Target :)

Add a training sample to the training data of a multilayer perceptron.

add_sample_class_mlp adds a training sample to the multilayer perceptron (MLP) given by MLPHandle. The training sample is given by Features and Target. Features is the feature vector of the sample, and consequently must be a real vector of length NumInput, as specified in create_class_mlp. Target is the target vector of the sample, which must have the length NumOutput (see create_class_mlp) for all three types of activation functions of the MLP (exception: see below). If the MLP is used for regression (function approximation), i.e., if OutputFunction = *'linear'*, Target is the value of the function at the coordinate Features. In this case, Target can contain arbitrary real numbers. For OutputFunction = *'logistic'*, Target can only contain the values *0.0* and *1.0*. A value of *1.0* specifies that the attribute in question is present, while a value of *0.0* specifies that the attribute is absent. Because in this case the attributes are independent, arbitrary combinations of *0.0* and *1.0* can be passed. For OutputFunction = *'softmax'*, Target also can only contain the values *0.0* and *1.0*. In contrast to OutputFunction = *'logistic'*, the value *1.0* must be present for exactly one element of the tuple Target. The location in the tuple designates the class of the sample. For ease of use, a single integer value may be passed if OutputFunction = *'softmax'*. This value directly designates the class of the sample, which is counted from 0, i.e., the class must be an integer between 0 and NumOutput − 1. The class is converted to a target vector of length NumOutput internally.

Before the MLP can be trained with train_class_mlp, *all* training samples must be added to the MLP with add_sample_class_mlp.

The number of currently stored training samples can be queried with get_sample_num_class_mlp. Stored training samples can be read out again with get_sample_class_mlp.

Normally, it is useful to save the training samples in a file with write_samples_class_mlp to facilitate reusing the samples, and to facilitate that, if necessary, new training samples can be added to the data set, and hence to facilitate that a *newly created* MLP can be trained *anew* with the extended data set.

Parameters

▷ **MLPHandle** (input_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

▷ **Features** (input_control) . . . . . real-array ⇝ *real*
Feature vector of the training sample to be stored.

▷ **Target** (input_control) . . . . . number(-array) ⇝ *integer* / real
Class or target vector of the training sample to be stored.

Result

If the parameters are valid, the operator add_sample_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MLPHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_mlp

Possible Successors

train_class_mlp, write_samples_class_mlp

Alternatives

read_samples_class_mlp

See also

clear_samples_class_mlp, get_sample_num_class_mlp, get_sample_class_mlp

Module

Foundation

classify_class_mlp (: : MLPHandle, Features, Num : Class, Confidence)

Calculate the class of a feature vector by a multilayer perceptron.

classify_class_mlp computes the best Num classes of the feature vector Features with the multilayer perceptron (MLP) MLPHandle and returns the classes in Class and the corresponding confidences (probabilities) of the classes in Confidence. Before calling classify_class_mlp, the MLP must be trained with train_class_mlp.

classify_class_mlp can only be called if the MLP is used as a classifier with OutputFunction = *'softmax'* (see create_class_mlp). Otherwise, an error message is returned. classify_class_mlp corresponds to a call to evaluate_class_mlp and an additional step that extracts the best Num classes. As described with evaluate_class_mlp, the output values of the MLP can be interpreted as probabilities of the occurrence of the respective classes. In most cases it should be sufficient to use Num = *1* in order to decide whether the probability of the best class is high enough. In some applications it may be interesting to also take the second best class into account (Num = *2*), particularly if it can be expected that the classes show a significant degree of overlap.

Parameters

- ▷ **MLPHandle** (input_control) . . . . . class_mlp ⇝ *handle*
  MLP handle.
- ▷ **Features** (input_control) . . . . . real-array ⇝ *real*
  Feature vector.
- ▷ **Num** (input_control) . . . . . integer-array ⇝ *integer*
  Number of best classes to determine.
  Default: 1
  Suggested values: Num ∈ {1, 2, 3, 4, 5}
- ▷ **Class** (output_control) . . . . . integer(-array) ⇝ *integer*
  Result of classifying the feature vector with the MLP.
- ▷ **Confidence** (output_control) . . . . . real(-array) ⇝ *real*
  Confidence(s) of the class(es) of the feature vector.

Result

If the parameters are valid, the operator classify_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_mlp, read_class_mlp

Alternatives

apply_dl_classifier, evaluate_class_mlp

See also

create_class_mlp

References

Christopher M. Bishop: "Neural Networks for Pattern Recognition"; Oxford University Press, Oxford; 1995.

Andrew Webb: "Statistical Pattern Recognition"; Arnold, London; 1999.

Module

Foundation

clear_class_mlp (: : MLPHandle :)

Clear a multilayer perceptron.

clear_class_mlp clears the multilayer perceptron (MLP) given by MLPHandle and frees all memory required for the MLP. After calling clear_class_mlp, the MLP can no longer be used. The handle MLPHandle becomes invalid.

Parameters

▷ **MLPHandle** (input_control) ..... class_mlp(-array) ⇝ *handle*
MLP handle.

Result

If MLPHandle is valid, the operator clear_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MLPHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

classify_class_mlp, evaluate_class_mlp

See also

create_class_mlp, read_class_mlp, write_class_mlp, train_class_mlp

Module

Foundation

clear_samples_class_mlp (: : MLPHandle :)

Clear the training data of a multilayer perceptron.

clear_samples_class_mlp clears all training samples that have been added to the multilayer perceptron (MLP) MLPHandle with add_sample_class_mlp or read_samples_class_mlp. clear_samples_class_mlp should only be used if the MLP is trained in the same process that uses the MLP for evaluation with evaluate_class_mlp or for classification with classify_class_mlp. In

this case, the memory required for the training samples can be freed with clear_samples_class_mlp, and hence memory can be saved. In the normal usage, in which the MLP is trained offline and written to a file with write_class_mlp, it is typically unnecessary to call clear_samples_class_mlp because write_class_mlp does not save the training samples, and hence the online process, which reads the MLP with read_class_mlp, requires no memory for the training samples.

Parameters

▷ **MLPHandle** (input_control) . . . . . class_mlp(-array) ⇝ *handle*
MLP handle.

Result

If the parameters are valid, the operator clear_samples_class_mlp returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MLPHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

train_class_mlp, write_samples_class_mlp

See also

create_class_mlp, clear_class_mlp, add_sample_class_mlp, read_samples_class_mlp

Module

Foundation

create_class_mlp (: : NumInput, NumHidden, NumOutput,
OutputFunction, Preprocessing, NumComponents,
RandSeed : MLPHandle)

Create a multilayer perceptron for classification or regression.

create_class_mlp creates a neural net in the form of a multilayer perceptron (MLP), which can be used for classification or regression (function approximation), depending on how OutputFunction is set. The MLP consists of three layers: an input layer with NumInput input variables (units, neurons), a hidden layer with NumHidden units, and an output layer with NumOutput output variables. The MLP performs the following steps to calculate the activations z_j of the hidden units from the input data x_i (the so-called feature vector):

$$\begin{aligned} a_j^{(1)} &= \sum_{i=1}^{n_i} w_{ji}^{(1)} x_i + b_j^{(1)}, \qquad j = 1, \ldots, n_h \\ z_j &= \tanh\left(a_j^{(1)}\right), \qquad j = 1, \ldots, n_h \end{aligned}$$

Here, the matrix $w_{ji}^{(1)}$ and the vector $b_j^{(1)}$ are the weights of the input layer (first layer) of the MLP. In the hidden layer (second layer), the activations z_j are transformed in a first step by using linear combinations of the variables in an analogous manner as above:

$$a_k^{(2)} = \sum_{j=1}^{n_h} w_{kj}^{(2)} z_j + b_k^{(2)}, \qquad k = 1, \ldots, n_o$$

Here, the matrix $w_{kj}^{(2)}$ and the vector $b_k^{(2)}$ are the weights of the second layer of the MLP.

The activation function used in the output layer can be determined by setting OutputFunction. For OutputFunction = *'linear'*, the data are simply copied:

$$y_k = a_k^{(2)}, \qquad k = 1, \ldots, n_o$$

This type of activation function should be used for regression problems (function approximation). This activation function is not suited for classification problems.

For OutputFunction = *'logistic'*, the activations are computed as follows:

$$y_k = \frac{1}{1 + \exp\left(-a_k^{(2)}\right)}, \qquad k = 1, \ldots, n_o$$

This type of activation function should be used for classification problems with multiple (NumOutput) independent logical attributes as output. This kind of classification problem is relatively rare in practice.

For OutputFunction = *'softmax'*, the activations are computed as follows:

$$y_k = \frac{\exp\left(a_k^{(2)}\right)}{\sum_{l=1}^{n_o} \exp\left(a_l^{(2)}\right)}, \qquad k = 1, \ldots, n_o$$

This type of activation function should be used for common classification problems with multiple (NumOutput) mutually exclusive classes as output. In particular, OutputFunction = *'softmax'* must be used for the classification of pixel data with classify_image_class_mlp.

The parameters Preprocessing and NumComponents can be used to specify a preprocessing of the feature vectors. For Preprocessing = *'none'*, the feature vectors are passed unaltered to the MLP. NumComponents is ignored in this case.

For all other values of Preprocessing, the training data set is used to compute a transformation of the feature vectors during the training as well as later in the classification or evaluation.

For Preprocessing = *'normalization'*, the feature vectors are normalized by subtracting the mean of the training vectors and dividing the result by the standard deviation of the individual components of the training vectors. Hence, the transformed feature vectors have a mean of 0 and a standard deviation of 1. The normalization does not change the length of the feature vector. NumComponents is ignored in this case. This transformation can be used if the mean and standard deviation of the feature vectors differs substantially from 0 and 1, respectively, or for data in which the components of the feature vectors are measured in different units (e.g., if some of the data are gray value features and some are region features, or if region features are mixed, e.g., 'circularity' (unit: scalar) and 'area' (unit: pixel squared)). In these cases, the training of the net will typically require fewer iterations than without normalization.

For Preprocessing = *'principal_components'*, a principal component analysis is performed. First, the feature vectors are normalized (see above). Then, an orthogonal transformation (a rotation in the feature space) that decorrelates the training vectors is computed. After the transformation, the mean of the training vectors is 0 and the covariance matrix of the training vectors is a diagonal matrix. The transformation is chosen such that the transformed features that contain the most variation is contained in the first components of the transformed feature vector. With this, it is possible to omit the transformed features in the last components of the feature vector, which typically are mainly influenced by noise, without losing a large amount of information. The parameter NumComponents can be used to determine how many of the transformed feature vector components should be used. Up to NumInput components can be selected. The operator get_prep_info_class_mlp can be used to determine how much information each transformed component contains. Hence, it aids the selection of NumComponents. Like data normalization, this transformation can be used if the mean and standard deviation of the feature vectors differs substantially from 0 and 1, respectively, or for feature vectors in which the components of the data are measured in different units. In addition, this transformation is useful if it can be expected that the features are highly correlated.

In contrast to the above three transformations, which can be used for all MLP types, the transformation specified by Preprocessing = *'canonical_variates'* can only be used if the MLP is used as a classifier with

OutputFunction = *'softmax'*). The computation of the canonical variates is also called linear discriminant analysis. In this case, a transformation that first normalizes the training vectors and then decorrelates the training vectors on average over all classes is computed. At the same time, the transformation maximally separates the mean values of the individual classes. As for Preprocessing = *'principal_components'*, the transformed components are sorted by information content, and hence transformed components with little information content can be omitted. For canonical variates, up to $\min(\text{NumOutput} - 1, \text{NumInput})$ components can be selected. Also in this case, the information content of the transformed components can be determined with get_prep_info_class_mlp. Like principal component analysis, canonical variates can be used to reduce the amount of data without losing a large amount of information, while additionally optimizing the separability of the classes after the data reduction.

For the last two types of transformations (*'principal_components'* and *'canonical_variates'*), the actual number of input units of the MLP is determined by NumComponents, whereas NumInput determines the dimensionality of the input data (i.e., the length of the untransformed feature vector). Hence, by using one of these two transformations, the number of input variables, and thus usually also the number of hidden units can be reduced. With this, the time needed to train the MLP and to evaluate and classify a feature vector is typically reduced.

Usually, NumHidden should be selected in the order of magnitude of NumInput and NumOutput. In many cases, much smaller values of NumHidden already lead to very good classification results. If NumHidden is chosen too large, the MLP may overfit the training data, which typically leads to bad generalization properties, i.e., the MLP learns the training data very well, but does not return very good results on unknown data.

create_class_mlp initializes the above described weights with random numbers. To ensure that the results of training the classifier with train_class_mlp are reproducible, the seed value of the random number generator is passed in RandSeed. If the training results in a relatively large error, it sometimes may be possible to achieve a smaller error by selecting a different value for RandSeed and retraining an MLP.

After the MLP has been created, typically training samples are added to the MLP by repeatedly calling add_sample_class_mlp or read_samples_class_mlp. After this, the MLP is typically trained using train_class_mlp. Hereafter, the MLP can be saved using write_class_mlp. Alternatively, the MLP can be used immediately after training to evaluate data using evaluate_class_mlp or, if the MLP is used as a classifier (i.e., for OutputFunction = *'softmax'*), to classify data using classify_class_mlp.

The training of the MLP will usually result in very sharp boundaries between the different classes, i.e., the confidence for one class will drop from close to 1 (within the region of the class) to close to 0 (within the region of a different class) within a very narrow "band" in the feature space. If the classes do not overlap, this transition happens at a suitable location between the classes; if the classes overlap, the transition happens at a suitable location within the overlapping area. While this sharp transition is desirable in many applications, in some applications a smoother transition between different classes (i.e., a transition within a wider "band" in the feature space) is desirable to reflect a level of uncertainty within the region in the feature space between the classes. Furthermore, as described above, it may be desirable to prevent overfitting of the MLP to the training data. For these purposes, the MLP can be regularized by using set_regularization_params_class_mlp.

An MLP, as defined above, has no inherent capability for novelty detection, i.e., it will classify a random feature vector into one of the classes with a confidence close to 1 (unless the random feature vector happens to lie in a region of the feature space in which the training samples of different classes overlap). In some applications, however, it is desirable to reject feature vectors that do not lie close to any class, where "closesness" defined by the proximity of the feature vector to the collection of feature vectors in the training set. To provide an MLP with the ability for novelty detection, i.e., to reject feature vectors that do not belong to any class, an explicit rejection class can be created by setting NumOutput to the number of actual classes plus 1. Then, set_rejection_params_class_mlp can be used to configure train_class_mlp to automatically generate samples for this rejection class.

The combination of regularization and an automatic generation of a rejection class is useful in many applications since it provides a smooth transition between the actual classes and from the actual classes to the rejection class. This reflects the requirement of these applications that only feature vectors within the area of the feature space that corresponds to the training samples of each class should have a confidence close to 1, whereas random feature vectors not belonging to any class should have a confidence close to 0, and that transitions between the classes should be smooth, reflecting a growing degree of uncertainty the farther a feature vector lies from the respective class. In particular, OCR applications sometimes have this requirement (see create_ocr_class_mlp).

A comparison of the MLP and the support vector machine (SVM) (see create_class_svm) typically shows that SVMs are generally faster at training, especially for huge training sets, and achieve slightly better recognition rates than MLPs. The MLP is faster at classification and should therefore be preferred in time critical applications. Please note that this guideline assumes optimal tuning of the parameters.

Parameters

▷ **NumInput** (input_control) . . . . . integer ⇝ *integer*
Number of input variables (features) of the MLP.
Default: 20
Suggested values: NumInput ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Restriction: NumInput >= 1

▷ **NumHidden** (input_control) . . . . . integer ⇝ *integer*
Number of hidden units of the MLP.
Default: 10
Suggested values: NumHidden ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 150}
Restriction: NumHidden >= 1

▷ **NumOutput** (input_control) . . . . . integer ⇝ *integer*
Number of output variables (classes) of the MLP.
Default: 5
Suggested values: NumOutput ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 150}
Restriction: NumOutput >= 1

▷ **OutputFunction** (input_control) . . . . . string ⇝ *string*
Type of the activation function in the output layer of the MLP.
Default: 'softmax'
List of values: OutputFunction ∈ {'linear', 'logistic', 'softmax'}

▷ **Preprocessing** (input_control) . . . . . string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
Default: 'normalization'
List of values: Preprocessing ∈ {'none', 'normalization', 'principal_components', 'canonical_variates'}

▷ **NumComponents** (input_control) . . . . . integer ⇝ *integer*
Preprocessing parameter: Number of transformed features (ignored for Preprocessing = *'none'* and Preprocessing = *'normalization'*).
Default: 10
Suggested values: NumComponents ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Restriction: NumComponents >= 1

▷ **RandSeed** (input_control) . . . . . integer ⇝ *integer*
Seed value of the random number generator that is used to initialize the MLP with random values.
Default: 42

▷ **MLPHandle** (output_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

Example

```
* Use the MLP for regression (function approximation)
create_class_mlp (1, NumHidden, 1, 'linear', 'none', 1, 42, MLPHandle)
* Generate the training data
* D = [...]
* T = [...]
* Add the training data
for J := 0 to NumData-1 by 1
    add_sample_class_mlp (MLPHandle, D[J], T[J])
endfor
* Train the MLP
train_class_mlp (MLPHandle, 200, 0.001, 0.001, Error, ErrorLog)
* Generate test data
* X = [...]
* Compute the output of the MLP on the test data
for J := 0 to N-1 by 1
    evaluate_class_mlp (MLPHandle, X[J], Y)
endfor

* Use the MLP for classification
create_class_mlp (NumIn, NumHidden, NumOut, 'softmax', \
```

```
                  'normalization', NumIn, 42, MLPHandle)
* Generate and add the training data
for J := 0 to NumData-1 by 1
    * Generate training features and classes
    * Data = [...]
    * Class = [...]
    add_sample_class_mlp (MLPHandle, Data, Class)
endfor
* Train the MLP
train_class_mlp (MLPHandle, 100, 1, 0.01, Error, ErrorLog)
* Use the MLP to classify unknown data
for J := 0 to N-1 by 1
    * Extract features
    * Features = [...]
    classify_class_mlp (MLPHandle, Features, 1, Class, Confidence)
endfor
```

Result

If the parameters are valid, the operator `create_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`add_sample_class_mlp`, `set_regularization_params_class_mlp`, `set_rejection_params_class_mlp`

Alternatives

`read_dl_classifier`, `create_class_svm`, `create_class_gmm`

See also

`clear_class_mlp`, `train_class_mlp`, `classify_class_mlp`, `evaluate_class_mlp`

References

Christopher M. Bishop: “Neural Networks for Pattern Recognition”; Oxford University Press, Oxford; 1995.

Andrew Webb: “Statistical Pattern Recognition”; Arnold, London; 1999.

Module

Foundation

deserialize_class_mlp (: : SerializedItemHandle : MLPHandle)

Deserialize a serialized multilayer perceptron.

`deserialize_class_mlp` deserializes a multilayer perceptron (MLP) (including its training samples), that was serialized by `serialize_class_mlp` (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The serialized multilayer perceptron is defined by the handle `SerializedItemHandle`. The deserialized values are stored in an automatically created multilayer perceptron with the handle `MLPHandle`.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **MLPHandle** (output_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

Result

If the parameters are valid, the operator `deserialize_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_class_mlp`

Possible Successors

`classify_class_mlp`, `evaluate_class_mlp`, `create_class_lut_mlp`

See also

`create_class_mlp`, `write_class_mlp`, `serialize_class_mlp`

Module

Foundation

evaluate_class_mlp (: : MLPHandle, Features : Result)

Calculate the evaluation of a feature vector by a multilayer perceptron.

`evaluate_class_mlp` computes the result `Result` of evaluating the feature vector `Features` with the multilayer perceptron (MLP) `MLPHandle`. The formulas used for the evaluation are described with `create_class_mlp`. Before calling `evaluate_class_mlp`, the MLP must be trained with `train_class_mlp`.

If the MLP is used for regression (function approximation), i.e., if (`OutputFunction` = *'linear'*), `Result` is the value of the function at the coordinate `Features`. For `OutputFunction` = *'logistic'* and *'softmax'*, the values in `Result` can be interpreted as probabilities. Hence, for `OutputFunction` = *'logistic'* the elements of `Result` represent the probabilities of the presence of the respective independent attributes. Typically, a threshold of 0.5 is used to decide whether the attribute is present or not. Depending on the application, other thresholds may be used as well. For `OutputFunction` = *'softmax'* usually the position of the maximum value of `Result` is interpreted as the class of the feature vector, and the corresponding value as the probability of the class. In this case, `classify_class_mlp` should be used instead of `evaluate_class_mlp` because `classify_class_mlp` directly returns the class and corresponding probability.

Parameters

▷ **MLPHandle** (input_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

▷ **Features** (input_control) . . . . . real-array ⇝ *real*
Feature vector.

▷ **Result** (output_control) . . . . . real-array ⇝ *real*
Result of evaluating the feature vector with the MLP.

Result

If the parameters are valid, the operator `evaluate_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_mlp, read_class_mlp

Alternatives

classify_class_mlp

See also

create_class_mlp

References

Christopher M. Bishop: "Neural Networks for Pattern Recognition"; Oxford University Press, Oxford; 1995.

Andrew Webb: "Statistical Pattern Recognition"; Arnold, London; 1999.

Module

Foundation

get_class_train_data_mlp (: : MLPHandle : ClassTrainDataHandle)

Get the training data of a multilayer perceptron (MLP).

get_class_train_data_mlp gets the training data of a multilayer perceptron (MLP) and returns it in ClassTrainDataHandle.

Parameters

▷ **MLPHandle** (input_control) ..... class_mlp ⇝ *handle*
Handle of a MLP that contains training data.

▷ **ClassTrainDataHandle** (output_control) ..... class_train_data ⇝ *handle*
Handle of the training data of the classifier.

Result

If the parameters are valid, the operator get_class_train_data_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

add_sample_class_mlp, read_samples_class_mlp

Possible Successors

add_class_train_data_svm, add_class_train_data_gmm, add_class_train_data_knn

See also

create_class_train_data

Module

Foundation

get_params_class_mlp (: : MLPHandle : NumInput, NumHidden, NumOutput, OutputFunction, Preprocessing, NumComponents)

Return the parameters of a multilayer perceptron.

get_params_class_mlp returns the parameters of a multilayer perceptron (MLP) that were specified when the MLP was created with create_class_mlp. This is particularly useful if the MLP was read from a file with read_class_mlp. The output of get_params_class_mlp can, for example, be used to check whether the feature vectors and, if necessary, the target data to be used with the MLP have the correct lengths. For a description of the parameters, see create_class_mlp.

Parameters

- ▷ **MLPHandle** (input_control) ..... class_mlp ⇝ *handle*
  MLP handle.
- ▷ **NumInput** (output_control) ..... integer ⇝ *integer*
  Number of input variables (features) of the MLP.
- ▷ **NumHidden** (output_control) ..... integer ⇝ *integer*
  Number of hidden units of the MLP.
- ▷ **NumOutput** (output_control) ..... integer ⇝ *integer*
  Number of output variables (classes) of the MLP.
- ▷ **OutputFunction** (output_control) ..... string ⇝ *string*
  Type of the activation function in the output layer of the MLP.
- ▷ **Preprocessing** (output_control) ..... string ⇝ *string*
  Type of preprocessing used to transform the feature vectors.
- ▷ **NumComponents** (output_control) ..... integer ⇝ *integer*
  Preprocessing parameter: Number of transformed features.

Result

If the parameters are valid, the operator get_params_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_class_mlp, read_class_mlp

Possible Successors

add_sample_class_mlp, train_class_mlp

See also

evaluate_class_mlp, classify_class_mlp

Module

Foundation

get_prep_info_class_mlp (: : MLPHandle, Preprocessing : InformationCont, CumInformationCont)

Compute the information content of the preprocessed feature vectors of a multilayer perceptron.

get_prep_info_class_mlp computes the information content of the training vectors that have been transformed with the preprocessing given by Preprocessing. Preprocessing can be set to *'principal_components'* or *'canonical_variates'*. The preprocessing methods are described with create_class_mlp. The information content is derived from the variations of the transformed components of the feature vector, i.e., it is computed solely based on the training data, independent of any error rate on the training data. The information content is computed for all relevant components of the transformed feature vectors (NumInput for *'principal_components'* and $\min(\text{NumOutput}-1, \text{NumInput})$ for *'canonical_variates'*, see create_class_mlp), and is returned in InformationCont as a number between 0 and 1. To convert the information content into a percentage, it simply needs to be multiplied by 100. The cumulative information content of the first n components is returned in the n-th component of CumInformationCont, i.e., CumInformationCont contains

the sums of the first n elements of InformationCont. To use get_prep_info_class_mlp, a sufficient number of samples must be added to the multilayer perceptron (MLP) given by MLPHandle by using add_sample_class_mlp or read_samples_class_mlp.

InformationCont and CumInformationCont can be used to decide how many components of the transformed feature vectors contain relevant information. An often used criterion is to require that the transformed data must represent x% (e.g., 90%) of the data. This can be decided easily from the first value of CumInformationCont that lies above x%. The number thus obtained can be used as the value for NumComponents in a new call to create_class_mlp. The call to get_prep_info_class_mlp already requires the creation of an MLP, and hence the setting of NumComponents in create_class_mlp to an initial value. However, if get_prep_info_class_mlp is called it is typically not known how many components are relevant, and hence how to set NumComponents in this call. Therefore, the following two-step approach should typically be used to select NumComponents: In a first step, an MLP with the maximum number for NumComponents is created (NumInput for *'principal_components'* and min(NumOutput − 1NumInput) for *'canonical_variates'*). Then, the training samples are added to the MLP and are saved in a file using write_samples_class_mlp. Subsequently, get_prep_info_class_mlp is used to determine the information content of the components, and with this NumComponents. After this, a new MLP with the desired number of components is created, and the training samples are read with read_samples_class_mlp. Finally, the MLP is trained with train_class_mlp.

Parameters

▷ **MLPHandle** (input_control) ..... class_mlp ⇝ *handle*
MLP handle.

▷ **Preprocessing** (input_control) ..... string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
Default: 'principal_components'
List of values: Preprocessing ∈ {'principal_components', 'canonical_variates'}

▷ **InformationCont** (output_control) ..... real-array ⇝ *real*
Relative information content of the transformed feature vectors.

▷ **CumInformationCont** (output_control) ..... real-array ⇝ *real*
Cumulative information content of the transformed feature vectors.

Example

```
* Create the initial MLP
create_class_mlp (NumIn, NumHidden, NumOut, 'softmax', \
                  'principal_components', NumIn, 42, MLPHandle)
* Generate and add the training data
for J := 0 to NumData-1 by 1
    * Generate training features and classes
    * Data = [...]
    * Class = [...]
    add_sample_class_mlp (MLPHandle, Data, Class)
endfor
write_samples_class_mlp (MLPHandle, 'samples.mtf')
* Compute the information content of the transformed features
get_prep_info_class_mlp (MLPHandle, 'principal_components',\
                         InformationCont, CumInformationCont)
* Determine NumComp by inspecting InformationCont and CumInformationCont
* NumComp = [...]
* Create the actual MLP
create_class_mlp (NumIn, NumHidden, NumOut, 'softmax', \
                  'principal_components', NumComp, 42, MLPHandle)
* Train the MLP
read_samples_class_mlp (MLPHandle, 'samples.mtf')
train_class_mlp (MLPHandle, 100, 1, 0.01, Error, ErrorLog)
write_class_mlp (MLPHandle, 'classifier.mlp')
```

Result

If the parameters are valid, the operator get_prep_info_class_mlp returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

get_prep_info_class_mlp may return the error 9211 (Matrix is not positive definite) if Preprocessing = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_mlp, read_samples_class_mlp

Possible Successors

clear_class_mlp, create_class_mlp

References

Christopher M. Bishop: "Neural Networks for Pattern Recognition"; Oxford University Press, Oxford; 1995.

Andrew Webb: "Statistical Pattern Recognition"; Arnold, London; 1999.

Module

Foundation

get_regularization_params_class_mlp (: : MLPHandle, GenParamName : GenParamValue)

Return the regularization parameters of a multilayer perceptron.

get_regularization_params_class_mlp returns the regularization parameters of a multilayer perceptron (MLP) that were specified with set_regularization_params_class_mlp. Furthermore, get_regularization_params_class_mlp returns the parameters that were determined by an automatic determination of the regularization parameters. For a description of the parameters, see set_regularization_params_class_mlp.

Parameters

▷ **MLPHandle** (input_control) ..... class_mlp ⇝ *handle*
MLP handle.

▷ **GenParamName** (input_control) ..... string ⇝ *string*
Name of the regularization parameter to return.
Default: 'weight_prior'
List of values: GenParamName ∈ {'weight_prior', 'noise_prior', 'num_well_determined_params', 'fraction_well_determined_params', 'num_outer_iterations', 'num_inner_iterations'}

▷ **GenParamValue** (output_control) ..... number(-array) ⇝ *real* / integer
Value of the regularization parameter.

Result

If the parameters are valid, the operator get_regularization_params_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_regularization_params_class_mlp, read_class_mlp

Possible Successors

train_class_mlp

Module

Foundation

get_rejection_params_class_mlp (: : MLPHandle, GenParamName : GenParamValue)

Get the parameters of a rejection class.

get_rejection_params_class_mlp returns the rejection class parameters of a multilayer perceptron (MLP) that were specified with set_rejection_params_class_mlp. For a description of the parameters, see set_rejection_params_class_mlp.

Parameters

▷ **MLPHandle** (input_control) ..... class_mlp ⇝ *handle*
MLP handle.

▷ **GenParamName** (input_control) ..... string(-array) ⇝ *string*
Names of the generic parameters to return.
Default: 'sampling_strategy'
List of values: GenParamName ∈ {'sampling_strategy', 'hyperbox_tolerance', 'rejection_sample_factor', 'random_seed', 'rejection_class_index'}

▷ **GenParamValue** (output_control) ..... string(-array) ⇝ *string* / real / integer
Values of the generic parameters.

Result

If the parameters are valid, the operator get_rejection_params_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_class_mlp

Possible Successors

train_class_mlp

Module

Foundation

get_sample_class_mlp (: : MLPHandle, IndexSample : Features, Target)

Return a training sample from the training data of a multilayer perceptron.

get_sample_class_mlp reads out a training sample from the multilayer perceptron (MLP) given by MLPHandle that was added with add_sample_class_mlp or read_samples_class_mlp. The index of the sample is specified with IndexSample. The index is counted from 0, i.e., IndexSample must be a number between 0 and NumSamples − 1, where NumSamples can be determined with get_sample_num_class_mlp. The training sample is returned in Features and Target. Features is a feature vector of length NumInput, while Target is a target vector of length NumOutput (see add_sample_class_mlp and create_class_mlp).

get_sample_class_mlp can, for example, be used to reclassify the training data with classify_class_mlp in order to determine which training samples, if any, are classified incorrectly.

Parameters

▷ **MLPHandle** (input_control) ..... class_mlp ⇝ *handle*
MLP handle.

▷ **IndexSample** (input_control) ..... integer ⇝ *integer*
Number of stored training sample.

▷ **Features** (output_control) . . . . . real-array ⇝ *real*
Feature vector of the training sample.

▷ **Target** (output_control) . . . . . real-array ⇝ *real*
Target vector of the training sample.

Example

```
* Train an MLP
create_class_mlp (NumIn, NumHidden, NumOut, 'softmax', \
                  'canonical_variates', NumComp, 42, MLPHandle)
read_samples_class_mlp (MLPHandle, 'samples.mtf')
train_class_mlp (MLPHandle, 100, 1, 0.01, Error, ErrorLog)
* Reclassify the training samples
get_sample_num_class_mlp (MLPHandle, NumSamples)
for I := 0 to NumSamples-1 by 1
    get_sample_class_mlp (MLPHandle, I, Data, Target)
    classify_class_mlp (MLPHandle, Data, 1, Class, Confidence)
    Result := gen_tuple_const(NumOut,0)
    Result[Class] := 1
    Diffs := Target-Result
    if (sum(fabs(Diffs)) > 0)
        * Sample has been classified incorrectly
    endif
endfor
```

Result

If the parameters are valid, the operator `get_sample_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`add_sample_class_mlp`, `read_samples_class_mlp`, `get_sample_num_class_mlp`

Possible Successors

`classify_class_mlp`, `evaluate_class_mlp`

See also

`create_class_mlp`

Module

Foundation

get_sample_num_class_mlp (: : MLPHandle : NumSamples)

Return the number of training samples stored in the training data of a multilayer perceptron.

`get_sample_num_class_mlp` returns in `NumSamples` the number of training samples that are stored in the multilayer perceptron (MLP) given by `MLPHandle`. `get_sample_num_class_mlp` should be called before the individual training samples are accessed with `get_sample_class_mlp`, e.g., for the purpose of reclassifying the training data (see `get_sample_class_mlp`).

Parameters

▷ **MLPHandle** (input_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

▷ **NumSamples** (output_control) . . . . . integer ⇝ *integer*
Number of stored training samples.

Result

If MLPHandle is valid, the operator get_sample_num_class_mlp returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_mlp, read_samples_class_mlp

Possible Successors

get_sample_class_mlp

See also

create_class_mlp

Module

Foundation

read_class_mlp (: : FileName : MLPHandle)

Read a multilayer perceptron from a file.

read_class_mlp reads a multilayer perceptron (MLP) that has been stored with write_class_mlp. Since the training of an MLP can consume a relatively long time, the MLP is typically trained in an offline process and written to a file with write_class_mlp. In the online process the MLP is read with read_class_mlp and subsequently used for evaluation with evaluate_class_mlp or for classification with classify_class_mlp. The default HALCON file extension for the MLP classifier is 'gmc'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .gmc

▷ **MLPHandle** (output_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

Result

If the parameters are valid, the operator read_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

classify_class_mlp, evaluate_class_mlp, create_class_lut_mlp

Alternatives

read_dl_classifier

See also

create_class_mlp, write_class_mlp

Module

Foundation

read_samples_class_mlp (: : MLPHandle, FileName :)

Read the training data of a multilayer perceptron from a file.

read_samples_class_mlp reads training samples from the file given by FileName and adds them to the training samples that have already been added to the multilayer perceptron (MLP) given by MLPHandle. The MLP must be created with create_class_mlp before calling read_samples_class_mlp. As described with train_class_mlp and write_samples_class_mlp, the operators read_samples_class_mlp, add_sample_class_mlp, and write_samples_class_mlp can be used to build up a extensive set of training samples, and hence to improve the performance of the MLP by retraining the MLP with extended data sets.

It should be noted that the training samples must have the correct dimensionality. The feature vectors and target vectors stored in FileName must have the lengths NumInput and NumOutput that were specified with create_class_mlp. If this is not the case an error message is returned.

Parameters

▷ **MLPHandle** (input_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.

Result

If the parameters are valid, the operator read_samples_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MLPHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_mlp

Possible Successors

train_class_mlp

Alternatives

add_sample_class_mlp

See also

write_samples_class_mlp, clear_samples_class_mlp

Module

Foundation

select_feature_set_mlp (: : ClassTrainDataHandle, SelectionMethod, GenParamName, GenParamValue : MLPHandle, SelectedFeatureIndices, Score)

Selects an optimal combination of features to classify the provided data.

select_feature_set_mlp selects an optimal subset from a set of features to solve a given classification problem. The classification problem has to be specified with annotated training data in ClassTrainDataHandle and will be classified by a Multilayer Perceptron. Details of the properties of this classifier can be found in create_class_mlp.

The result of the operator is a trained classifier that is returned in MLPHandle. Additionally, the list of indices or names of the selected features is returned in SelectedFeatureIndices. To use this classifier, calculate for new input data all features mentioned in SelectedFeatureIndices and pass them to the classifier.

A possible application of this operator can be a comparison of different parameter sets for certain feature extraction techniques. Another application is to search for a feature that is discriminating between different classes.

To define the features that should be selected from ClassTrainDataHandle, the dimensions of the feature vectors in ClassTrainDataHandle can be grouped into subfeatures by calling set_feature_lengths_class_train_data. A subfeature can contain several subsequent elements of a feature vector. select_feature_set_mlp decides for each of these subfeatures, if it is better to use it for the classification or leave it out.

The indices of the selected subfeatures are returned in SelectedFeatureIndices. If names were set in set_feature_lengths_class_train_data, these names are returned instead of the indices. If set_feature_lengths_class_train_data was not called for ClassTrainDataHandle before, each element of the feature vector is considered as a subfeature.

The selection method SelectionMethod is either a greedy search *'greedy'* (iteratively add the feature with highest gain) or the dynamically oscillating search *'greedy_oscillating'* (add the feature with highest gain and test then if any of the already added features can be left out without great loss). The method *'greedy'* is generally preferable, since it is faster. Only in cases when the subfeatures are low-dimensional or redundant, the method *'greedy_oscillating'* should be chosen.

The optimization criterion is the classification rate of a two-fold cross-validation of the training data. The best achieved value is returned in Score.

With GenParamName and GenParamValue the number of hidden layer can be set by *'num_hidden'*. The default value is *80*. Larger values for this parameter lead to longer classification times, while it allows a more expressive classifier.

Attention

This operator may take considerable time, depending on the size of the data and the number of features.

Please note, that this operator should not be called, if only a small set of training data is available. Due to the risk of overfitting the operator select_feature_set_mlp may deliver a classifier with a very high score. However, the classifier may perform poorly when tested.

Parameters

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data.

▷ **SelectionMethod** (input_control) . . . . . string ⇝ *string*
Method to perform the selection.
Default: 'greedy'
List of values: SelectionMethod ∈ {'greedy', 'greedy_oscillating'}

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Names of generic parameters to configure the selection process and the classifier.
Default: []
List of values: GenParamName ∈ {'num_hidden'}

▷ **GenParamValue** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Values of generic parameters to configure the selection process and the classifier.
Default: []
Suggested values: GenParamValue ∈ {50, 80, 100}

▷ **MLPHandle** (output_control) . . . . . class_mlp ⇝ *handle*
A trained MLP classifier using only the selected features.

▷ **SelectedFeatureIndices** (output_control) . . . . . string-array ⇝ *string*
The selected feature set, contains indices referring.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
The achieved score using two-fold cross-validation.

Example

```
* Find out which of the two features distinguishes two Classes
NameFeature1 := 'Good Feature'
NameFeature2 := 'Bad Feature'
LengthFeature1 := 3
LengthFeature2 := 2
* Create training data
create_class_train_data (LengthFeature1+LengthFeature2,\
  ClassTrainDataHandle)
* Define the features which are in the training data
set_feature_lengths_class_train_data (ClassTrainDataHandle, [LengthFeature1,\
  LengthFeature2], [NameFeature1, NameFeature2])
* Add training data
*                                                        |Feat1| |Feat2|
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  2,1  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  2,1  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  3,4  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  3,4  ], 1)
* Add more data
* ...
* Select the better feature with a MLP
select_feature_set_mlp (ClassTrainDataHandle, 'greedy', [], [], MLPHandle,\
  SelectedFeatureMLP, Score)
* Use the classifier
* ...
```

Result

If the parameters are valid, the operator `select_feature_set_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`create_class_train_data`, `add_sample_class_train_data`, `set_feature_lengths_class_train_data`

Possible Successors

`classify_class_mlp`

Alternatives

`select_feature_set_knn`, `select_feature_set_svm`, `select_feature_set_gmm`

See also

`select_feature_set_trainf_mlp`, `gray_features`, `region_features`

Module

Foundation

serialize_class_mlp (: : MLPHandle : SerializedItemHandle)

Serialize a multilayer perceptron (MLP).

serialize_class_mlp serializes a multilayer perceptron (MLP) and its stored training samples (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_class_mlp and write_samples_class_mlp is converted to a serialized item. The multilayer perceptron is defined by the handle MLPHandle. The serialized multilayer perceptron is returned by the handle SerializedItemHandle and can be deserialized by deserialize_class_mlp.

Parameters

▷ **MLPHandle** (input_control) ..... class_mlp ⇝ *handle*
MLP handle.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_mlp

Possible Successors

clear_class_mlp, fwrite_serialized_item, send_serialized_item, deserialize_class_mlp

See also

create_class_mlp, read_class_mlp, write_samples_class_mlp, deserialize_class_mlp

Module

Foundation

set_regularization_params_class_mlp (: : MLPHandle, GenParamName, GenParamValue :)

Set the regularization parameters of a multilayer perceptron.

set_regularization_params_class_mlp sets the regularization parameters of the multilayer perceptron (MLP) passed in MLPHandle. The regularization parameter to be set is specified with GenParamName. Its value is specified with GenParamValue.

GenParamName can assume the following values:

'num_outer_iterations': This parameter determines whether the regularization parameters should be determined automatically (GenParamValue >= 1) or manually (GenParamValue = 0, default), as described below in the sections "Technical Background" and "Automatic Determination of the Regularization Parameters". As described in detail in the section "Automatic Determination of the Regularization Parameters", *'num_outer_iterations'* should not be set too large (in the range of 1 to 5) to enable manual checking of the convergence of the automatic determination of the regularization parameters.

'num_inner_iterations': This parameter potentially enables somewhat faster convergence of the automatic determination of the regularization parameters, as described below in the section "Automatic Determination of the Regularization Parameters". It should typically be left at its default value of 1.

'weight_prior': On the one hand, this selects the regularization model to be used, as described below in the section "Technical Background". On the other hand, if manual determination of the regularization parameters has been selected (i.e., *'num_outer_iterations'* = 0), the regularization parameters are set with GenParamName, whereas the initial values of the regularization parameters are set if automatic determination of the regularization parameters has been selected (i.e., *'num_outer_iterations'* >= 1), as described below in the section "Automatic Determination of the Regularization Parameters". Manual determination of the regularization parameters (see the section "Regularization Parameters" below) is only realistic if a single regularization parameter is used. In all other cases, the regularization parameters should be determined automatically.

'noise_prior': This allows to specify a noise prior for MLPs that have been configured for regression, as described below in the section "Application Areas". If manual determination of the regularization parameters has been selected, the noise prior is set with GenParamName, whereas the initial value of the noise prior is set if automatic determination of the regularization parameters has been selected. Typically, it is only useful to use this parameter if the regularization parameters are determined automatically.

Please note that the automatic determination of the regularization parameters requires a very large amount of memory and runtime, as described in detail in the section "Complexity" below. Therefore, NumHidden should not be selected too large when the MLP is created with create_class_mlp. For example, normal OCR applications seldom require NumHidden to be larger than 30-60.

Application Areas

As described at create_class_mlp, it may be desirable to regularize the MLP to enforce a smoother transition of the confidences between the different classes and to prevent overfitting of the MLP to the training data. To achieve this, a penalty for large MLP weights (which are the main reason for very sharp transitions between classes) can be added to the training of the MLP in train_class_mlp by setting GenParamName to *'weight_prior'* and setting GenParamValue to a value > 0.

If the MLP has been configured for regression (i.e., if OutputFunction was set to *'linear'* in create_class_mlp), an inverse variance of the expected noise in the data can be specified by setting GenParamName to *'noise_prior'* and setting GenParamValue to a value > 0. Setting the noise prior only has an effect if a weight prior has been specified. In this case, it can be used to weight the data error term (the output error of the MLP) against the weight error term.

As described in more detail below, the regularization parameters of the MLP may be determined automatically (at the expense of significantly increased training times) by setting GenParamName to *'num_outer_iterations'* and setting GenParamValue to a value > 0.

Technical Background

There are three different kinds of penalty terms that can be set with *'weight_prior'*. Note that in the following the parameters $w_{ji}^{(l)}$ and $b_k^{(l)}$ refer to the weights of the different layers of the MLP, as described in create_class_mlp.

If a single value α is specified, all MLP weights are penalized equally by adding the following term to the optimization in train_class_mlp:

$$E_W = \frac{\alpha}{2}\left(\sum_{i=1}^{n_i}\sum_{j=1}^{n_h}\left(w_{ji}^{(1)}\right)^2 + \sum_{j=1}^{n_h}\left(b_j^{(1)}\right)^2 + \sum_{j=1}^{n_h}\sum_{k=1}^{n_o}\left(w_{kj}^{(2)}\right)^2 + \sum_{k=1}^{n_o}\left(b_k^{(2)}\right)^2\right)$$

Alternatively, four values $[\alpha_{w1}, \alpha_{b1}, \alpha_{w2}, \alpha_{b2}]$ can be specified. These four parameters enable the individual regularization of the four groups of weights:

$$E_W = \frac{\alpha_{w1}}{2}\sum_{i=1}^{n_i}\sum_{j=1}^{n_h}\left(w_{ji}^{(1)}\right)^2 + \frac{\alpha_{b1}}{2}\sum_{j=1}^{n_h}\left(b_j^{(1)}\right)^2 + \frac{\alpha_{w2}}{2}\sum_{j=1}^{n_h}\sum_{k=1}^{n_o}\left(w_{kj}^{(2)}\right)^2 + \frac{\alpha_{b2}}{2}\sum_{k=1}^{n_o}\left(b_k^{(2)}\right)^2$$

Finally, $n_i + 3$ values $[\alpha_1, \ldots, \alpha_{n_i}, \alpha_{b1}, \alpha_{w2}, \alpha_{b2}]$ can be specified. These $n_i + 3$ parameters enable the individual regularization of each input variable $x_1, \ldots, x_{n_i}$ and the regularization of the remaining three groups of weights:

$$E_W = \sum_{i=1}^{n_i}\frac{\alpha_i}{2}\sum_{j=1}^{n_h}\left(w_{ji}^{(1)}\right)^2 + \frac{\alpha_{b1}}{2}\sum_{j=1}^{n_h}\left(b_j^{(1)}\right)^2 + \frac{\alpha_{w2}}{2}\sum_{j=1}^{n_h}\sum_{k=1}^{n_o}\left(w_{kj}^{(2)}\right)^2 + \frac{\alpha_{b2}}{2}\sum_{k=1}^{n_o}\left(b_k^{(2)}\right)^2$$

This kind of regularization is only useful in conjunction with the automatic determination of the regularization parameters described below. If the automatic determination of the regularization parameters returns a very large value of α_j (compared to the smallest value of the n_i values α_i), the corresponding input variable has little relevance for the MLP output. If this is the case, it should be tested whether the input variable can be omitted from the input of the MLP without negatively affecting the MLP's performance. The advantage of omitting irrelevant input variables is an increased speed of the MLP for classification.

The parameters α can be regarded as the inverse variance of a Gaussian prior distribution on the MLP weights, i.e., they express an expectation about the size of the MLP weights. The larger the α are chosen, the smaller the MLP weights will be.

Regularization Parameters

The larger the regularization parameter(s) *'weight_prior'* are chosen, the smoother the transition of the confidences between the different classes will be. The required values for the regularization parameter(s) depend on the MLP, especially the number of hidden units, the training data, and the scale of the training data (if no normalization is used). Typically, a higher value for the regularization parameter(s) is necessary if the MLP has more hidden units and if the training data consists of more points. For typical applications, the regularization parameters are determined by verifying the MLP performance on a test data set that is independent from the training data set. If an independent test data set is unavailable, cross validation can be used. Cross validation works by splitting the data set into separate parts (for example, 80% of the data set for training and 20% for testing), training the MLP with the training data set (the 80% of the data in the above example), and testing the MLP performance on the test set (the 20% of the data in the above example). The procedure can be repeated for the other possible splits of the data (in the 80%–20% example, there are five possible splits). This procedure can, for example, start with relatively large values of the weight regularization parameters (which will typically result in misclassifications on the test data set). The weight regularization parameters can then be decreased until an acceptable performance on the test data sets is reached.

Automatic Determination of the Regularization Parameters

The regularization parameters, i.e., the weight priors and the noise prior, can also be determined automatically by `train_class_mlp` using the so-called evidence procedure (for details about the evidence procedure, please refer to the articles in the section "References" below). This training mode can be selected by setting `GenParamName` to *'num_outer_iterations'* and setting `GenParamValue` to a value > 0. Note that this typically results in training times that are one to three orders of magnitude larger than simply training the MLP with fixed regularization parameters.

The evidence procedure is an iterative algorithm that performs the following two steps for a number of outer iterations: first, the network is trained using the current values of the regularization parameters; next, the regularization parameters are re-estimated using the weights of the optimized MLP. In the first iteration, the weight priors and noise priors specified with *'weight_prior'* and *'noise_prior'* are used. Thus, for the automatic determination of the regularization parameters, the values specified by the user serve as the starting parameters for the evidence procedure. The starting parameters for the weight priors should not be set too large because this might over-regularize the training and may result in badly determined regularization parameters. The initial values for the weight priors should typically be in the range 0.01-0.1.

The number of outer iterations can be set by setting `GenParamName` to *'num_outer_iterations'* and setting `GenParamValue` to a value > 0. If `GenParamValue` is set to 0 (this is the default value), the evidence procedure is not executed and the MLP is simply trained using the user-specified regularization parameters.

The number of outer iterations should be set high enough to ensure the convergence of the regularization parameters. In contrast to the training of the MLP's weights, a numerical convergence criterion is typically very difficult to specify and some human judgment is typically required to decide whether the regularization parameters have converged sufficiently. Therefore, it might not be possible to set the number of outer iterations a-priori to ensure convergence of the regularization parameters. In these cases, the outer loop over the steps of the evidence procedure can be implemented manually by setting *'num_outer_iterations'* to 1 and calling `train_class_mlp` repeatedly. This has the advantage that the weight priors and noise prior can be queried after each iteration and can be checked manually for convergence. In this approach, the performance of the MLP can even be checked after each iteration on an independent test set to check the generalization performance of the classifier.

If the number of outer iterations has been determined (approximately) for a class of applications, it may be possible to reduce the run time of the training (if MLPs should be trained in the future with similar data sets) by setting `GenParamName` to *'num_inner_iterations'* and setting `GenParamValue` to a value > 1 (the default value is 1) and by reducing the number of outer iterations. The number of outer iterations can typically not be reduced by the same factor by which the number of inner iterations is increased. Using this approach, the run time of the training

can be optimized. However, this approach is only useful if many MLPs are trained with similar data sets. If this is not the case, *'num_inner_iterations'* should be left at its default value of 1.

The automatically determined weight priors and noise prior can be queried after the training using `get_regularization_params_class_mlp` by setting `GenParamName` to *'weight_prior'* or *'noise_prior'*, respectively.

In addition to the weight prior and noise prior, the evidence procedure determines an estimate of the number of parameters of the MLP that can be determined well using the training data. This result can be queried using `get_regularization_params_class_mlp` by setting `GenParamName` to *'num_well_determined_params'*. Alternatively, the fraction of well-determined parameters can be queried by setting `GenParamName` to *'fraction_well_determined_params'*. If the number of well-determined parameters is significantly smaller than n_w (where n_w is the number of weights in the MLP, as described in the section "Complexity" below) or the fraction of well-determined parameters is significantly smaller than 1, consider reducing the number of hidden units or, if the number of hidden units cannot be decreased without increasing the error rate of the MLP significantly, consider performing a preprocessing that reduces the number of input variables to the net, i.e., canonical variates or principal components.

Please note that the number of well-determined parameters can only be determined after the weight priors and noise prior have been determined. This is the reason why the evidence procedure ends with the determination of the regularization parameters and not with the training of the MLP weights. Hence, after the evidence procedure the MLP will not have been trained with the latest regularization parameters. This should make no difference if they have converged. If you want the training to end with an optimization of the weights using the latest values of the regularization parameters, you can set *'num_outer_iterations'* to 0 and can call `train_class_mlp` again. If you do so, please note, however, that the number of well-determined parameters may change and, therefore, the value returned by `get_regularization_params_class_mlp` is technically inconsistent.

Saved Parameters

Note that the parameters *'num_outer_iterations'* and *'num_inner_iterations'* only affect the training of the MLP. Therefore, they are not saved when the MLP is stored using `write_class_mlp` or `serialize_class_mlp`. Thus, they must be set anew if the MLP is loaded again using `read_class_mlp` or `deserialize_class_mlp` and if training using the automatic determination of the regularization parameters should be continued. All other parameters described above (*'weight_prior'*, *'noise_prior'*, *'num_well_determined_params'*, and *'fraction_well_determined_params'*) are saved.

Parameters

▷ **`MLPHandle`** (input_control) ..... class_mlp ⇝ *handle*
MLP handle.

▷ **`GenParamName`** (input_control) ..... string ⇝ *string*
Name of the regularization parameter to set.
Default: 'weight_prior'
List of values: `GenParamName` ∈ {'weight_prior', 'noise_prior', 'num_outer_iterations', 'num_inner_iterations'}

▷ **`GenParamValue`** (input_control) ..... number(-array) ⇝ *real* / integer
Value of the regularization parameter.
Default: 1.0
Suggested values: `GenParamValue` ∈ {0.01, 0.1, 1.0, 10.0, 100.0, 0, 1, 2, 3, 5, 10, 15, 20}

Example

```
* This example shows how to determine the regularization parameters
* automatically without examining the convergence of the
* regularization parameters.
* Create the MLP.
create_class_mlp (NumIn, NumHidden, NumOut, 'softmax', \
                  'normalization', NumIn, 42, MLPHandle)
* Generate and add the training data
for J := 0 to NumData-1 by 1
    * Generate training features and classes.
    * Data = [...]
    * Class = [...]
    add_sample_class_mlp (MLPHandle, Data, Class)
```

```
endfor
* Set up the automatic determination of the regularization
* parameters.
set_regularization_params_class_mlp (MLPHandle, 'weight_prior', \
                                     [0.01,0.01,0.01,0.01])
set_regularization_params_class_mlp (MLPHandle, \
                                     'num_outer_iterations', 10)
* Train the MLP.
train_class_mlp (MLPHandle, 100, 1, 0.01, Error, ErrorLog)
* Read out the estimate of the number of well-determined
* parameters.
get_regularization_params_class_mlp (MLPHandle, \
                                     'fraction_well_determined_params', \
                                     FractionParams)
* If FractionParams differs substantially from 1, consider reducing
* NumHidden appropriately and consider performing a preprocessing that
* reduces the number of input variables to the net, i.e., canonical
* variates or principal components.
write_class_mlp (MLPHandle, 'classifier.mlp')



* This example shows how to determine the regularization parameters
* automatically while examining the convergence of the
* regularization parameters.
* Create the MLP.
create_class_mlp (NumIn, NumHidden, NumOut, 'softmax', \
                  'normalization', NumIn, 42, MLPHandle)
* Generate and add the training data.
for J := 0 to NumData-1 by 1
    * Generate training features and classes
    * Data = [...]
    * Class = [...]
    add_sample_class_mlp (MLPHandle, Data, Class)
endfor
* Set up the automatic determination of the regularization
* parameters.
set_regularization_params_class_mlp (MLPHandle, 'weight_prior', \
                                     [0.01,0.01,0.01,0.01])
set_regularization_params_class_mlp (MLPHandle, \
                                     'num_outer_iterations', 1)
for OuterIt := 1 to 10 by 1
    * Train the MLP
    train_class_mlp (MLPHandle, 100, 1, 0.01, Error, ErrorLog)
    * Read out the regularization parameters
    get_regularization_params_class_mlp (MLPHandle, 'weight_prior', \
                                         WeightPrior)
    * Inspect the regularization parameters manually for
    * convergence and exit the loop manually if they have
    * converged.
    * [...]
endfor
* Read out the estimate of the number of well-determined
* parameters.
get_regularization_params_class_mlp (MLPHandle,\
                                     'fraction_well_determined_params',\
                                     FractionParams)
* If FractionParams differs substantially from 1, consider reducing
* NumHidden appropriately and consider performing a preprocessing that
```

```
* reduces the number of input variables to the net, i.e., canonical
* variates or principal components.
write_class_mlp (MLPHandle, 'classifier.mlp')
```

Complexity

Let n_i denote the number of input units of the MLP (i.e., n_i = `NumHidden` or n_i = `NumComponents`, depending on the value of `Preprocessing`, as described at `create_class_mlp`), n_h the number of hidden units, and n_o the number of output units. Then, the number of weights of the MLP is $n_w = (n_i+1)n_h+(n_h+1)n_o$. Let n_d denote the number of training samples. Let n_M denote the number of iterations set with `MaxIterations` in `train_class_mlp`. Let n_O and n_I denote the number of outer and inner iterations, respectively.

The run time of the training without regularization or with regularization with fixed regularization parameters is of complexity $O(n_M n_w n_d)$. In contrast, the runtime of the training with automatic determination of the regularization parameters is of complexity

$O(n_O n_M n_w n_d) + O(n_O n_w^2 n_d) + O(n_O n_w^3) + O(n_O n_I n_w^3)$.

The training without regularization or with regularization with fixed regularization parameters requires at least $48n_w + 24n_h n_d + 16n_o n_d$ bytes of memory. The training with automatic determination of the regularization parameters requires at least $24n_w^2 + 48n_w + 72n_h n_d + 56n_o n_d$ bytes of memory. Under special circumstances, another $24n_w^2 + 8n_w$ bytes of memory are required.

Result

If the parameters are valid, the operator `set_regularization_params_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MLPHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_class_mlp`

Possible Successors

`get_regularization_params_class_mlp`, `train_class_mlp`

References

David J. C. MacKay: “Bayesian Interpolation”; Neural Computation 4(3):415-447; 1992.

David J. C. MacKay: “A Practical Bayesian Framework for Backpropagation Networks”; Neural Computation 4(3):448-472; 1992.

David J. C. MacKay: “The Evidence Framework Applied to Classification Networks”; Neural Computation 4(5): 720-736; 1992.

David J. C. MacKay: “Comparison of Approximate Methods for Handling Hyperparameters”; Neural Computation 11(5):1035-1068; 1999.

Module

Foundation

set_rejection_params_class_mlp (: : MLPHandle, GenParamName, GenParamValue :)

Set the parameters of a rejection class.

set_rejection_params_class_mlp sets the parameters of an automatically generated rejection class inside of a multilayer perceptron (MLP) given by MLPHandle. In some applications, it is desirable to know whether a feature vector is similar to one of the training set. If a feature vector lies outside of the provided training set, it should be classified as a special rejection class. This means that the feature vector is different to the confidence area of the classifier. If a rejection class should be created automatically, an additional class must be specified while creating the classifier in create_class_mlp. Here, the parameter NumOutput must be increased by one.

The parameters of the rejection class are selected with GenParamName and the respective values with GenParamValue.

'rejection_class_index': By default, the last class serves as the rejection class. If another class should be used, GenParamName must be set to *'rejection_class_index'* and GenParamValue to the class index.

'sampling_strategy': Currently, three strategies exist to generate samples for the rejection class during the training of the MLP. These strategies can be selected by setting GenParamName to *'sampling_strategy'* and GenParamValue to *'hyperbox_around_all_classes'*, *'hyperbox_around_each_class'*, or *'hyperbox_ring_around_each_class'*. The sampling strategy *'hyperbox_around_all_classes'* takes the bounding box of all training samples that have been provided so far. The sampling strategy *'hyperbox_around_each_class'* is similar with the only difference that the bounding box around each class is taken as the area where the rejection samples are generated. The sampling strategy *'hyperbox_ring_around_each_class'* generates samples only in the enlarged areas around the bounding box of each class, thus generating a hyperbox ring around the original samples. Please note that with increasing dimensionality the sampling strategies *'hyperbox_around_each_class'* and *'hyperbox_ring_around_each_class'* provide the same result. If no rejection class sampling strategy should be used, which is the default, GenParamValue must be set to *'no_rejection_class'*.

'hyperbox_tolerance': The factor *'hyperbox_tolerance'* describes by what amount the bounding box should be enlarged in all dimensions. Then, inside this box samples are randomly generated from a uniform distribution. The default value is 0.2.

'rejection_sample_factor': The number of rejection samples is the number of provided samples multiplied by *'rejection_sample_factor'*. If not enough samples are generated, the rejection class may not be classified correctly. If the rejection class has too many samples, the normal classes are classified as rejection class. The default value is 1.0. Note that the training time will increase by a factor of $1 + f$, where f is the value of *'rejection_sample_factor'*.

'random_seed': To ensure reproducible results, a random seed can be set with *'random_seed'*. The default value is 42.

Because this operator only parametrizes the training of the MLP, the values are not saved by write_class_mlp.

Parameters

▷ **MLPHandle** (input_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Names of the generic parameters.
Default: 'sampling_strategy'
List of values: GenParamName ∈ {'sampling_strategy', 'hyperbox_tolerance', 'rejection_sample_factor', 'random_seed', 'rejection_class_index'}

▷ **GenParamValue** (input_control) . . . . . string(-array) ⇝ *string* / real / integer
Values of the generic parameters.
Default: 'hyperbox_around_all_classes'
List of values: GenParamValue ∈ {'no_rejection_class', 'hyperbox_around_all_classes', 'hyperbox_around_each_class', 'hyperbox_ring_around_each_class'}

Result

If the parameters are valid, the operator set_rejection_params_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MLPHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_mlp

Possible Successors

train_class_mlp

Module

Foundation

train_class_mlp (: : MLPHandle, MaxIterations, WeightTolerance, ErrorTolerance : Error, ErrorLog)

Train a multilayer perceptron.

train_class_mlp trains the multilayer perceptron (MLP) given in MLPHandle. Before the MLP can be trained, *all* training samples to be used for the training must be stored in the MLP using add_sample_class_mlp or read_samples_class_mlp. If after the training new additional training samples should be used a new MLP must be created with create_class_mlp, in which again *all* training samples to be used (i.e., the original ones and the additional ones) must be stored. In these cases, it is useful to save and read the training data with write_samples_class_mlp and read_samples_class_mlp, respectively. A second training with additional training samples is not explicitly forbidden by train_class_mlp. However, this typically does not lead to good results because the training of an MLP is a complex nonlinear optimization problem, and consequently the second training with new data will very likely lead to the fact that the optimization gets stuck in a local minimum.

If a rejection class has been specified using set_rejection_params_class_mlp, before the actual training the samples for the rejection class are generated.

During the training, the error the MLP achieves on the stored training samples is minimized by using a nonlinear optimization algorithm. If the MLP has been regularized with set_regularization_params_class_mlp, an additional weight penalty term is taken into account. With this, the MLP weights described in create_class_mlp are determined. Furthermore, if an automatic determination of the regularization parameters has been specified with set_regularization_params_class_mlp, these parameters are optimized as well. As described at set_regularization_params_class_mlp, training the MLP with automatic determination of the regularization parameters requires significantly more time than training an unregularized MLP or an MLP with fixed regularization parameters.

create_class_mlp initializes the MLP weights with random values to make it very likely that the optimization converges to the global minimum of the error function. Nevertheless, in rare cases it may happen that the random values determined with RandSeed in create_class_mlp result in a relatively large optimum error, i.e., that the optimization gets stuck in a local minimum. If it can be conjectured that this has happened the MLP should be created anew with a different value for RandSeed in order to check whether a significantly smaller error can be achieved.

The parameters MaxIterations, WeightTolerance, and ErrorTolerance control the nonlinear optimization algorithm. Note that if an automatic determination of the regularization parameters has been specified with set_regularization_params_class_mlp, these parameters refer to one training within one step of the evidence procedure. MaxIterations specifies the maximum number of iterations of the optimization algorithm. In practice, values between *100* and *200* should be sufficient for most problems. WeightTolerance specifies a threshold for the change of the weights per iteration. Here, the absolute value of the change of the weights between two iterations is summed. Hence, this value depends on the number of weights as well as the size of the weights, which in turn depend on the scaling of the training data. Typically, values between *0.00001* and *1* should be used. ErrorTolerance specifies a threshold for the change of the error value per iteration. This

value depends on the number of training samples as well as the number of output variables of the MLP. Also here, values between *0.00001* and *1* should typically be used. The optimization is terminated if the weight change is smaller than `WeightTolerance` and the change of the error value is smaller than `ErrorTolerance`. In any case, the optimization is terminated after at most `MaxIterations` iterations. It should be noted that, depending on the size of the MLP and the number of training samples, the training can take from a few seconds to several hours.

On output, `train_class_mlp` returns the error of the MLP with the optimal weights on the training samples in `Error`. Furthermore, `ErrorLog` contains the error value as a function of the number of iterations. With this, it is possible to decide whether a second training of the MLP with the same training data without creating the MLP anew makes sense. If `ErrorLog` is regarded as a function, it should drop off steeply initially, while leveling out very flatly at the end. If `ErrorLog` is still relatively steep at the end, it usually makes sense to call `train_class_mlp` again. It should be noted, however, that this mechanism should *not* be used to train the MLP successively with `MaxIterations` = *1* (or other small values for `MaxIterations`) because this will substantially increase the number of iterations required to train the MLP. Note that if an automatic determination of the regularization parameters has been specified with `set_regularization_params_class_mlp`, `Error` and `ErrorLog` refer to the last training that was executed in the evidence procedure. If the error log should be monitored within the individual iterations of the evidence procedure, the outer iteration of the evidence procedure must be implemented explicitly, as described at `set_regularization_params_class_mlp`.

Parameters

- ▷ **`MLPHandle`** (input_control) ..... class_mlp ⇝ *handle*
  MLP handle.
- ▷ **`MaxIterations`** (input_control) ..... integer ⇝ *integer*
  Maximum number of iterations of the optimization algorithm.
  Default: 200
  Suggested values: `MaxIterations` ∈ {20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280, 300}
- ▷ **`WeightTolerance`** (input_control) ..... real ⇝ *real*
  Threshold for the difference of the weights of the MLP between two iterations of the optimization algorithm.
  Default: 1.0
  Suggested values: `WeightTolerance` ∈ {1.0, 0.1, 0.01, 0.001, 0.0001, 0.00001}
  Restriction: WeightTolerance >= 1.0e-8
- ▷ **`ErrorTolerance`** (input_control) ..... real ⇝ *real*
  Threshold for the difference of the mean error of the MLP on the training data between two iterations of the optimization algorithm.
  Default: 0.01
  Suggested values: `ErrorTolerance` ∈ {1.0, 0.1, 0.01, 0.001, 0.0001, 0.00001}
  Restriction: ErrorTolerance >= 1.0e-8
- ▷ **`Error`** (output_control) ..... real ⇝ *real*
  Mean error of the MLP on the training data.
- ▷ **`ErrorLog`** (output_control) ..... real-array ⇝ *real*
  Mean error of the MLP on the training data as a function of the number of iterations of the optimization algorithm.

Example

```
* Train an MLP
create_class_mlp (NumIn, NumHidden, NumOut, 'softmax', \
                  'normalization', 1, 42, MLPHandle)
read_samples_class_mlp (MLPHandle, 'samples.mtf')
train_class_mlp (MLPHandle, 100, 1, 0.01, Error, ErrorLog)
write_class_mlp (MLPHandle, 'classifier.mlp')
```

Result

If the parameters are valid, the operator `train_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

`train_class_mlp` may return the error 9211 (Matrix is not positive definite) if `Preprocessing` = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- MLPHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`add_sample_class_mlp`, `read_samples_class_mlp`, `set_regularization_params_class_mlp`

Possible Successors

`evaluate_class_mlp`, `classify_class_mlp`, `write_class_mlp`, `create_class_lut_mlp`

Alternatives

`train_dl_classifier_batch`, `read_class_mlp`

See also

`create_class_mlp`

References

Christopher M. Bishop: "Neural Networks for Pattern Recognition"; Oxford University Press, Oxford; 1995.

Andrew Webb: "Statistical Pattern Recognition"; Arnold, London; 1999.

Module

Foundation

`write_class_mlp` (: : MLPHandle, FileName :)

Write a multilayer perceptron to a file.

`write_class_mlp` writes the multilayer perceptron (MLP) `MLPHandle` to the file given by `FileName`. The default HALCON file extension for the MLP classifier is 'gmc'. `write_class_mlp` is typically called after the MLP has been trained with `train_class_mlp`. The MLP can be read with `read_class_mlp`. `write_class_mlp` does *not* write any training samples that possibly have been stored in the MLP. For this purpose, `write_samples_class_mlp` should be used.

Parameters

▷ **`MLPHandle`** (input_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

▷ **`FileName`** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: `.gmc`

Result

If the parameters are valid, the operator `write_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_mlp

Possible Successors

clear_class_mlp

See also

create_class_mlp, read_class_mlp, write_samples_class_mlp

Module

Foundation

write_samples_class_mlp (: : MLPHandle, FileName :)

Write the training data of a multilayer perceptron to a file.

write_samples_class_mlp writes the training samples stored in the multilayer perceptron (MLP) MLPHandle to the file given by FileName. write_samples_class_mlp can be used to build up a database of training samples, and hence to improve the performance of the MLP by training it with an extended data set (see train_class_mlp). For other possible uses of write_samples_class_mlp see get_prep_info_class_mlp.

The file FileName is overwritten by write_samples_class_mlp. Nevertheless, extending the database of training samples is easy to do because read_samples_class_mlp and add_sample_class_mlp add the training samples to the training samples that are already stored in memory with the MLP.

Parameters

▷ **MLPHandle** (input_control) . . . . . class_mlp ⇝ *handle*
MLP handle.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.

Result

If the parameters are valid, the operator write_samples_class_mlp returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_mlp

Possible Successors

clear_samples_class_mlp

See also

create_class_mlp, get_prep_info_class_mlp, read_samples_class_mlp

Module

Foundation

7.6 Support Vector Machines

add_class_train_data_svm (: : SVMHandle,
ClassTrainDataHandle :)

Add training data to a support vector machine (SVM).

add_class_train_data_svm adds the training data specified by ClassTrainDataHandle to a support vector machine (SVM) specified by SVMHandle.

Parameters

▷ **SVMHandle** (input_control) . . . . . class_svm ⇝ *handle*
Handle of a SVM which receives the training data.

▷ **ClassTrainDataHandle** (input_control) . . . . . class_train_data ⇝ *handle*
Training data for a classifier.

Result

If the parameters are valid, the operator add_class_train_data_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SVMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_svm, create_class_train_data

Possible Successors

get_sample_class_svm

Alternatives

add_sample_class_svm

See also

create_class_svm

Module

Foundation

add_sample_class_svm (: : SVMHandle, Features, Class :)

Add a training sample to the training data of a support vector machine.

add_sample_class_svm adds a training sample to the support vector machine (SVM) given by SVMHandle. The training sample is given by Features and Class. Features is the feature vector of the sample, and consequently must be a real vector of length NumFeatures, as specified in create_class_svm. Class is the target of the sample, which must be in the range of 0 to NumClasses-1 (see create_class_svm). In the special case of *'novelty-detection'* the class is to be set to *0* as only one class is assumed. Before the SVM can be trained with train_class_svm, training samples must be added to the SVM with add_sample_class_svm. The usage of support vectors of an already trained SVM as training samples is described in train_class_svm.

The number of currently stored training samples can be queried with get_sample_num_class_svm. Stored training samples can be read out again with get_sample_class_svm.

Normally, it is useful to save the training samples in a file with write_samples_class_svm to facilitate reusing the samples and to facilitate that, if necessary, new training samples can be added to the data set, and hence to facilitate that a *newly created* SVM can be trained with the extended data set.

Parameters

▷ **SVMHandle** (input_control) . . . . . class_svm ⇝ *handle*
SVM handle.

▷ **Features** (input_control) . . . . . real-array ⇝ *real*
Feature vector of the training sample to be stored.

▷ **Class** (input_control) . . . . . number ⇝ *integer* / real
Class of the training sample to be stored.

Result

If the parameters are valid the operator `add_sample_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SVMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_class_svm`

Possible Successors

`train_class_svm`, `write_samples_class_svm`, `get_sample_num_class_svm`, `get_sample_class_svm`

Alternatives

`read_samples_class_svm`

See also

`clear_samples_class_svm`, `get_support_vector_class_svm`

Module

Foundation

classify_class_svm (: : SVMHandle, Features, Num : Class)

Classify a feature vector by a support vector machine.

`classify_class_svm` computes the best `Num` classes of the feature vector `Features` with the SVM `SVMHandle` and returns them in `Class`. If the classifier was created in the `Mode` = *'one-versus-one'*, the classes are ordered by the number of votes of the sub-classifiers. If `Mode` = *'one-versus-all'* was used, the classes are ordered by the value of each sub-classifier (see `create_class_svm` for more details). If the classifier was created in the `Mode` = *'novelty-detection'*, it determines whether the feature vector belongs to the same class as the training data (`Class` = 1) or is regarded as outlier (`Class` = 0). In this case `Num` must be set to 1 as the classifier only determines membership.

Before calling `classify_class_svm`, the SVM must be trained with `train_class_svm`.

Parameters

▷ **SVMHandle** (input_control) . . . . . class_svm ⇝ *handle*
SVM handle.

▷ **Features** (input_control) . . . . . real-array ⇝ *real*
Feature vector.

▷ **Num** (input_control) . . . . . integer-array ⇝ *integer*
Number of best classes to determine.
Default: 1
Suggested values: Num ∈ {1, 2, 3, 4, 5}

▷ **Class** (output_control) . . . . . integer(-array) ⇝ *integer*
Result of classifying the feature vector with the SVM.

Result

If the parameters are valid the operator classify_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_svm, read_class_svm

Alternatives

apply_dl_classifier

See also

create_class_svm

References

John Shawe-Taylor, Nello Cristianini: "Kernel Methods for Pattern Analysis"; Cambridge University Press, Cambridge; 2004.

Bernhard Schölkopf, Alexander J.Smola: "Learning with Kernels"; MIT Press, London; 1999.

Module

Foundation

clear_class_svm (: : SVMHandle :)

Clear a support vector machine.

clear_class_svm clears the support vector machine (SVM) given by SVMHandle and frees all memory required for the SVM. After calling clear_class_svm, the SVM can no longer be used. The handle SVMHandle becomes invalid.

Parameters

▷ **SVMHandle** (input_control) . . . . . class_svm(-array) ⇝ *handle*
SVM handle.

Result

If SVMHandle is valid the operator clear_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SVMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

classify_class_svm

See also

create_class_svm, read_class_svm, write_class_svm, train_class_svm

Module

Foundation

clear_samples_class_svm (: : SVMHandle :)

Clear the training data of a support vector machine.

clear_samples_class_svm clears all training samples that have been added to the support vector machine (SVM) SVMHandle with add_sample_class_svm or read_samples_class_svm. clear_samples_class_svm should only be used if the SVM is trained in the same process that uses the SVM for classification with classify_class_svm. In this case, the memory required for the training samples can be freed with clear_samples_class_svm, and hence memory can be saved. In the normal usage, in which the SVM is trained offline and written to a file with write_class_svm, it is typically unnecessary to call clear_samples_class_svm because write_class_svm does not save the training samples, and hence the online process, which reads the SVM with read_class_svm, requires no memory for the training samples.

Parameters

▷ **SVMHandle** (input_control) ..... class_svm(-array) ⇝ *handle*
SVM handle.

Result

If the parameters are valid the operator clear_samples_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SVMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

train_class_svm, write_samples_class_svm

See also

create_class_svm, clear_class_svm, add_sample_class_svm, read_samples_class_svm

Module

Foundation

create_class_svm (: : NumFeatures, KernelType, KernelParam, Nu, NumClasses, Mode, Preprocessing, NumComponents : SVMHandle)

Create a support vector machine for pattern classification.

create_class_svm creates a support vector machine that can be used for pattern classification. The dimension of the patterns to be classified is specified in NumFeatures, the number of different classes in NumClasses.

For a binary classification problem in which the classes are linearly separable the SVM algorithm selects data vectors from the training set that are utilized to construct the optimal separating hyperplane between different classes. This hyperplane is optimal in the sense that the margin between the convex hulls of the different classes is maximized. The training patterns that are located at the margin define the hyperplane and are called *support vectors* (SV).

Classification of a feature vector z is performed with the following formula:

$$f(z) \quad = \quad sign\left(\sum_{i=1}^{n_{sv}} \alpha_i y_i < x_i, z > +b\right)$$

Here, x_i are the support vectors, y_i encodes their class membership (± 1) and α_i the weight coefficients. The distance of the hyperplane to the origin is b. The α and b are determined during training with `train_class_svm`. Note that only a subset of the original training set (n_{sv}: number of support vectors) is necessary for the definition of the decision boundary and therefore data vectors that are not support vectors are discarded. The classification speed depends on the evaluation of the dot product between support vectors and the feature vector to be classified, and hence depends on the length of the feature vector and the number n_{sv} of support vectors.

For classification problems in which the classes are not linearly separable the algorithm is extended in two ways. First, during training a certain amount of errors (overlaps) is compensated with the use of slack variables. This means that the α are upper bounded by a regularization constant. To enable an intuitive control of the amount of training errors, the *Nu*-SVM version of the training algorithm is used. Here, the regularization parameter `Nu` is an asymptotic upper bound on the number of training errors and an asymptotic lower bound on the number of support vectors. As a rule of thumb, the parameter `Nu` should be set to the prior expectation of the application's specific error ratio, e.g., *0.01* (corresponding to a maximum training error of 1%). Please note that a too big value for `Nu` might lead to an infeasible training problem, i.e., the SVM cannot be trained correctly (see `train_class_svm` for more details). Since this can only be determined during training, an exception can only be raised there. In this case, a new SVM with `Nu` chosen smaller must be created.

Second, because the above SVM exclusively calculates dot products between the feature vectors, it is possible to incorporate a kernel function into the training and testing algorithm. This means that the dot products are substituted by a kernel function, which implicitly performs the dot product in a higher dimensional feature space. Given the appropriate kernel transformation, an originally not linearly separable classification task becomes linearly separable in the higher dimensional feature space.

Different kernel functions can be selected with the parameter `KernelType`. For `KernelType` = *'linear'* the dot product, as specified in the above formula is calculated. This kernel should solely be used for linearly or nearly linearly separable classification tasks. The parameter `KernelParam` is ignored here.

The radial basis function (RBF) `KernelType` = *'rbf'* is the best choice for a kernel function because it achieves good results for many classification tasks. It is defined as:

$$K(x,z) \quad = \quad e^{-\gamma \cdot \left\|x-z\right\|^2}$$

Here, the parameter `KernelParam` is used to select γ. The intuitive meaning of γ is the amount of influence of a support vector upon its surroundings. A big value of γ (small influence on the surroundings) means that each training vector becomes a support vector. The training algorithm learns the training data "by heart", but lacks any generalization ability (over-fitting). Additionally, the training/classification times grow significantly. A too small value for γ (big influence on the surroundings) leads to few support vectors defining the separating hyperplane (under-fitting). One typical strategy is to select a small γ-`Nu` pair and consecutively increase the values as long as the recognition rate increases.

With `KernelType` = *'polynomial_homogeneous'* or *'polynomial_inhomogeneous'*, polynomial kernels can be selected. They are defined in the following way:

$$\begin{aligned} K(x,z) &= (< x,z >)^d \\ K(x,z) &= (< x,z > +1)^d \end{aligned}$$

The degree of the polynomial kernel must be set with `KernelParam`. Please note that a too high degree polynomial ($d > 10$) might result in numerical problems.

As a rule of thumb, the RBF kernel provides a good choice for most of the classification problems and should therefore be used in almost all cases. Nevertheless, the linear and polynomial kernels might be better suited for certain applications and can be tested for comparison. Please note that the novelty-detection Mode and the operator reduce_class_svm are provided only for the RBF kernel.

Mode specifies the general classification task, which is either how to break down a multi-class decision problem to binary sub-cases or whether to use a special classifier mode called *'novelty-detection'*. Mode = *'one-versus-all'* creates a classifier where each class is compared to the rest of the training data. During testing the class with the largest output (see the classification formula without sign) is chosen. Mode = *'one-versus-one'* creates a binary classifier between each single class. During testing a vote is cast and the class with the majority of the votes is selected. The optimal Mode for multi-class classification depends on the number of classes. Given n classes *'one-versus-all'* creates n classifiers, whereas *'one-versus-one'* creates $n(n-1)/2$. Note that for a binary decision task *'one-versus-one'* would create exactly one, whereas *'one-versus-all'* unnecessarily creates two symmetric classifiers. For few classes (approximately up to 10) *'one-versus-one'* is faster for training and testing, because the sub-classifier all consist of fewer training data and result in overall fewer support vectors. In case of many classes *'one-versus-all'* is preferable, because *'one-versus-one'* generates a prohibitively large amount of sub-classifiers, as their number increases to the square of the number of classes.

A special case of classification is Mode = *'novelty-detection'*, where the test data is classified only with regard to membership to the training data, i.e., NumClasses must be set to 1. The separating hyperplane lies around the training data and thereby implicitly divides the training data from the rejection class. The advantage is that the rejection class is not defined explicitly, which is difficult to do in certain applications like texture classification. The resulting support vectors are all lying at the border. With the parameter Nu, the ratio of outliers in the training data set is specified. Note, that when classifying in the *'novelty-detection'* mode, the class of the training data is returned with index 1 and the rejection class is returned with index 0. Thus, the first class serves as rejection class. In contrast, when using the MLP classifier, the last class serves as rejection class by default.

The parameters Preprocessing and NumComponents can be used to specify a preprocessing of the feature vectors. For Preprocessing = *'none'*, the feature vectors are passed unaltered to the SVM. NumComponents is ignored in this case.

For all other values of Preprocessing, the training data set is used to compute a transformation of the feature vectors during the training as well as later in the classification.

For Preprocessing = *'normalization'*, the feature vectors are normalized. In case of a polynomial kernel, the minimum and maximum value of the training data set is transformed to -1 and +1. In case of the RBF kernel, the data is normalized by subtracting the mean of the training vectors and dividing the result by the standard deviation of the individual components of the training vectors. Hence, the transformed feature vectors have a mean of 0 and a standard deviation of 1. The normalization does not change the length of the feature vector. NumComponents is ignored in this case. This transformation can be used if the mean and standard deviation of the feature vectors differs substantially from 0 and 1, respectively, or for data in which the components of the feature vectors are measured in different units (e.g., if some of the data are gray value features and some are region features, or if region features are mixed, e.g., 'circularity' (unit: scalar) and 'area' (unit: pixel squared)). The normalization transformation should be performed in general, because it increases the numerical stability during training/testing.

For Preprocessing = *'principal_components'*, a principal component analysis (PCA) is performed. First, the feature vectors are normalized (see above). Then, an orthogonal transformation (a rotation in the feature space) that decorrelates the training vectors is computed. After the transformation, the mean of the training vectors is 0 and the covariance matrix of the training vectors is a diagonal matrix. The transformation is chosen such that the transformed features that contain the most variation is contained in the first components of the transformed feature vector. With this, it is possible to omit the transformed features in the last components of the feature vector, which typically are mainly influenced by noise, without losing a large amount of information. The parameter NumComponents can be used to determine how many of the transformed feature vector components should be used. Up to NumFeatures components can be selected. The operator get_prep_info_class_svm can be used to determine how much information each transformed component contains. Hence, it aids the selection of NumComponents. Like data normalization, this transformation can be used if the mean and standard deviation of the feature vectors differs substantially from 0 and 1, respectively, or for feature vectors in which the components of the data are measured in different units. In addition, this transformation is useful if it can be expected that the features are highly correlated. Please note that the RBF kernel is very robust against the dimensionality reduction performed by PCA and should therefore be the first choice when speeding up the classification time.

The transformation specified by Preprocessing = *'canonical_variates'* first normalizes the training vectors and then decorrelates the training vectors on average over all classes. At the same time, the transformation maxi-

mally separates the mean values of the individual classes. As for Preprocessing = *'principal_components'*, the transformed components are sorted by information content, and hence transformed components with little information content can be omitted. For canonical variates, up to min(NumClasses−1, NumFeatures) components can be selected. Also in this case, the information content of the transformed components can be determined with get_prep_info_class_svm. Like principal component analysis, canonical variates can be used to reduce the amount of data without losing a large amount of information, while additionally optimizing the separability of the classes after the data reduction. The computation of the canonical variates is also called linear discriminant analysis.

For the last two types of transformations (*'principal_components'* and *'canonical_variates'*), the length of input data of the SVM is determined by NumComponents, whereas NumFeatures determines the dimensionality of the input data (i.e., the length of the untransformed feature vector). Hence, by using one of these two transformations, the size of the SVM with respect to data length is reduced, leading to shorter training/classification times by the SVM.

After the SVM has been created with create_class_svm, typically training samples are added to the SVM by repeatedly calling add_sample_class_svm or read_samples_class_svm. After this, the SVM is typically trained using train_class_svm. Hereafter, the SVM can be saved using write_class_svm. Alternatively, the SVM can be used immediately after training to classify data using classify_class_svm.

A comparison of the SVM and the multi-layer perceptron (MLP) (see create_class_mlp) typically shows that SVMs are generally faster at training, especially for huge training sets, and achieve slightly better recognition rates than MLPs. The MLP is faster at classification and should therefore be preferred in time critical applications. Please note that this guideline assumes optimal tuning of the parameters.

Parameters

▷ **NumFeatures** (input_control) . . . integer ⇝ *integer*
Number of input variables (features) of the SVM.
Default: 10
Suggested values: NumFeatures ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Restriction: NumFeatures >= 1

▷ **KernelType** (input_control) . . . string ⇝ *string*
The kernel type.
Default: 'rbf'
List of values: KernelType ∈ {'linear', 'rbf', 'polynomial_inhomogeneous', 'polynomial_homogeneous'}

▷ **KernelParam** (input_control) . . . real ⇝ *real*
Additional parameter for the kernel function. In case of RBF kernel the value for γ. For polynomial kernel the degree
Default: 0.02
Suggested values: KernelParam ∈ {0.01, 0.02, 0.05, 0.1, 0.5}

▷ **Nu** (input_control) . . . real ⇝ *real*
Regularization constant of the SVM.
Default: 0.05
Suggested values: Nu ∈ {0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3}
Restriction: Nu > 0.0 && Nu < 1.0

▷ **NumClasses** (input_control) . . . integer ⇝ *integer*
Number of classes.
Default: 5
Suggested values: NumClasses ∈ {2, 3, 4, 5, 6, 7, 8, 9, 10}
Restriction: NumClasses >= 1

▷ **Mode** (input_control) . . . string ⇝ *string*
The mode of the SVM.
Default: 'one-versus-one'
List of values: Mode ∈ {'novelty-detection', 'one-versus-all', 'one-versus-one'}

▷ **Preprocessing** (input_control) . . . string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
Default: 'normalization'
List of values: Preprocessing ∈ {'none', 'normalization', 'principal_components', 'canonical_variates'}

▷ **NumComponents** (input_control) . . . integer ⇝ *integer*
Preprocessing parameter: Number of transformed features (ignored for Preprocessing = *'none'* and

Preprocessing = *'normalization'*).
Default: 10
Suggested values: NumComponents ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Restriction: NumComponents >= 1

▷ **SVMHandle** (output_control) ..... class_svm ⇝ *handle*
SVM handle.

Example

```
create_class_svm (NumFeatures, 'rbf', 0.01, 0.01, NumClasses,\
                  'one-versus-all', 'normalization', NumFeatures,\
                  SVMHandle)
* Generate and add the training data
for J := 0 to NumData-1 by 1
    * Generate training features and classes
    * Data = [...]
    * Class = ...
    add_sample_class_svm (SVMHandle, Data, Class)
endfor
* Train the SVM
train_class_svm (SVMHandle, 0.001, 'default')
* Use the SVM to classify unknown data
for J := 0 to N-1 by 1
    * Extract features
    * Features = [...]
    classify_class_svm (SVMHandle, Features, 1, Class)
endfor
```

Result

If the parameters are valid the operator create_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

add_sample_class_svm

Alternatives

read_dl_classifier, create_class_mlp, create_class_gmm

See also

clear_class_svm, train_class_svm, classify_class_svm

References

Bernhard Schölkopf, Alexander J.Smola: "Learning with Kernels"; MIT Press, London; 1999.

John Shawe-Taylor, Nello Cristianini: "Kernel Methods for Pattern Analysis"; Cambridge University Press, Cambridge; 2004.

Module

Foundation

deserialize_class_svm (: : SerializedItemHandle : SVMHandle)

Deserialize a serialized support vector machine (SVM).

deserialize_class_svm deserializes a support vector machine (SVM) (including its training samples), that was serialized by serialize_class_svm (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized support vector machine is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created support vector machine with the handle SVMHandle.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **SVMHandle** (output_control) ..... class_svm ⇝ *handle*
SVM handle.

Result

If the parameters are valid, the operator deserialize_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_class_svm

Possible Successors

classify_class_svm, create_class_lut_svm

See also

create_class_svm, write_class_svm, serialize_class_svm

Module

Foundation

evaluate_class_svm (: : SVMHandle, Features : Result)

Evaluate a feature vector by a support vector machine.

evaluate_class_svm calculates for a feature vector provided in Features the Result given a SVM in SVMHandle. The operator evaluate_class_svm can only be used if the SVM was created in the Mode = *'novelty-detection'*. If the feature vector lies in the class, a Result value bigger *1.0* is returned. If the feature vector lies outside the class boundary, e.g., is an outlier, a value smaller *1.0* is returned.

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **Features** (input_control) ..... real-array ⇝ *real*
Feature vector.

▷ **Result** (output_control) ..... real(-array) ⇝ *real*
Result of evaluating the feature vector with the SVM.

Result

If the parameters are valid the operator evaluate_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_svm, read_class_svm

See also

create_class_svm

Module

Foundation

get_class_train_data_svm (: : SVMHandle : ClassTrainDataHandle)

Get the training data of a support vector machine (SVM).

get_class_train_data_svm gets the training data of a support vector machine (SVM) and returns it in ClassTrainDataHandle.

Parameters

▷ **SVMHandle** (input_control) . . . . . class_svm ⇝ *handle*
Handle of a SVM that contains training data.

▷ **ClassTrainDataHandle** (output_control) . . . . . class_train_data ⇝ *handle*
Handle of the training data of the classifier.

Result

If the parameters are valid, the operator get_class_train_data_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

add_sample_class_svm, read_samples_class_svm

Possible Successors

add_class_train_data_mlp, add_class_train_data_gmm, add_class_train_data_knn

See also

create_class_train_data

Module

Foundation

get_params_class_svm (: : SVMHandle : NumFeatures, KernelType, KernelParam, Nu, NumClasses, Mode, Preprocessing, NumComponents)

Return the parameters of a support vector machine.

get_params_class_svm returns the parameters of a support vector machine (SVM) that were specified when the SVM was created with create_class_svm. This is particularly useful if the SVM was read from a file with read_class_svm. The output of get_params_class_svm can, for example, be used to check whether the feature vectors and, if necessary, the target data to be used with the SVM have the correct lengths. For a description of the parameters, see create_class_svm.

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **NumFeatures** (output_control) ..... integer ⇝ *integer*
Number of input variables (features) of the SVM.

▷ **KernelType** (output_control) ..... string ⇝ *string*
The kernel type.

▷ **KernelParam** (output_control) ..... real ⇝ *real*
Additional parameter for the kernel.

▷ **Nu** (output_control) ..... real ⇝ *real*
Regularization constant of the SVM.

▷ **NumClasses** (output_control) ..... integer ⇝ *integer*
Number of classes of the test data.

▷ **Mode** (output_control) ..... string ⇝ *string*
The mode of the SVM.

▷ **Preprocessing** (output_control) ..... string ⇝ *string*
Type of preprocessing used to transform the feature vectors.

▷ **NumComponents** (output_control) ..... integer ⇝ *integer*
Preprocessing parameter: Number of transformed features (ignored for `Preprocessing` = *'none'* and `Preprocessing` = *'normalization'*).

Result

If the parameters are valid the operator `get_params_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_class_svm`, `read_class_svm`

Possible Successors

`add_sample_class_svm`, `train_class_svm`

See also

`classify_class_svm`

Module

Foundation

get_prep_info_class_svm (: : SVMHandle, Preprocessing : InformationCont, CumInformationCont)

Compute the information content of the preprocessed feature vectors of a support vector machine

`get_prep_info_class_svm` computes the information content of the training vectors that have been transformed with the preprocessing given by `Preprocessing`. `Preprocessing` can be set to *'principal_components'* or *'canonical_variates'*. The preprocessing methods are described with `create_class_svm`. The information content is derived from the variations of the transformed components of the feature vector, i.e., it is computed solely based on the training data, independent of any error rate on the training data. The information content is computed for all relevant components of the transformed feature vectors (`NumFeatures` for *'principal_components'* and $\min(\texttt{NumClasses} - 1, \texttt{NumFeatures})$ for *'canonical_variates'*, see `create_class_svm`), and is returned in `InformationCont` as a number between 0 and 1. To convert the information content into a percentage, it simply needs to be multiplied by 100. The cumulative information content of the first n components is returned in the n-th component of `CumInformationCont`, i.e., `CumInformationCont` contains the sums of the first n elements of `InformationCont`. To use

get_prep_info_class_svm, a sufficient number of samples must be added to the support vector machine (SVM) given by SVMHandle by using add_sample_class_svm or read_samples_class_svm.

InformationCont and CumInformationCont can be used to decide how many components of the transformed feature vectors contain relevant information. An often used criterion is to require that the transformed data must represent x% (e.g., 90%) of the data. This can be decided easily from the first value of CumInformationCont that lies above x%. The number thus obtained can be used as the value for NumComponents in a new call to create_class_svm. The call to get_prep_info_class_svm already requires the creation of an SVM, and hence the setting of NumComponents in create_class_svm to an initial value. However, when get_prep_info_class_svm is called, it is typically not known how many components are relevant, and hence how to set NumComponents in this call. Therefore, the following two-step approach should typically be used to select NumComponents: In a first step, an SVM with the maximum number for NumComponents is created (NumFeatures for *'principal_components'* and $\min(\texttt{NumClasses}-1, \texttt{NumFeatures})$ for *'canonical_variates'*). Then, the training samples are added to the SVM and are saved in a file using write_samples_class_svm. Subsequently, get_prep_info_class_svm is used to determine the information content of the components, and with this NumComponents. After this, a new SVM with the desired number of components is created, and the training samples are read with read_samples_class_svm. Finally, the SVM is trained with train_class_svm.

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **Preprocessing** (input_control) ..... string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
Default: 'principal_components'
List of values: Preprocessing ∈ {'principal_components', 'canonical_variates'}

▷ **InformationCont** (output_control) ..... real-array ⇝ *real*
Relative information content of the transformed feature vectors.

▷ **CumInformationCont** (output_control) ..... real-array ⇝ *real*
Cumulative information content of the transformed feature vectors.

Example

```
* Create the initial SVM
create_class_svm (NumFeatures, 'rbf', 0.01, 0.01, NumClasses,\
                  'one-versus-all', 'normalization', NumFeatures,\
                  SVMHandle)
* Generate and add the training data
for J := 0 to NumData-1 by 1
    * Generate training features and classes
    * Data = [...]
    * Class = ...
    add_sample_class_svm (SVMHandle, Data, Class)
endfor
write_samples_class_svm (SVMHandle, 'samples.mtf')
* Compute the information content of the transformed features
get_prep_info_class_svm (SVMHandle, 'principal_components',\
                         InformationCont, CumInformationCont)
* Determine NumComp by inspecting InformationCont and CumInformationCont
* NumComp = [...]
* Create the actual SVM
create_class_svm (NumFeatures, 'rbf', 0.01, 0.01, NumClasses, \
                  'one-versus-all', 'principal_components', \
                  NumComp, SVMHandle)
* Train the SVM
read_samples_class_svm (SVMHandle, 'samples.mtf')
train_class_svm (SVMHandle, 0.001, 'default')
write_class_svm (SVMHandle, 'classifier.svm')
```

Result

If the parameters are valid the operator get_prep_info_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

get_prep_info_class_svm may return the error 9211 (Matrix is not positive definite) if Preprocessing = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_svm, read_samples_class_svm

Possible Successors

clear_class_svm, create_class_svm

References

Christopher M. Bishop: "Neural Networks for Pattern Recognition"; Oxford University Press, Oxford; 1995.

Andrew Webb: "Statistical Pattern Recognition"; Arnold, London; 1999.

Module

Foundation

get_sample_class_svm (: : SVMHandle, IndexSample : Features, Target)

Return a training sample from the training data of a support vector machine.

get_sample_class_svm reads out a training sample from the support vector machine (SVM) given by SVMHandle that was added with add_sample_class_svm or read_samples_class_svm. The index of the sample is specified with IndexSample. The index is counted from 0, i.e., IndexSample must be a number between 0 and NumSamples − 1, where NumSamples can be determined with get_sample_num_class_svm. The training sample is returned in Features and Target. Features is a feature vector of length NumFeatures (see create_class_svm), while Target is the index of the class, ranging between 0 and NumClasses-1 (see add_sample_class_svm).

get_sample_class_svm can, for example, be used to reclassify the training data with classify_class_svm in order to determine which training samples, if any, are classified incorrectly.

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **IndexSample** (input_control) ..... integer ⇝ *integer*
Number of the stored training sample.

▷ **Features** (output_control) ..... real-array ⇝ *real*
Feature vector of the training sample.

▷ **Target** (output_control) ..... integer ⇝ *integer*
Target vector of the training sample.

Example

```
* Train an SVM
create_class_svm (NumFeatures, 'rbf', 0.01, 0.01, NumClasses,\
                  'one-versus-all', 'normalization', NumFeatures,\
                  SVMHandle)
read_samples_class_svm (SVMHandle, 'samples.mtf')
train_class_svm (SVMHandle, 0.001, 'default')
* Reclassify the training samples
```

```
get_sample_num_class_svm (SVMHandle, NumSamples)
for I := 0 to NumSamples-1 by 1
    get_sample_class_svm (SVMHandle, I, Data, Target)
    classify_class_svm (SVMHandle, Data, 1, Class)
    if (Class != Target)
        * Sample has been classified incorrectly
    endif
endfor
```

Result

If the parameters are valid the operator `get_sample_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`add_sample_class_svm`, `read_samples_class_svm`, `get_sample_num_class_svm`, `get_support_vector_class_svm`

Possible Successors

`classify_class_svm`

See also

`create_class_svm`

Module

Foundation

get_sample_num_class_svm (: : SVMHandle : NumSamples)

Return the number of training samples stored in the training data of a support vector machine.

`get_sample_num_class_svm` returns in `NumSamples` the number of training samples that are stored in the support vector machine (SVM) given by `SVMHandle`. `get_sample_num_class_svm` should be called before the individual training samples are accessed with `get_sample_class_svm`, e.g., for the purpose of reclassifying the training data (see `get_sample_class_svm`).

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **NumSamples** (output_control) ..... integer ⇝ *integer*
Number of stored training samples.

Result

If `SVMHandle` is valid the operator `get_sample_num_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`add_sample_class_svm`, `read_samples_class_svm`

Possible Successors

get_sample_class_svm

See also

create_class_svm

Module

Foundation

get_support_vector_class_svm (: : SVMHandle, IndexSupportVector : Index)

Return the index of a support vector from a trained support vector machine.

The operator get_support_vector_class_svm maps a support vector of a trained SVM (given in SVMHandle) to the original training data set. The index of the SV is specified with IndexSupportVector. The index is counted from 0, i.e., IndexSupportVector must be a number between 0 and NumSupportVectors − 1, where NumSupportVectors can be determined with get_support_vector_num_class_svm. The index of this SV in the training data is returned in Index. This Index can be used for a query with get_sample_class_svm to obtain the feature vectors that become support vectors. get_sample_class_svm can, for example, be used to visualize the support vectors.

Note that when using train_class_svm with a mode different from *'default'* or reducing the SVM with reduce_class_svm, the returned Index will always be *-1*, i.e., it will be invalid. The reason for this is that a consistent mapping between SV and training data becomes impossible.

Parameters

- ▷ **SVMHandle** (input_control) . . . . . class_svm ⇝ *handle*
  SVM handle.
- ▷ **IndexSupportVector** (input_control) . . . . . integer ⇝ *integer*
  Index of the stored support vector.
- ▷ **Index** (output_control) . . . . . real ⇝ *real*
  Index of the support vector in the training set.

Result

If the parameters are valid the operator get_sample_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_svm, get_support_vector_num_class_svm

Possible Successors

get_sample_class_svm

See also

create_class_svm

Module

Foundation

get_support_vector_num_class_svm (: : SVMHandle : NumSupportVectors, NumSVPerSVM)

Return the number of support vectors of a support vector machine.

get_support_vector_num_class_svm returns in NumSupportVectors the number of support vectors that are stored in the support vector machine (SVM) given by SVMHandle. get_support_vector_num_class_svm should be called before the labels of individual support vectors are read out with get_support_vector_class_svm, e.g., for visualizing which the training data become a SV (see get_support_vector_class_svm). The number of SVs in each classifier is listed in NumSVPerSVM. The reason that its sum differs from the Number obtained in NumSupportVectors is that SV evaluations are reused throughout different sub-classifiers. NumSVPerSVM provides the possibility for controlling the process of speeding up SVM classification time with the operator reduce_class_svm.

Parameters

- ▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
  SVM handle.
- ▷ **NumSupportVectors** (output_control) ..... integer ⇝ *integer*
  Total number of support vectors.
- ▷ **NumSVPerSVM** (output_control) ..... integer-array ⇝ *integer*
  Number of SV of each sub-SVM.

Result

If SVMHandle is valid the operator get_sample_num_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_svm

Possible Successors

get_sample_class_svm

See also

create_class_svm

Module

Foundation

read_class_svm (: : FileName : SVMHandle)

Read a support vector machine from a file.

read_class_svm reads a support vector machine (SVM) that has been stored with write_class_svm. Since the training of an SVM can consume a relatively long time, the SVM is typically trained in an offline process and written to a file with write_class_svm. In the online process the SVM is read with read_class_svm and subsequently used for classification with classify_class_svm. The default HALCON file extension for the SVM classifier is 'gsc'.

Parameters

- ▷ **FileName** (input_control) ..... filename.read ⇝ *string*
  File name.
  File extension: .gsc
- ▷ **SVMHandle** (output_control) ..... class_svm ⇝ *handle*
  SVM handle.

Result

If the parameters are valid the operator read_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

classify_class_svm, create_class_lut_svm

Alternatives

read_dl_classifier

See also

create_class_svm, write_class_svm

Module

Foundation

read_samples_class_svm (: : SVMHandle, FileName :)

Read the training data of a support vector machine from a file.

read_samples_class_svm reads training samples from the file given by FileName and adds them to the training samples that have already been added to the support vector machine (SVM) given by SVMHandle. The SVM must be created with create_class_svm before calling read_samples_class_svm. As described with train_class_svm and write_samples_class_svm, the operators read_samples_class_svm, add_sample_class_svm, and write_samples_class_svm can be used to build up a extensive set of training samples, and hence to improve the performance of the SVM by retraining the SVM with extended data sets.

It should be noted that the training samples must have the correct dimensionality. The feature vectors and target vectors stored in FileName must have the lengths NumFeatures and NumClasses that were specified with create_class_svm. The target is stored in vector form for compatibility reason with the MLP (see read_samples_class_mlp). If the dimensions are incorrect an error message is returned.

Parameters

▷ **SVMHandle** (input_control) . . . . . class_svm ⇝ *handle*
SVM handle.

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.

Result

If the parameters are valid the operator read_samples_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SVMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_svm

Possible Successors

train_class_svm

Alternatives

add_sample_class_svm

See also

write_samples_class_svm, clear_samples_class_svm

Module

Foundation

reduce_class_svm (: : SVMHandle, Method, MinRemainingSV, MaxError : SVMHandleReduced)

Approximate a trained support vector machine by a reduced support vector machine for faster classification.

As described in create_class_svm, the classification time of a SVM depends on the number of kernel evaluations between the support vectors and the feature vectors. While the length of the data vectors can be reduced in a preprocessing step like *'principal_components'* or *'canonical_variates'* (see create_class_svm for details), the number of resulting SV depends on the complexity of the classification problem. The number of SVs is determined during training. To further reduce classification time, the number of SVs can be reduced by approximating the original separating hyperplane with fewer SVs than originally required. For this purpose, a copy of the original SVM provided by SVMHandle is created and returned in SVMHandleReduced. This new SVM has the same parametrization as the original SVM, but a different SV expansion. The training samples that are included in SVMHandle are not copied. The original SVM is not modified by reduce_class_svm.

The reduction method is selected with Method. Currently, only a bottom up approach is supported, which iteratively merges SVs. The algorithm stops if either the minimum number of SVs is reached (MinRemainingSV) or if the accumulated maximum error exceeds the threshold MaxError. Note that the approximation reduces the complexity of the hyperplane and thereby leads to a deteriorated classification rate. A common approach is therefore to start from a small MaxError e.g., *0.001*, and to increase its value step by step. To control the reduction ratio, at each step the number of remaining SVs is determined with get_support_vector_num_class_svm and the classification rate is checked on a separate test data set with classify_class_svm.

Parameters

- ▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
  Original SVM handle.
- ▷ **Method** (input_control) ..... string ⇝ *string*
  Type of postprocessing to reduce number of SV.
  Default: 'bottom_up'
  List of values: Method ∈ {'bottom_up'}
- ▷ **MinRemainingSV** (input_control) ..... integer ⇝ *integer*
  Minimum number of remaining SVs.
  Default: 2
  Suggested values: MinRemainingSV ∈ {2, 3, 4, 5, 7, 10, 15, 20, 30, 50}
  Restriction: MinRemainingSV >= 2
- ▷ **MaxError** (input_control) ..... real ⇝ *real*
  Maximum allowed error of reduction.
  Default: 0.001
  Suggested values: MaxError ∈ {0.0001, 0.0002, 0.0005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05}
  Restriction: MaxError > 0.0
- ▷ **SVMHandleReduced** (output_control) ..... class_svm ⇝ *handle*
  SVMHandle of reduced SVM.

Example

```
* Train an SVM
create_class_svm (NumFeatures, 'rbf', 0.01, 0.01, NumClasses,\
                  'one-versus-all', 'normalization', NumFeatures,\
                  SVMHandle)
read_samples_class_svm (SVMHandle, 'samples.mtf')
train_class_svm (SVMHandle, 0.001, 'default')
```

```
* Create a reduced SVM
reduce_class_svm (SVMHandle, 'bottom_up', 2, 0.01, SVMHandleReduced)
write_class_svm (SVMHandleReduced, 'classifier.svm')
```

Result

If the parameters are valid the operator `train_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`train_class_svm`, `get_support_vector_num_class_svm`

Possible Successors

`classify_class_svm`, `write_class_svm`, `get_support_vector_num_class_svm`

See also

`train_class_svm`

Module

Foundation

select_feature_set_svm (: : ClassTrainDataHandle, SelectionMethod, GenParamName, GenParamValue : SVMHandle, SelectedFeatureIndices, Score)

Selects an optimal combination of features to classify the provided data.

`select_feature_set_svm` selects an optimal subset from a set of features to solve a given classification problem. The classification problem has to be specified with annotated training data in `ClassTrainDataHandle` and will be classified by a support vector machine (SVM). Details of the properties of this classifier can be found in `create_class_svm`.

The result of the operator is a trained classifier that is returned in `SVMHandle`. Additionally, the list of indices or names of the selected features is returned in `SelectedFeatureIndices`. To use this classifier, calculate for new input data all features mentioned in `SelectedFeatureIndices` and pass them to the classifier.

A possible application of this operator can be a comparison of different parameter sets for certain feature extraction techniques. Another application is to search for a feature that is discriminating between different classes.

Additionally, the values for *'nu'* and *'gamma'* can be estimated for the SVM. To only estimate these two parameters without altering the feature set, the feature vector has to be specified as one large subfeature.

To define the features that should be selected from `ClassTrainDataHandle`, the dimensions of the feature vectors in `ClassTrainDataHandle` can be grouped into subfeatures by calling `set_feature_lengths_class_train_data`. A subfeature can contain several subsequent elements of a feature vector. The operator decides for each of these subfeatures, if it is better to use it for the classification or leave it out.

The indices of the selected subfeatures are returned in `SelectedFeatureIndices`. If names were set in `set_feature_lengths_class_train_data`, these names are returned instead of the indices. If `set_feature_lengths_class_train_data` was not called for `ClassTrainDataHandle` before, each element of the feature vector is considered as a subfeature.

The selection method `SelectionMethod` is either a greedy search *'greedy'* (iteratively add the feature with highest gain) or the dynamically oscillating search *'greedy_oscillating'* (add the feature with highest gain and test then if any of the already added features can be left out without great loss). The method *'greedy'* is generally preferable, since it is faster. Only in cases when the subfeatures are low-dimensional or redundant, the method *'greedy_oscillating'* should be chosen.

The optimization criterion is the classification rate of a two-fold cross-validation of the training data. The best achieved value is returned in `Score`.

The parameters *'nu'* and *'gamma'* for the SVM that is used to classify can be set to *'auto'* by using the parameters `GenParamName` and `GenParamValue`. If they are set to *'auto'*, the estimated optimal *'nu'* and/or *'gamma'* is estimated. The automatic estimation of *'nu'* and *'gamma'* can take a substantial amount of time (up to days, depending on the data set and the number of features).

Additionally, there is the parameter *'mode'* which can be either set to *'one-versus-all'* or *'one-versus-one'*. An explanation of the two modes as well as of the parameters *'nu'* and *'gamma'* as the kernel parameter of the radial basis function (RBF) kernel can be found in `create_class_svm`.

Attention

This operator may take considerable time, depending on the size of the data set in the training file, and the number of features.

Please note, that this operator should not be called, if only a small set of training data is available. Due to the risk of overfitting the operator `select_feature_set_svm` may deliver a classifier with a very high score. However, the classifier may perform poorly when tested.

Parameters

- ▷ **`ClassTrainDataHandle`** (input_control) . . . . . class_train_data ⇝ *handle*
  Handle of the training data.
- ▷ **`SelectionMethod`** (input_control) . . . . . string ⇝ *string*
  Method to perform the selection.
  Default: 'greedy'
  List of values: `SelectionMethod` ∈ {'greedy', 'greedy_oscillating'}
- ▷ **`GenParamName`** (input_control) . . . . . string(-array) ⇝ *string*
  Names of generic parameters to configure the selection process and the classifier.
  Default: []
  List of values: `GenParamName` ∈ {'nu', 'gamma', 'mode'}
- ▷ **`GenParamValue`** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
  Values of generic parameters to configure the selection process and the classifier.
  Default: []
  Suggested values: `GenParamValue` ∈ {0.02, 0.05, 'auto', 'one-versus-one', 'one-versus-all'}
- ▷ **`SVMHandle`** (output_control) . . . . . class_svm ⇝ *handle*
  A trained SVM classifier using only the selected features.
- ▷ **`SelectedFeatureIndices`** (output_control) . . . . . string-array ⇝ *string*
  The selected feature set, contains indices.
- ▷ **`Score`** (output_control) . . . . . real-array ⇝ *real*
  The achieved score using two-fold cross-validation.

Example

```
* Find out which of the two features distinguishes two Classes
NameFeature1 := 'Good Feature'
NameFeature2 := 'Bad Feature'
LengthFeature1 := 3
LengthFeature2 := 2
* Create training data
create_class_train_data (LengthFeature1+LengthFeature2,\
  ClassTrainDataHandle)
* Define the features which are in the training data
set_feature_lengths_class_train_data (ClassTrainDataHandle, [LengthFeature1,\
  LengthFeature2], [NameFeature1, NameFeature2])
* Add training data
*                                                    |Feat1| |Feat2|
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  2,1  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  2,1  ], 1)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [1,1,1,  3,4  ], 0)
add_sample_class_train_data (ClassTrainDataHandle, 'row', [2,2,2,  3,4  ], 1)
```

```
* Add more data
* ...
* Select the better feature with a SVM
select_feature_set_svm (ClassTrainDataHandle, 'greedy', [], [], SVMHandle,\
  SelectedFeatureSVM, Score)
* Use the classifier
* ...
```

Result

If the parameters are valid, the operator `select_feature_set_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`create_class_train_data`, `add_sample_class_train_data`, `set_feature_lengths_class_train_data`

Possible Successors

`classify_class_svm`

Alternatives

`select_feature_set_mlp`, `select_feature_set_knn`, `select_feature_set_gmm`

See also

`select_feature_set_trainf_svm`, `gray_features`, `region_features`

Module

Foundation

serialize_class_svm (: : SVMHandle : SerializedItemHandle)

Serialize a support vector machine (SVM).

`serialize_class_svm` serializes a support vector machine (SVM) and its stored training samples (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The same data that is written in a file by `write_class_svm` and `write_samples_class_svm` is converted to a serialized item. The support vector machine is defined by the handle `SVMHandle`. The serialized support vector machine is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_class_svm`.

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

train_class_svm

Possible Successors

clear_class_svm, fwrite_serialized_item, send_serialized_item, deserialize_class_svm

See also

create_class_svm, read_class_svm, write_samples_class_svm, deserialize_class_svm

Module

Foundation

train_class_svm (: : SVMHandle, Epsilon, TrainMode :)

Train a support vector machine.

train_class_svm trains the support vector machine (SVM) given in SVMHandle. Before the SVM can be trained, the training samples to be used for the training must be added to the SVM using add_sample_class_svm or read_samples_class_svm.

Technically, training an SVM means solving a convex quadratic optimization problem. This implies that it can be assured that training terminates after finite steps at the global optimum. In order to recognize termination, the gradient of the function that is optimized internally must fall below a threshold, which is set in Epsilon. By default, a value of *0.001* should be used for Epsilon since this yields the best results in practice. A too big value leads to a too early termination and might result in suboptimal solutions. With a too small value the optimization requires a longer time, often without changing the recognition rate significantly. Nevertheless, if longer training times are possible, a smaller value than *0.001* might be chosen. There are two common reasons for changing Epsilon: First, if you specified a very small value for Nu when calling (create_class_svm), e.g., Nu = *0.001*, a smaller Epsilon might significantly improve the recognition rate. A second case is the determination of the optimal kernel function and its parametrization (e.g., the KernelParam-Nu pair for the RBF kernel) with the computationally intensive n-fold cross validation. Here, choosing a bigger Epsilon reduces the computational time without changing the parameters of the optimal kernel that would be obtained when using the default Epsilon. After the optimal KernelParam-Nu pair is obtained, the final training is conducted with a small Epsilon.

The duration of the training depends on the training data, in particular on the number of resulting support vectors (SVs), and Epsilon. It can lie between seconds and several hours. It is therefore recommended to choose the SVM parameter Nu in create_class_svm so that as few SVs as possible are generated without decreasing the recognition rate. Special care must be taken with the parameter Nu in create_class_svm so that the optimization starts from a feasible region. If too many training errors are chosen with a too big Nu, an exception is raised. In this case, an SVM with the same training data, but with smaller Nu must be trained.

With the parameter TrainMode you can choose between different training modes. Normally, you train an SVM without additional information and TrainMode is set to *'default'*. If multiple SVMs for the same data set but with different kernels are trained, subsequent training runs can reuse optimization results and thus speedup the overall training time of all runs. For this mode, in TrainMode a SVM handle of a previously trained SVM is passed. Note that the SVM handle passed in SVMHandle and the SVMHandle passed in TrainMode must have the same training data, the same mode and the same number of classes (see create_class_svm). The application for this training mode is the evaluation of different kernel functions given the same training set. In the literature this is referred to as alpha seeding.

With TrainMode = *'add_sv_to_train_set'* it is possible to append the support vectors that were generated by a previous call of train_class_svm to the currently saved training set. This mode has two typical application areas: First, it is possible to gradually train a SVM. For this, the complete training set is divided into disjunctive chunks. The first chunk is trained normally using TrainMode = *'default'*. Afterwards, the previous training set is removed with clear_samples_class_svm, the next chunk is added with add_sample_class_svm and trained with TrainMode = *'add_sv_to_train_set'*. This is repeated until all chunks are trained. This approach has the advantage that even huge training data sets can be trained efficiently with respect to memory consumption. A second application area for this mode is that a general purpose classifier can be specialized by adding characteristic

training samples and then retraining it. Please note that the preprocessing (as described in create_class_svm) is not changed when training with TrainMode = *'add_sv_to_train_set'*.

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **Epsilon** (input_control) ..... real ⇝ *real*
Stop parameter for training.
Default: 0.001
Suggested values: Epsilon ∈ {0.00001, 0.0001, 0.001, 0.01, 0.1}

▷ **TrainMode** (input_control) ..... number ⇝ *string* / integer
Mode of training. For normal operation: 'default'. If SVs already included in the SVM should be used for training: 'add_sv_to_train_set'. For alpha seeding: the respective SVM handle.
Default: 'default'
List of values: TrainMode ∈ {'default', 'add_sv_to_train_set'}

Example

```
* Train an SVM
create_class_svm (NumFeatures, 'rbf', 0.01, 0.01, NumClasses,\
                  'one-versus-all', 'normalization', NumFeatures,\
                  SVMHandle)
read_samples_class_svm (SVMHandle, 'samples.mtf')
train_class_svm (SVMHandle, 0.001, 'default')
write_class_svm (SVMHandle, 'classifier.svm')
```

Result

If the parameters are valid the operator train_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SVMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

add_sample_class_svm, read_samples_class_svm

Possible Successors

classify_class_svm, write_class_svm, create_class_lut_svm

Alternatives

train_dl_classifier_batch, read_class_svm

See also

create_class_svm

References

John Shawe-Taylor, Nello Cristianini: "Kernel Methods for Pattern Analysis"; Cambridge University Press, Cambridge; 2004.

Bernhard Schölkopf, Alexander J.Smola: "Learning with Kernels"; MIT Press, London; 1999.

Module

Foundation

write_class_svm (: : SVMHandle, FileName :)

Write a support vector machine to a file.

write_class_svm writes the support vector machine (SVM) SVMHandle to the file given by FileName. The default HALCON file extension for the SVM classifier is 'gsc'. write_class_svm is typically called after the SVM has been trained with train_class_svm. The SVM can be read with read_class_svm. write_class_svm does *not* write any training samples that possibly have been stored in the SVM. For this purpose, write_samples_class_svm should be used.

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
File name.
File extension: .gsc

Result

If the parameters are valid the operator write_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_class_svm

Possible Successors

clear_class_svm

See also

create_class_svm, read_class_svm, write_samples_class_svm

Module

Foundation

write_samples_class_svm (: : SVMHandle, FileName :)

Write the training data of a support vector machine to a file.

write_samples_class_svm writes the training samples currently stored in the support vector machine (SVM) SVMHandle to the file given by FileName. write_samples_class_svm can be used to build up a database of training samples, and hence to improve the performance of the SVM by training it with an extended data set (see train_class_svm). The file FileName is overwritten by write_samples_class_svm. Nevertheless, extending the database of training samples is easy to do because read_samples_class_svm and add_sample_class_svm add the training samples to the training samples that are already stored in memory with the SVM.

Parameters

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
File name.

Result

If the parameters are valid the operator write_samples_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_class_svm

Possible Successors

clear_samples_class_svm

See also

create_class_svm, get_prep_info_class_svm, read_samples_class_svm

Module

Foundation

Chapter 8

Control

assign (: : Input : Result)

Assign a new value to a variable.

assign assigns a new value to a variable.

In the full text editor, an assignment is simply entered with the help of the assignment operator ':=', e.g.:

```
u := sin(x) + cos(y)
```

This is equivalent to the C syntax assignment:

```
u = sin(x) + cos(y);
```

If the operator window is used for entering an assignment, assign must be entered into the operator combo box as an operator name. This opens the parameter area, where the parameter Input represents the expression that has to be evaluated to one value and assigned to the variable, i.e., this is the right side of the assignment. The parameter Result gets the name of the variable, i.e., this is the left side of assignment.

Attention

In addition to the parameter type control, which is indicated in the parameter description, assign also supports iconic variables and vector variables. For an assignment, the parameter types of the two parameters Input and Result must be identical. For the assignment of iconic objects, the operator copy_obj is used internally.

Parameters

▷ **Input** (input_control) ..... real(-array) ⇝ *real* / integer / string
New value.
Default: 1

▷ **Result** (output_control) ..... real(-array) ⇝ *real* / integer / string
Variable that has to be changed.

Example

```
Tuple1 := [1,0,3,4,5,6,7,8,9]
Val := sin(1.2) + cos(1.2)
Tuple2 := []
```

Result

If the expression is correct assign returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Alternatives

insert

Module

Foundation

assign_at (: : Index, Value : Result)

Assignment of one or several values to one or several tuple elements.

assign_at assigns a single value to one or several elements of a tuple, or it assigns a number of values elementwise to the specified elements of the output tuple. All other elements of the output tuple keep their values. If the passed indices are out of the current range of the output tuple, the tuple is increased and the new values are initialized to a default value.

In the full text editor an assign_at operation is simply entered with the help of the assignment operator symbol ':=' and the index access operator symbol '[ ]' following the output variable. The Index parameter can be any expression that evaluates to any number of positive integer values. The Value parameter must evaluate to exactly one value or to the same number of indices that are provided via the Index parameter, e.g.:

```
Areas[Radius-1]     := Area
Areas[0,4,|Rad|-1] := 0
FileNames[0,2,4]    := ['f1','f2','f3']
```

The operator assign_at replaces and extends the modifying version of the old insert operator.

Parameters

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *integer*
Indices of the elements that have to be replaced by the new value(s).
Default: 0
Suggested values: Index ∈ {0, 1, 2, 3, 4, 5, 6}
Minimum increment: 1

▷ **Value** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Value(s) that is to be assigned.
Default: 1

▷ **Result** (output_control) . . . . . tuple(-array) ⇝ *real* / integer / string
Result tuple containing the assigned values.

Result

If the expression is correct assign_at returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Alternatives

assign, tuple_replace

Module

Foundation

break (: : :)

Terminate loop execution or leave a switch block.

break terminates the smallest enclosing for, while, or repeat..until loop. In addition, the break statement is used to leave a switch block, in particular at the end of a case branch. The program execution is continued at the program line following the corresponding block.

break statements that are not enclosed by a loop or switch block are invalid.

Example

```
read_image (Image, 'monkey')
threshold (Image, Region, 160, 180)
connection (Region, Regions)
Number := |Regions|
AllRegionsValid := 1
* check if for all regions area <=30
for i := 1 to Number by 1
```

```
  select_obj (Regions, ObjectSelected, i)
  area_center (ObjectSelected, Area, Row, Column)
  if (Area > 30)
    AllRegionsValid := 0
    break
  endif
endfor
```

Result

`break` (as an operator) always returns `2` (`H_MSG_TRUE`).

Alternatives

`continue`

See also

`for`, `while`, `repeat`, `until`, `switch`, `case`

Module

Foundation

`case` (: : Constant :)

Jump label that starts a branch within a `switch` block.

`case` defines a jump label within a `switch` block. It starts a branch that is executed if the value of the control expression of the `switch` statement matches the constant integer expression that is defined in `Constant`. For this parameter only constant integer expressions are accepted. Variable expressions and other data types are not allowed.

As in the programming languages C, C++, and C# the `case` statement does not open a block that is automatically left at the next `case` or `default` statement. In contrast, it works just like a goto label that is accessed if the label matches. In order to leave a `case` branch and continue execution after the end of the `switch` block, the `break` statement can be used anywhere within the `switch` block.

Parameters

▷ **`Constant`** (input_control) . . . . . integer ⇝ *integer*
Constant integer expressions that determines for which value of the switch control expression the branch is accessed.
Default: 1

Result

`case` (as an operator) always returns `2` (`H_MSG_TRUE`).

Alternatives

`elseif`

See also

`switch`, `default`, `endswitch`, `if`

Module

Foundation

`catch` (: : : Exception)

Catches exceptions that were thrown in the preceding `try` block.

With the help of the operators `try`, `catch`, `endtry`, and `throw` it is possible to implement a dynamic exception handling in HDevelop, which is comparable to the exception handling in C++ and C#. The basic concepts of the exception handling in HDevelop are described at the operators `try`, `throw`, and `dev_set_check` as well as in the "`HDevelop User's Guide`".

The operator `catch` ends a block of watched program lines and starts a block of program lines that have to be executed in an error case. If the `try`-`catch` block is executed without an exception, the `catch`-`endtry` block

is ignored and program execution continues after the corresponding endtry operator. In contrast, in an error case the program execution jumps directly from the operator where the error occurred (or from the throw operator) to the catch operator of the surrounding try-catch block. The output control parameter Exception returns a tuple that contains a predefined set of data describing the error in case an operator error occurred. If the exception was thrown by the throw operator, an arbitrary user-defined tuple can be returned.

The most important data within the Exception tuple is the error code. Therefore, this is passed as the first item of the Exception tuple and can be accessed directly with Exception[0]. However, all other data has to be accessed through the operator dev_get_exception_data, because the order and the extent of the provided data may change in future versions and may vary for different programming language exports. Especially, it has to be taken into account that in the exported code there are some items of the error tuple that are not available and others that might not be determined until they are requested (like error messages).

If the exception was thrown by an operator error, a HALCON error code (< 10000) or if the aborted operator belongs to an extension package, a user-defined error code (> 10000) is returned as the error code. A list of all HALCON error codes can be found in the appendix of the "Extension Package Programmer's Manual". The first element of a user-defined Exception tuple thrown by the operator throw should be an error code $\geq$ 30000. Additional tuple elements can be chosen without any restrictions.

If an operator error occurred within HDevelop or HDevEngine, the following information about the error is provided by the Exception tuple:

- The HALCON error code.
- An additional HDevelop specific error code that specifies whether an error was caught within the HALCON operator (code = 21000) or outside the operator, e.g., during the evaluation and assignment of the parameter expressions. In the latter case the error code specifies the kind of error more precisely.
- The HALCON error message.
- An appropriate HDevelop-specific error message.
- The number of the program line, where the error occurred.
- The name of the operator that threw the exception (if the exception was thrown in a protected procedure, '--protected--' is returned instead of the operator name).
- The depth of the call stack (if the error occurred in 'main' a depth of 1 is returned).
- The name of the procedure, where the error occurred.

In most cases, for an automatic exception handling it is sufficient to use the HALCON error code. Additional data is primarily passed in order to provide some information about the error condition to the developer of the HDevelop program for debugging reasons. Attention: in the exported code, in general, information about the error location will not be available.

Attention

The export of the operators try, catch, endtry, and throw is not supported for the language C, but only for the languages C++, C# and VisualBasic/.NET. Only the latter support throwing exceptions across procedures.

Parameters

▷ **Exception** (output_control) ..... exception-array ⇝ *integer* / string
Tuple returning the exception data.

Result

catch always returns 2 (H_MSG_TRUE).

Possible Successors

dev_get_exception_data

See also

try, endtry, throw, dev_get_exception_data, dev_set_check

Module

Foundation

comment (: : Comment :)

Add a comment of one line to the program.

comment allows to add a comment of one line to the program. As parameter value, i.e., as comment, all characters are allowed. If the operator window is used to enter a comment and if there are newlines in the comment line parameter, one comment statement for every text line is inserted.

In the full text editor a comment is marked by entering an asterisk ('*') as first non-whitespace character.

This operator has no effect on the program execution.

Parameters

▷ **Comment** (input_control) . . . . . string ⇝ *string*
Arbitrary sequence of characters.

Example

```
* This is a program with comments
* 'this is a string as comment'
* here are numbers: 4711, 0.815
stop ()
```

Result

comment is never executed.

Module

Foundation

continue (: : :)

Skip the current loop execution.

continue skips the smallest enclosing for, while or repeat..until loop execution. Program execution is continued at the condition line of the loop or at the next line after the continue statement in case no enclosing loop exists.

Result

continue (as operators) always returns 2 (H_MSG_TRUE).

Alternatives

break

See also

for, while, repeat, until

Module

Foundation

convert_tuple_to_vector_1d (: : InputTuple,
SubTupleLength : ResultVector)

Distribute the elements of a tuple to a vector.

convert_tuple_to_vector_1d transforms a tuple into a vector variable. The input tuple InputTuple is split into sub-tuples each comprising of SubTupleLength elements. The sub-tuples are stored in the output vector ResultVector.

Parameters

▷ **InputTuple** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Input tuple.

▷ **SubTupleLength** (input_control) . . . . . number ⇝ *integer*
Desired length of the resulting tuples in the output vector.
Default: 1

▷ **ResultVector** (output_control) . . . . . number-vector1 ⇝ *real* / integer / string
Output vector.

Result

If the values of the specified parameters are correct, `convert_tuple_to_vector_1d` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

See also

`convert_vector_to_tuple`

Module

Foundation

convert_vector_to_tuple (: : InputVector : ResultTuple)

Concatenate the elements of a vector to a single tuple.

`convert_vector_to_tuple` transforms a vector into a tuple. The elements of the input vector `InputVector` get concatenated and stored in the output tuple `ResultTuple`. If `InputVector` has a dimension of 2 or greater its elements are collected in a depth-first search. E.g., the input vector *'1,2,3,4'* will be turned into the result tuple *[1,2,3,4]*.

Parameters

- ▷ **InputVector** (input_control) ..... number-vector ⇝ *real* / integer / string
  Input vector.
- ▷ **ResultTuple** (output_control) ..... real(-array) ⇝ *real* / integer / string
  Output tuple.

Result

If the values of the specified parameters are correct, `convert_vector_to_tuple` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

See also

`convert_tuple_to_vector_1d`

Module

Foundation

default (: : :)

Alternative branch in a `switch` block.

`default` opens an alternative branch in a `switch` block. This branch is accessed if the calculated control expression of the `switch` statement does not match any of the integer constants of the previous `case` statements.

Result

`default` (as an operator) always returns `2` (`H_MSG_TRUE`).

Alternatives

`case`, `elseif`, `else`

See also

`switch`, `case`, `endswitch`, `if`

Module

Foundation

else (: : :)

Alternative of conditional statement.

`else` continues after an `if` or `elseif` block with an alternative block. If the conditions of all corresponding `if` or `elseif` blocks evaluated to *'false'* (*0*), i.e., none of the corresponding `if` or `elseif` blocks has been executed, the following `else` block is executed.

Result

`else` (as operator) always returns `2` (`H_MSG_TRUE`).

Alternatives

`if`, `elseif`

See also

`until`, `for`, `while`

Module

Foundation

elseif (: : Condition :)

Conditional statement with alternative.

`elseif` is a conditional statement that continues after an `if` or another `elseif` block with an alternative block. The `Condition` parameter must evaluate to a Boolean or integer expression.

If `Condition` evaluates to *'true'* (not *0*), the following block body up to the next corresponding block statement `elseif`, `else`, or `endif` is executed. Reaching the end of the block the execution continues after the corresponding `endif` statement.

If `Condition` evaluates to *'false'* (*0*), the execution is continued at the next corresponding block statement `elseif`, `else`, or `endif`.

Parameters

▷ **Condition** (input_control) ..... integer ⇝ *integer*
Condition for the if statement.
Default: 1

Result

If the condition is correct `elseif` (as operator) returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Alternatives

`if`

See also

`else`, `for`, `while`, `until`

Module

Foundation

endfor (: : :)

End statement of a for loop.

`endfor` is the last statement of a `for` loop.

Result

`endfor` always returns `2` (`H_MSG_TRUE`).

See also

`for`

Module

Foundation

endif (: : :)

End of if command.

endif is the last statement of an if, elseif, or else block.

Result

endif always returns 2 (H_MSG_TRUE).

See also

if

Module

Foundation

endswitch (: : :)

Ends a multiway branch block.

endswitch ends a multiway branch block that has been opened by switch.

Result

endswitch always returns 2 (H_MSG_TRUE).

See also

switch

Module

Foundation

endtry (: : :)

Ends a block where exceptions are handled.

With the help of the operators try, catch, endtry, and throw it is possible to implement a dynamic exception handling in HDevelop, which is comparable to the exception handling in C++ and C#. The basic concepts of the exception handling in HDevelop are described at the operators try, throw, and dev_set_check as well as in the "HDevelop User's Guide".

The operator endtry closes the exception handling block that was opened with the operators try and catch.

Attention

The export of the operators try, catch, endtry, and throw is not supported for the language C, but only for the languages C++, C# and VisualBasic/.NET. Only the latter support throwing exceptions across procedure.

Result

endtry always returns 2 (H_MSG_TRUE).

See also

try, catch, throw, dev_get_exception_data, dev_set_check

Module

Foundation

endwhile (: : :)

End statement of a while loop.

endwhile is the last statement of a while loop.

Result

endwhile always returns 2 (H_MSG_TRUE).

See also

while

Module

Foundation

executable_expression (: : Expression :)

Execute a stand-alone operation.

The HDevelop language contains a few operations that are executed stand-alone, i.e., not as an expression within another operator call. The operator `executable_expression` allows to enter such stand-alone operations into the operator window of HDevelop. In the full text editor however, those operations are entered verbatim.

Currently, the following modifying vector operations are stand-alone and can only be used in an executable expression:

- `.clear()`
- `.insert()`
- `.remove()`

For further details about these operations please refer to the HDevelop User's Guide.

Even though `Expression` formally is presented as a control parameter, nonetheless it is also possible to execute stand-alone operations with iconic vectors.

Parameters

▷ **Expression** (input_control) . . . . . number-vector ⇝ *real* / integer / string
Operation to be executed.

Example

```
read_image (Image1, 'fin1')
read_image (Image2, 'fin2')
ImageVector.insert(1, Image1).insert(2, Image2)
* process vector
ImageVector.clear()
```

Result

If the values of the specified parameters are correct, `executable_expression` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Module

Foundation

exit (: : :)

Terminate HDevelop.

`exit` terminates HDevelop. The operator is equivalent to the menu entry `File` ▷ `Quit`. Internally and for exported C++ code the C-function call `exit(0)` is used.

Example

```
read_image (Image, 'fabrik')
intensity (Image, Image, Mean, Deviation)
open_file ('intensity.txt', 'output', FileHandle)
fwrite_string (FileHandle, Mean + ' ' + Deviation)
close_file (FileHandle)
exit ()
```

Result

`exit` returns 0 (o.k.) to the calling environment of HDevelop = operating system.

See also

`stop`

Module

Foundation

export_def (: : Position, Declaration :)

Insert arbitrary text into the export code of a procedure.

export_def allows to define code lines or text blocks that are written verbatim into the output file of a procedure or program that is exported.

The parameter Position controls the placement of the text given in Declaration. The following options are supported:

'in_place' **- #** The text is inserted in the procedure at the actual place, i.e., in between the neighboring program lines.

'at_file_begin' **- #^^** The text is exported at the very beginning of the exported file.

'before_procedure' **- #^** The text is exported immediately before the procedure it is defined in.

'after_procedure' **- #$** The text is exported immediately after the procedure it is defined in.

'at_file_end' **- #$$** The text is exported at the very end of the exported file.

In the program listing, export_def is not represented in normal operator syntax but marked by a special character sequence. The first character within the line is the export marker # that can be followed by a position marker as listed above. If entering an export definition in the full text editor, please note that there must not be any spaces before #.

For better readability, the export character sequence may be followed by one space character that is not interpreted as part of the export text. All additional spaces are added to the export.

For lines that are exported within the current procedure, the export gets the same indentation as the current program lines get. There is one exception: if the export text starts with # immediately after the export markers or the optional space, the export text will not be indented at all, e.g.:

```
for Index := 1 to 5 by 1
\# \#ifdef MY\_SWITCH
\# int cnt = 100;
  * an optional code block
\# \#endif
endfor
```

is exported to:

```
proc (...)
{
  ...
  for (...)
  {
\#ifdef MY\_SWITCH
    int cnt = 100;
    // an optional block
\#endif
  }
  ...
}
```

An export definition can be activated and deactivated as any normal operator. Deactivated export definitions are not exported.

Parameters

▷ **Position** (input_control) . . . . . string ⇝ *string*
Place where the export text is written.
List of values: Position ∈ {'in_place', 'at_file_begin', 'before_procedure', 'after_procedure', 'at_file_end'}

▷ **Declaration** (input_control) . . . . . string ⇝ *string*
Text that is exported.

Result

`export_def` is never executed.

See also

`comment`

Module

Foundation

for (: : Start, End, Step : Index)

Starts a loop block that is usually executed for a fixed number of iterations.

Syntax in HDevelop: `for Index := Start to End by Step`

The `for` statement starts a loop block that is usually executed for a fixed number of iterations. The `for` block ends at the corresponding `endfor` statement.

The number of iterations is defined by the `Start` value, the `End` value, and the increment value `Step`. All of these parameters can be initialized with expressions or variables instead of constant values. Please note that these loop parameters are evaluated only once, namely, immediately before the `for` loop is entered. They are not re-evaluated after the loop cycles, i.e., any modifications of these variables within the loop body will have no influence on the number of iterations.

The passed loop parameters must be either of type `integer` or `real`. If all input parameters are of type `integer`, the `Index` variable will also be of type `integer`. In all other cases the `Index` variable will be of type `real`.

At the beginning of each iteration the loop variable `Index` is compared to the `End` parameter. If the increment value `Step` is positive, the `for` loop is executed as long as the `Index` variable is less than or equal to the `End` parameter. If the increment value `Step` is negative, the `for` loop is executed as long as the `Index` variable is greater than or equal to the `End` parameter.

Attention: If the increment value `Step` is set to a value of type `real`, it may happen that the last loop cycle is omitted owing to rounding errors in case the `Index` variable is expected to match the `End` value exactly in the last cycle. Hence, on some systems the following loop is not executed—as expected—for four times (with the `Index` variable set to 1.3, 1.4, 1.5, and 1.6), but only three times because after three additions the index variable is slightly greater than 1.6 due to rounding errors.

```
I:=[]
for Index := 1.3 to 1.6 by 0.1
  I := [I,Index]
endfor
```

After the execution of the loop body, i.e., upon reaching the corresponding `endfor` statement or a `continue` statement, the increment value (as initialized at the beginning of the `for` loop) is added to the current value of the loop counter `Index`. Then, the loop condition is re-evaluated as described above. Depending on the result the loop is either executed again or finished in which case execution continues with the first statement after the corresponding `endfor` statement.

A `break` statement within the loop—that is not covered by a more internal block—leaves the loop immediately and execution continues after the corresponding `endfor` statement. In contrast, the `continue` statement is used to ignore the rest of the loop body in the current cycle and continue execution with adapting the `Index` variable and re-evaluating the loop condition.

Attention: It is recommended to avoid modifying the `Index` variable of the `for` loop within its body.

If the `for` loop is stopped, e.g., by a `stop` statement or by pressing the `Stop` button, and if the PC is placed manually by the user, the `for` loop is continued at the current iteration as long as the PC remains within the `for` body or is set to the `endfor` statement. If the PC is set on the `for` statement (or before it) and executed again, the loop is reinitialized and restarts at the beginning.

Parameters

▷ **Start** (input_control) ..... number ⇝ *integer* / real
Start value of the loop variable.
Default: 1

▷ **End** (input_control) ..... number ⇝ *integer* / real
End value of the loop variable.
Default: 5

▷ **Step** (input_control) ..... number ⇝ *integer* / real
Increment value of the loop variable.
Default: 1

▷ **Index** (output_control) ..... number ⇝ *integer* / real
Loop variable.

Example

```
read_image (Image, 'fabrik')
threshold (Image, Region, 128, 255)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and', 150, 99999)
area_center (SelectedRegions, Area, Row, Column)
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
dev_display (Image)
dev_display (SelectedRegions)
dev_set_color ('white')
for Index := 0 to |Area| - 1 by 1
  set_tposition (WindowHandle, Row[Index], Column[Index])
  write_string (WindowHandle, 'Area=' + Area[Index])
endfor
```

Result

If the values of the specified parameters are correct, `for` (as an operator) returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code is returned.

Alternatives

`while`, `until`

See also

`repeat`, `break`, `continue`, `endfor`

Module

Foundation

global (: : Declaration :)

Declare a global variable.

The `global` statement can be used to declare a global variable. By declaring a variable as global the variable becomes visible to all other procedures that also declare the same variable explicitly as global.

If a variable is not explicitly declared as global inside a procedure, the variable is local within that procedure even if there is a global variable with the same name.

The parameter `Declaration` contains the variable declaration that consists of the optional keyword *'def'*, the type *'object'* or *'tuple'*, the optional keyword *'vector'* (followed by the desired dimension in round brackets), and the variable name.

Setting the type to *'object'* an iconic variable is declared, by setting it to *'tuple'* a control variable is declared.

The keyword *'def'* allows to mark one declaration explicitly as the place where the variable is defined. In most cases this will not be necessary because in HDevelop the variable instance is created as soon as it is declared somewhere. However, if several procedures are exported to a programming language and if the procedures are

not exported into one output file that contains all procedures together but into separate output files it will become necessary to mark one of the global variable declarations as the place where the variable is defined. A set of procedure export files that are linked to one library or application must contain exactly one definition of each global variable in order to avoid both undefined symbols and multiple definitions.

In the program listing, global variable declarations are displayed and must be entered without parenthesis in order to emphasize that the line is a declaration and not an executable operator. The syntax is as follows:

```
global [def] {object|tuple} [vector(<Dimension>)] <Variable Name>
```

Parameters

▷ **Declaration** (input_control) . . . . . string ⇝ *string*
Global variable declaration: optional keyword *'def'*, type, and variable name
Suggested values: Declaration ∈ {'object', 'tuple', 'def object', 'def tuple', 'object vector(1)', 'tuple vector(1)', 'def object vector(1)', 'def tuple vector(1)'}

Result

global is never executed.

Module

Foundation

if (: : Condition :)

Conditional statement.

if is a conditional statement that starts an if block. The Condition parameter must evaluate to a Boolean or integer expression.

If Condition evaluates to *'true'* (not *0*), the following block body up to the next corresponding block statement elseif, else, or endif is executed. Reaching the end of the block the execution continues after the corresponding endif statement.

If Condition evaluates to *'false'* (*0*), the execution is continued at the next corresponding block statement elseif, else, or endif.

Parameters

▷ **Condition** (input_control) . . . . . integer ⇝ *integer*
Condition for the if statement.
Default: 1

Result

If the condition is correct if (as operator) returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Alternatives

elseif, else

See also

for, while, until

Module

Foundation

import (: : ProcedureSource :)

Import one or more external procedures.

The import statement can be used to import additional external procedures from within a HDevelop program. The imported procedures become only available for the procedure that contains the import statement but not for other procedures.

import statements may occur in any line of a procedure. The imported procedures become only available below the import statement and may be overruled by later import statements.

```
proc()
* unresolved procedure call

import ./the\_one\_dir
proc()
* resolves to ./the\_one\_dir/proc.hdvp

import ./the\_other\_dir
proc()
* resolves to ./the\_other\_dir/proc.hdvp
```

The parameter `ProcedureSource` points to the source of the external procedures. It can either be the path of a directory that contains the procedures and/or the procedure libraries to be used or directly the file name of a procedure library. In both cases, the path may either be absolute or relative. In the latter case, HDevelop interprets the path as being relative to the file location of the procedure that contains the `import` statement. Thus, the location of this procedure can be included with '.'. The path has to be in quotes if it contains one or more spaces, otherwise the program line will become invalid.

Contrary to system, user-defined, and session directories HDevelop looks only in the directory specified by an `import` statement for external procedures but not recursively in its subdirectories.

Note, that an `import` statement is never executed and, therefore, `ProcedureSource` has to be evaluated already at the procedure's loading time. Therefore, `ProcedureSource` has to be a constant expression, and, in particular, it is not possible to pass a string variable to `ProcedureSource`.

However, `ProcedureSource` may also contain environment variables, which HDevelop resolves accordingly. Environment variables, regardless of the platform actually used, must always be denoted in Windows syntax, i.e., %VARIABLE%.

`import` neither tests whether the path `ProcedureSource` exists nor whether it points to a procedure library or a directory that contains procedures at all. Therefore, `import` statements with nonexistent or pointless paths nonetheless stay valid program lines, in any case.

Import paths are listed separately in HDevelop's procedure settings. Of course, these paths can't be modified or deactivated from within the procedure settings. Furthermore, procedures that are available only via an `import` statement are marked with a special icon.

In the program listing, import statements are displayed and must be entered without parenthesis in order to emphasize that the line is a declaration and not an executable operator.

Parameters

▷ **ProcedureSource** (input_control) ..... string ⇝ *string*
File location of the external procedures to be loaded: either a directory or a procedure library

Result

`import` is never executed.

Module

Foundation

insert (: : Input, Value, Index : Result)

Assignment of a value to a tuple element.

`insert` is obsolete and is only provided for reasons of backward compatibility. The modifying version of the `insert` operator was replaced by the operator `assign_at`. This operator uses the same notation in the full text editor, so that it is used automatically. The non-modifying version of the `insert` operator is replaced by the new operator `tuple_replace`.

`insert` assigns a single value to a specific element of a tuple.

In the full text editor an insert operation is simply entered with the help of the assignment operator sign ':=' and the index access operator sign '`[ ]`' for the result variable, e.g.:

```
Areas[Radius-1] := Area
```

If the operator window is used for entering the insert operator, `insert` must be entered into the operator combo box as the operator name. This opens the parameter area, where the parameter `Value` represents the expression that has to be evaluated to one value and assigned to the element at position `Index` within the tuple `Input`. The parameter `Result` gets the name of the variable where the result has to be stored.

If the input tuple that is passed via the parameter `Input` and the output tuple that is passed in `Result` are identical (and only in that case), the `insert` operator is listed and can be written in the full text editor in the above assignment notation. In this case, the input tuple is modified and the correct operator notation for above assignment would be:

```
insert (Areas, Area, Radius-1, Areas)
```

If the `Input` tuple and the `Result` tuple differ, the input tuple will not be modified. In this case, within the program listing only the operator notation can be used:

```
insert (Areas, Area, Radius-1, Result)
```

This is the same as:

```
Result := Areas
Result[Radius-1] := Area
```

Please note that the operator `insert` will not increase the tuple if the tuple already stores a value at the passed index. Instead of that the element at the position `Index` will be replaced. Hence, for the `Value` parameter exactly one single value (or an expression that evaluates to one single value) must be passed.

If the passed `Index` parameter is beyond the current tuple size, the tuple will be increased to the required size. The tuple elements that were inserted between the hitherto last element and the new element are undefined.

Parameters

▷ **`Input`** (input_control) . . . . . real(-array) ⇝ *real* / integer / string
Tuple, where the new value has to be inserted.
Default: []

▷ **`Value`** (input_control) . . . . . real ⇝ *real* / integer / string
Value that has to be inserted.
Default: 1
Value range: $0 \leq$ `Value` ≤ 1000000

▷ **`Index`** (input_control) . . . . . integer ⇝ *integer*
Index position for new value.
Default: 0
Suggested values: `Index` $\in \{0, 1, 2, 3, 4, 5, 6\}$
Minimum increment: 1

▷ **`Result`** (output_control) . . . . . real(-array) ⇝ *real* / integer / string
Result tuple with inserted values.

Result

If the expression is correct `insert` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Alternatives

`assign`

Module

Foundation

`par_join` (: : ThreadID :)

Wait for subthreads that were started with the `par_start` qualifier.

The `par_join` operator is used to wait in the calling procedure for all procedures or operators that have been started in separate subthreads by adding the `par_start` qualifier to the according program line. The subthreads to wait for are identified by their thread ids that are passed to the parameter `ThreadID`.

Attention: par_start is not an operator but a qualifier that is added at the begin of the program line that has to be executed in parallel to the calling procedure. The syntax is par_start <ThreadID> : followed by the actual procedure or operator call.

Parameters

▷ **ThreadID** (input_control) ..... thread_id(-array) ⇝ *integer*
Ids of all subthreads to wait for.

Example

```
* start two procedures in separate sub threads
par_start <ThreadID1> : producer_proc()
par_start <ThreadID2> : consumer_proc()
* wait until both procedures have finished
par_join ([ThreadID1, ThreadID2])
```

Result

If the values of the specified parameters are correct, par_join returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Module

Foundation

repeat (: : :)

Start statement of a repeat..until loop.

repeat is the first statement of an repeat..until loop.

Result

repeat always returns 2 (H_MSG_TRUE).

Alternatives

for, while

See also

until, break, continue

Module

Foundation

return (: : :)

Terminate procedure call.

return terminates the current procedure call and returns to the calling procedure. Program execution is continued at the next active program line after the procedure call in the calling procedure. If the current procedure is the main procedure, program execution is finished and the program counter jumps to the end of the program. Note that every procedure except the main procedure has to contain at least one reachable return operator line in order to be able to return from a call to the procedure.

Result

return always returns 2 (H_MSG_TRUE).

Module

Foundation

stop (: : :)

Stop program execution.

The stop operator stops the continuous program execution of the HDevelop program. If this happens, the PC remains on the stop statement (instead of being placed at the next executable program line) to show the reason for the program interruption directly even if numerous comments or other non-executable program lines follow.

The operator is equivalent to the Stop action (F9) in the menu bar. Unless parallel execution is used (via the par_start qualifier), the program can easily be continued with the Run action (F5). See also "Parallel Execution" in the HDevelop User's Guide.

It is possible to redefine the behavior by setting a time parameter in the preferences dialog. In this case, the execution will not stop but continue after waiting for the specified period of time. Within this period of time, the program can be interrupted with F9 or continued with one of the run commands. This is marked by an icon in the first column of the program window.

Attention

This operator is not supported for code export.

Trying to continue a program that uses parallel execution after calling stop may cause non-deterministic thread behavior or errors.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
count_obj (Regions, Number)
dev_update_window ('off')
for i := 1 to Number by 1
  select_obj (Regions, RegionSelected, i)
  dev_clear_window ()
  dev_display (RegionSelected)
  stop ()
endfor
```

Result

If the program stops at a stop statement, the return state of the previous operator is kept. If the program is continued with the stop operator, stop always returns 2 (H_MSG_TRUE).

See also

exit

Module

Foundation

switch (: : ControlExpression :)

Starts a multiway branch block.

switch starts a block that allows to control the program flow via a multiway branch. The parameter ControlExpression must result in an integer value. This value determines to what case label the execution jumps. Every case statement includes one integer constant. If the integer constant of a case statement is equal to the calculated value of the parameter ControlExpression, the program execution continues there. In addition, an optional default statement can be defined as the last jump label within a switch block. The program execution jumps to this default label, if no case constant matches the calculated ControlExpression value.

As in the programming languages C, C++, and C#, the case statement is a jump label and—in contrast to elseif—not the begin of an enclosed block that is automatically left at the next case or default statement.

In order to leave the switch block after the execution of the code lines of a case branch, as in C or C++ a break statement must be inserted at the end of the case branch. break statements can be used anywhere within a switch block. This causes the program execution to continue after the closing endswitch statement. Without a break statement at the end of a branch the program execution "falls through" to the statements of the following case or default branch.

If the same statements have to be executed in different cases, i.e., for multiple control values, several case statements with different constant expressions can be listed one below the other.

Parameters

▷ **ControlExpression** (input_control) ..... integer ⇝ *integer*
Integer expression that determines at which case label the program execution is continued.

Example

```
TestStr := ''
for Index := 1 to 8 by 1
  TestStr := TestStr + '<'
  switch (Index)
  case 1:
    TestStr := TestStr + '1'
    break
  case 2:
    TestStr := TestStr + '2'
    * intentionally fall through to 3
  case 3:
    TestStr := TestStr + '3'
    * intentionally fall through to 4
  case 4:
    TestStr := TestStr + '4'
    break
  case 5:
  case 6:
    * common case branch for 5 and 5
    TestStr := TestStr + '56'
    break
  case 7:
    * continue for loop
    TestStr := TestStr + '7'
    continue
  default:
    TestStr := TestStr + 'd'
    break
  endswitch
  TestStr := TestStr + '>'
endfor
```

Result

If the condition is correct, `switch` (as an operator) returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code is returned.

Alternatives

`if`, `elseif`, `else`

See also

`case`, `default`, `endswitch`, `if`

Module

Foundation

throw (: : Exception :)

Throws a user-defined exception or rethrows a caught exception.

With the help of the operators `try`, `catch`, `endtry`, and `throw` it is possible to implement a dynamic exception handling in HDevelop, which is comparable to the exception handling in C++ and C#. The basic concepts of the exception handling in HDevelop are also described at the operators `try`, and `dev_set_check` as well as in the "HDevelop User's Guide".

The operator throw provides an opportunity to throw an exception from an arbitrary place in the program. This exception can be caught by the catch operator of a surrounding try-catch block. By this means the developer is able to define his own specific error or exception states, for which the normal program execution is aborted in order to continue with a specific cross-procedure exception handling, e.g., for freeing resources or restarting from a defined state.

In such a user-defined exception a nearly arbitrary tuple can be thrown as the Exception parameter, merely the first element of the tuple should be set to a user-defined error code $\geq$ 30000. If different user-defined exception states are possible, they can be distinguished using different error codes ($\geq$ 30000) in the first element or by using additional elements.

In addition, with the help of the operator throw it is possible to rethrow an exception that was caught with the operator catch. This may be sensible, for instance, if within an inner try-catch-endtry block (e.g., within an external procedure) only specific exceptions can be handled in an adequate way and all other exceptions must be passed to the caller, where they can be caught and handled by an outer try-catch-endtry block.

For rethrowing a caught exception, it is possible to pass the Exception tuple that was caught by the catch operator directly to the Exception parameter of the throw operator. Furthermore, it is possible to append arbitrary (but no iconic) user data to the Exception tuple, that can be accessed after catching the exception as *'user_data'* with the operator dev_get_exception_data:

```
try
    ...
catch(Exception)
    ...
    UserData := ...
    throw([Exception, UserData])
endtry
```

Attention

The export of the operators try, catch, endtry, and throw is not supported for the language C, but only for the languages C++, C# and VisualBasic/.NET. Only the latter support throwing exceptions across procedures.

Parameters

▷ **Exception** (input_control) ..... exception-array ⇝ *integer* / string
Tuple returning the exception data or user defined error codes.

Result

If the values of the specified parameters are correct, throw (as operator) returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

See also

try, catch, endtry, dev_get_exception_data, dev_set_check

Module

Foundation

try (: : :)

Starts a program block where exceptions are detected and caught.

With the help of the operators try, catch, endtry, and throw it is possible to implement a dynamic exception handling in HDevelop, which is comparable to the exception handling in C++ and C#. By the operators try, catch, and endtry two code blocks are formed: the first one (try .. catch) contains the watched program lines that perform the normal program logic. The second block (catch .. endtry) contains the code that has is executed if an exception occurs.

The operator try enables the exception handling for the following program lines, i.e., the following code block up to the corresponding catch operator is watched for exceptions. If during the execution of the subsequent program lines an error or another exceptional state occurs, or if an exception is thrown explicitly by the operator throw, the try block is left immediately (or—depending on a user preference—after displaying an error message box) and the program execution continues at the corresponding catch operator. If the exception is thrown within

a procedure that was called from the try block (directly or via other procedure calls), the procedure call and all intermediate procedure calls that are on the call stack above the try block are immediately aborted (or, if applicable, also after displaying an error message box).

Whether an error message box is displayed before the exception is thrown or not, is controlled by the HDevelop preference 'Suppress error message dialogs within try-catch blocks' that can be reached via Edit->Preferences->General Options->Experienced User. This message box also offers the opportunity to stop the program execution before the exception is thrown in order to edit the possibly erroneous operator call.

The program block that is watched for exceptions ends with the corresponding catch operator. If within the watched try block no exception occurred, the following catch block is ignored and the program execution continues after the corresponding endtry operator.

try-catch-endtry blocks can be nested arbitrarily into each other, within a procedure or over different procedure calls, as long as any inner try-catch-endtry block lies completely *either* within an outer try-catch *or* a catch-endtry block. If an exception is thrown within an inner try-catch block, the exception handling is caught in the corresponding catch-endtry block. Hence, the exception is not visible for the outer try-catch blocks unless the exception is rethrown explicitly by calling a throw operator from the catch block.

If within a HALCON operator an error occurs, an exception tuple is created and passed to the catch operator that is responsible for catching the exception. The tuple collects information about the error such as the error code and the error text. After catching an exception, this information can be accessed with the help of the operator dev_get_exception_data. For more information about the passed exception data, how to access them, and considerations about the code export, see the description of that operator. The reference of the operator throw describes how to throw user-defined exception tuples.

HDevelop offers the opportunity to disable the handling of HALCON errors. This can be achieved by calling the operator dev_set_check('~give_error') or by unchecking the check box *Give Error* on the dialog Edit->Preferences->Runtime Settings. If the error handling is switched off, in case of an HALCON error no exception is thrown but the program execution is continued as normal at the next operator. In contrast to that, the operator throw will always throw an exception independently of the *'give_error'* setting. The same applies if an error occurred during the evaluation of an parameter expression.

Attention

The export of the operators try, catch, endtry, and throw is not supported for the language C, but only for the languages C++, C# and VisualBasic/.NET. Only the latter support throwing exceptions across procedures.

Example

```
try
  read_image (Image, 'may_be_not_available')
catch (Exception)
  if (Exception[0] == 5200)
    dev_get_exception_data (Exception, 'error_message', ErrMsg)
    set_tposition (3600, 24, 12)
    write_string (3600, ErrMsg)
    return ()
  else
    * rethrow the exception
    throw ([Exception,'unknown exception in myproc'])
  endif
endtry
```

Result

try always returns 2 (H_MSG_TRUE).

Alternatives

dev_set_check

See also

catch, endtry, throw, dev_get_exception_data, dev_set_check

Module

Foundation

until (: : Condition :)

Continue to execute the body as long as the condition is not true.

until ends a repeat..until loop. The repeat..until loop is executed as long as the Condition parameter evaluates to *'false'* (*0*). The body of the loop is executed at least once, because the condition will be checked at the end of the body.

Parameters

▷ **Condition** (input_control) ..... integer ⇝ *integer*
Condition for loop.

Result

If the values of the specified parameters are correct, until (as operator) returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Alternatives

for, while

See also

repeat, if, elseif, else, break, continue

Module

Foundation

while (: : Condition :)

Starts a loop block that is executed as long as the condition is true.

while executes the loop body up to the corresponding endwhile statement as long as the Condition parameter evaluates to *'true'* (or a number not equal *0*).

If the condition evaluates to *'false'* (*0*) the program is continued after the corresponding endwhile statement.

Parameters

▷ **Condition** (input_control) ..... integer ⇝ *integer*
Condition for loop.

Example

```
dev_update_window ('off')
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowID)
read_image (Image, 'particle')
dev_display (Image)
stop ()
threshold (Image, Large, 110, 255)
dilation_circle (Large, LargeDilation, 7.5)
dev_display (Image)
dev_set_draw ('margin')
dev_set_line_width (3)
dev_set_color ('green')
dev_display (LargeDilation)
dev_set_draw ('fill')
stop ()
complement (LargeDilation, NotLarge)
reduce_domain (Image, NotLarge, ParticlesRed)
mean_image (ParticlesRed, Mean, 31, 31)
dyn_threshold (ParticlesRed, Mean, SmallRaw, 3, 'light')
opening_circle (SmallRaw, Small, 2.5)
connection (Small, SmallConnection)
dev_display (Image)
```

```
dev_set_colored (12)
dev_display (SmallConnection)
stop ()
dev_set_color ('green')
dev_display (Image)
dev_display (SmallConnection)
Button := 1
while (Button == 1)
  dev_set_color ('green')
  get_mbutton (WindowID, Row, Column, Button)
  dev_display (Image)
  dev_display (SmallConnection)
  dev_set_color ('red')
  select_region_point (SmallConnection, SmallSingle, Row, Column)
  dev_display (SmallSingle)
  NumSingle := |SmallSingle|
  if (NumSingle == 1)
    intensity (SmallSingle, Image, MeanGray, DeviationGray)
    area_center (SmallSingle, Area, Row, Column)
    dev_set_color ('yellow')
    set_tposition (WindowID, Row, Column)
    write_string (WindowID, 'Area='+Area+', Int='+MeanGray)
  endif
endwhile
dev_set_line_width (1)
dev_update_window ('on')
```

Result

If the values of the specified parameters are correct, `while` (as operator) returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Alternatives

`for`, `until`

See also

`repeat`, `break`, `continue`, `if`, `elseif`, `else`

Module

Foundation

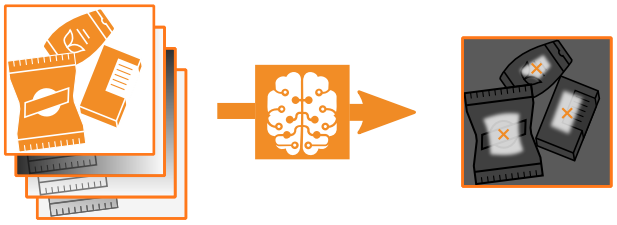
Chapter 9

Deep Learning

Introduction

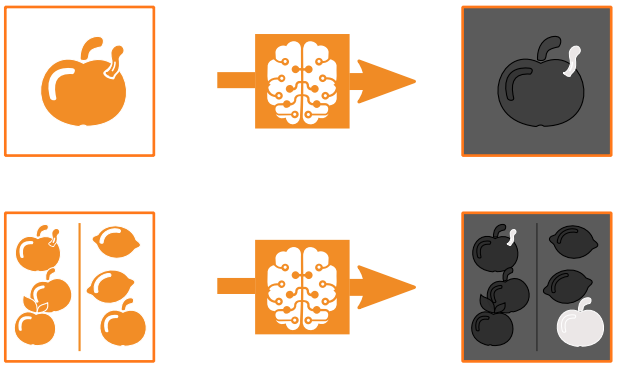
The term deep learning (DL) refers to a family of machine learning methods. In HALCON, the following methods are implemented:

3D Gripping Point Detection: Detect gripping points on objects in a 3D scene. For further information please see the chapter 3D Matching / 3D Gripping Point Detection.



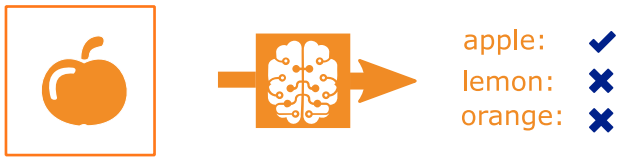
A possible example for a 3D Gripping Point Detection application: A 3D scene (e.g., an RGB image and XYZ-images) is analyzed and possible gripping points are suggested.

Anomaly Detection and Global Context Anomaly Detection Assign to each pixel the likelihood that it shows an unknown feature. For further information please see the chapter Deep Learning / Anomaly Detection and Global Context Anomaly Detection.

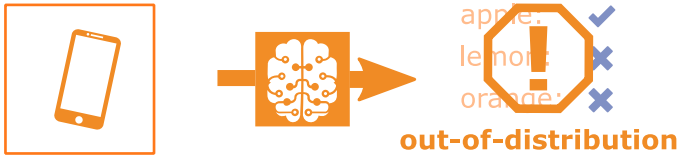


Top: A possible example for anomaly detection: A score is assigned to every pixel of the input image, indicating how likely it shows an unknown feature, i.e., an anomaly. Bottom: A possible example for Global Context Anomaly Detection: A score is assigned to every pixel of the input image, indicating how likely it shows a structural or logical anomaly.

Classification: Classify an image into one class out of a given set of classes. For further information please see the chapter Deep Learning / Classification.

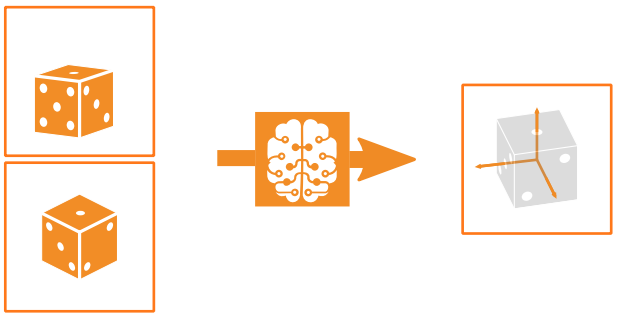


A possible example for classification: The image gets assigned to a class.



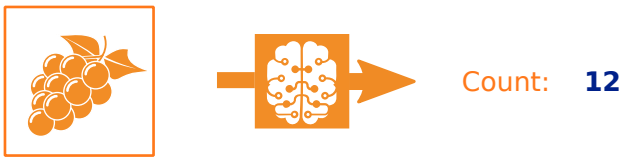
A possible example for Out-of-Distribution Detection for classification: The image is assigned to a class and identified as Out-of-Distribution if applicable.

Deep 3D Matching: Detect objects in a scene and compute their 3D pose. For further information please see the chapter 3D Matching / Deep 3D Matching.



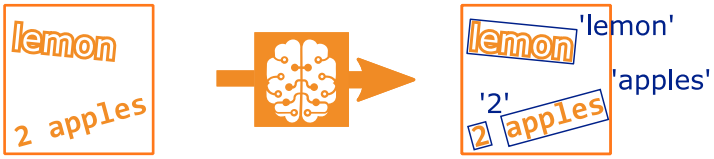
A possible example for a Deep 3D Matching application: Images from different angles are used to detect an object. As a result the 3D pose of the object is computed.

Deep Counting: Detect and count objects in images. For further information please see the chapter Matching / Deep Counting.



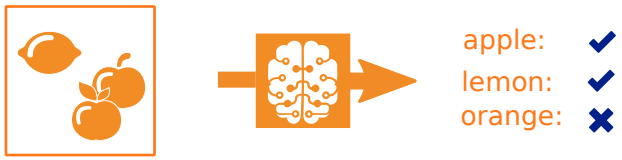
A possible example for a Deep Counting application: Objects in an image are counted and the object quantity is returned.

Deep OCR: Detect and recognize words (not just characters) in an image. For further information please see the chapter OCR / Deep OCR.



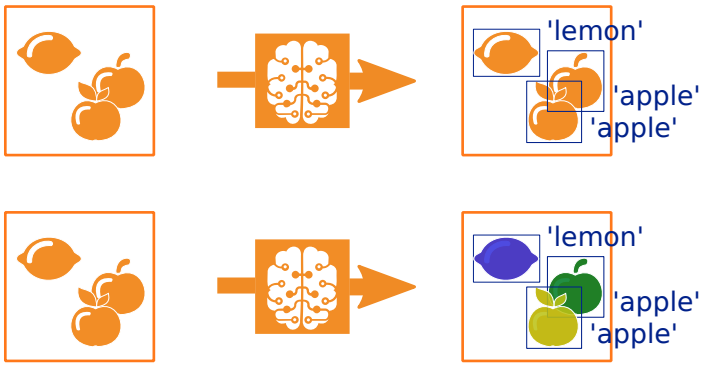
A possible example for deep-learning-based optical character recognition: Words in an image are detected and recognized.

Multi-label Classification: An image is assigned all contained classes from a given set of classes. For further information please see the chapter Deep Learning / Multi-Label Classification.



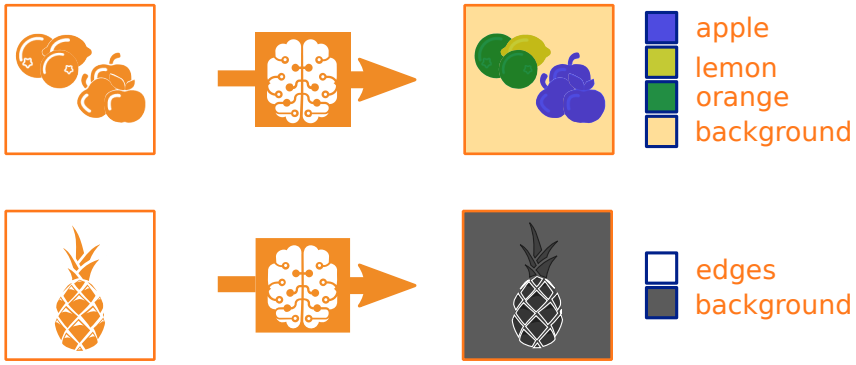
A possible example for multi-label classification: All contained classes are assigned to the image.

Object Detection and Instance Segmentation: Detect objects of the given classes and localize them within the image. Instance segmentation is a special case of object detection, where the model also predicts distinguished object instances and additionally assigns for the found instances their region within the image. For further information please see the chapter Deep Learning / Object Detection and Instance Segmentation.



Top: A possible example for object detection: Within the input image three instances are found and assigned to a class.
Bottom: A possible example for instance segmentation: Every instance gets its individual region marked.

Semantic Segmentation and Edge Extraction: Assign a class to each pixel of an image, but different instances of a class are not distinguished. A special case of semantic segmentation, where every pixel of the input image is assigned to one of the two classes 'edge' and 'background'. For further information please see the chapter Deep Learning / Semantic Segmentation and Edge Extraction.



Top: A possible example for semantic segmentation: Every pixel of the input image is assigned to a class.
Bottom: A possible example for edge extraction: Pixels belonging to specific edges are assigned to the class 'edge'.

All of the deep learning methods listed above use a network for the assignment task. In HALCON they are implemented within the general DL model, see Deep Learning / Model. The model is trained by only considering the input and output, which is also called end-to-end learning. Basically, using images and the information, what is visible in them, the training algorithm adjusts the model in a way to distinguish the different classes and eventually also how to find the corresponding objects. For you, it has the nice outcome of no need for manual feature specification. Instead you have to select and collect appropriate data.

System Requirements and License Information

For Deep learning additional prerequisites apply. Please see the requirements listed in the HALCON "`Installation Guide`", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods".

Note that the required module license depends on the model type used in your application. For a detailed description please refer to the "Installation Guide", paragraph "Dynamic Modules for Deep-Learning-Based Applications".

General Workflow

As the DL methods mentioned above differ in what they do and how they need the data, you need to know which method is most appropriate for your specific task. Once this is clear, you need to collect a suitable amount of data, meaning images and the information needed by the method. After that, there is a common general workflow for all these DL methods:

Prepare the Network and the Data The network needs to be prepared for your task and your data adapted to the specific network.

- Get a network: Read in a pretrained network or create one.
- The network needs to know which problem it shall solve, i.e., which classes are to be distinguished and what such samples look like. This is represented by your dataset, i.e., your images with the corresponding ground truth information.
- The network will impose several requirements on the images (as e.g., the image dimension, gray value range, ...). Therefore the images have to be preprocessed so that the network can process them.
- We recommend to split the dataset into three distinct datasets which are used for training, validation, and testing.

Train the Network and Evaluate the Training Progress Once your network is set up and your data prepared it is time to train the network for your specific task.

- Set the hyperparameters appropriate to your task and system.
- Optionally specify your data augmentation.
- Start the training and evaluate your network.

Apply and Evaluate the Final Network Your network is trained for your task and ready to be applied. But before deploying it in the real world you should evaluate how well the network performs on basis of your test dataset.

Inference Phase When your network is trained and you are satisfied with its performance, you can use it for inference on new images. Thereby the images need to be preprocessed according to the requirements of the network (thus, in the same way as for training).

Data

The term 'data' is used in the context of deep learning as the images and the information, what is in them. This last information has to be provided in a way the network can understand. Not surprisingly, the different DL methods have their own requirements concerning what information has to be provided and how. Please see the corresponding chapters for the specific requirements.

The network further poses requirements on the images regarding the image dimensions, the gray value range, and the type. The specific values depend on the network itself and can be queried with `get_dl_model_param`. Additionally, depending on the method there are also requirements regarding the information as e.g., the bounding boxes. To fulfill all these requirements, the data may have to be preprocessed, which can be done most conveniently with the corresponding procedure `preprocess_dl_samples`.

When you train your network, the network gets adapted to its task. But at one point you will want to evaluate what the network learned and at an even later point you will want to test the network. Therefore the dataset will be split into three subsets which should be independent and identically distributed. In simple words, the subsets should not be connected to each other in any way and each set contains for every class the same distribution of images. This splitting is conveniently done by the procedure `split_dl_dataset`. The clearly largest subset will be used for the retraining. We refer to this dataset as the training dataset. At a certain point the performance of the network is evaluated to check whether it is beneficial to continue the network optimization. For this validation the second set of data is used, the validation dataset. Even if the validation dataset is disjoint from the first one, it has an influence on the network optimization. Therefore to test the possible predictions when the model is deployed in

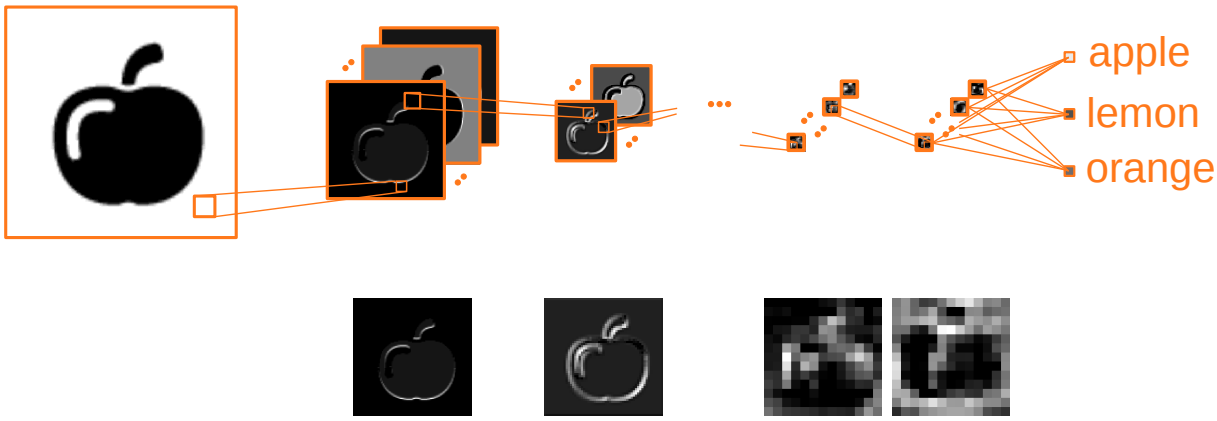
the real world, the third dataset is used, the test dataset. For a representative network validation or evaluation, the validation and test dataset should have statistically relevant data, which gives a lower bound on the amount of data needed.

Note also, that for training the network, you best use representative images, i.e., images like the ones you want to process later and not only 'perfect' images, as otherwise the network may have difficulties with non-'perfect' images.

The Network and the Training Process

In the context of deep learning, the assignments are performed by sending the input image through a network. The output of the total network consists of a number of predictions. Such predictions are e.g., for a classification task the confidence for each class, expressing how likely the image shows an instance of this class.

The specific network will vary, especially from one method to another. Some methods like e.g., object detection, use a subnetwork to generate feature maps (see the explanations given below and in Deep Learning / Object Detection and Instance Segmentation). Here, we will explain a basic Convolutional Neural Network (CNN). Such a network consists of a certain number of layers or filters, which are arranged and connected in a specific way. In general, any layer is a building block performing specific tasks. It can be seen as a container, which receives input, transforms it according to a function, and returns the output to the next layer. Thereby different functions are possible for different types of layers. Several possible examples are given in the "`Solution Guide on Classification`". Many layers or filters have weights, parameters which are also called filter weights. These are the parameters modified during the training of a network. The output of most layers are feature maps. Thereby the number of feature maps (the depth of the layer output) and their size (width and height) depends on the specific layer.



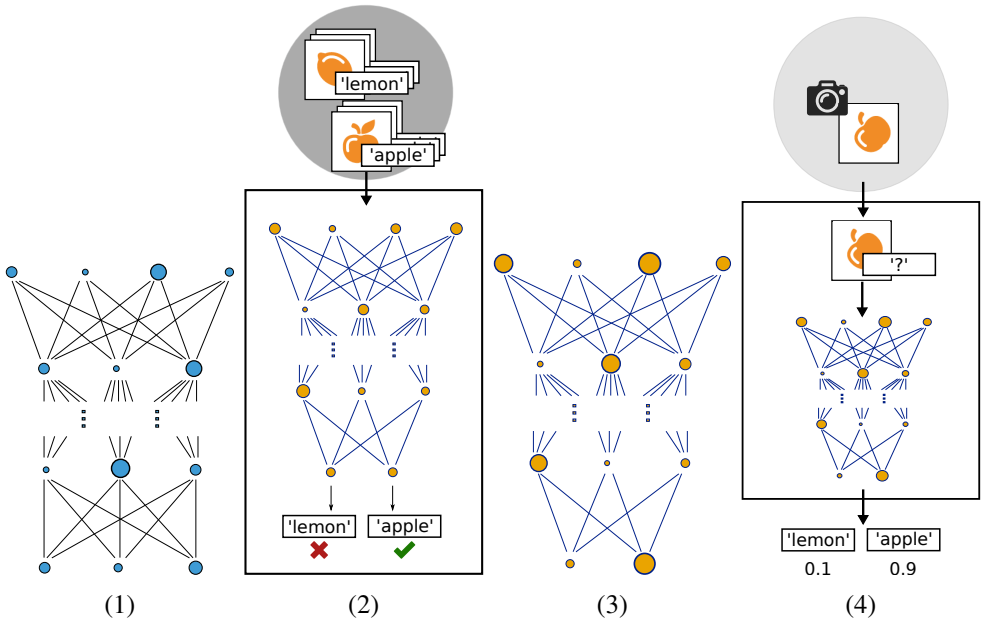
Schema of an extract of a possible classification network. Below we show feature maps corresponding to the layers, zoomed to a uniform size.

To train a network for a specific task, a loss function is added. There are different loss functions depending on the task, but they all work according to the following principle. A loss function compares the prediction from the network with the given information, what it should find in the image (and, if applicable, also where), and penalizes deviations. Now the filter weights are updated in such a way that the loss function is minimized. Thus, training the network for the specific tasks, one strives to minimize the loss (an error function) of the network, in the hope of doing so will also improve the performance measure. In practice, this optimization is done by calculating the gradient and updating the parameters of the different layers (filter weights) accordingly. This is repeated by iterating multiple times over the training data.

There are additional parameters that influence the training, but which are not directly learned during the regular training. These parameters have values set before starting the training. We refer to this last type of parameters as hyperparameters in order to distinguish them from the network parameters that are optimized during training. See the section "Setting the Training Parameters: The Hyperparameters".

To train all filter weights from scratch a lot of resources are needed. Therefore one can take advantage from the following observation. The first layers detect low level features like edges and curves. The feature map of the following layers are smaller, but they represent more complex features. For a large network, the low level features are general enough so the weights of the corresponding layers will not change much among different tasks. This leads to a technique called transfer learning: One takes an already trained network and retrains it for a specific task,

benefiting from already quite suitable filter weights for the lower layers. As a result, considerably less resources are needed. While in general the network should be more reliable when trained on a larger dataset, the amount of data needed for retraining also depends on the complexity of the task. A basic schema for the workflow of transfer learning is shown with the aid of classification in the figure below.



Basic schema of transfer learning with the aid of classification. (1) A pretrained network is read. (2) Training phase, the network gets trained with the training data. (3) The trained model with new capabilities. (4) Inference phase, the trained network infers on new images.

Setting the Training Parameters: The Hyperparameters

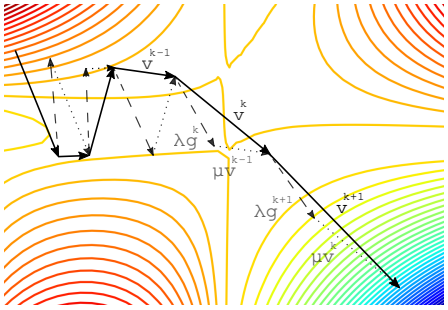
The different DL methods are designed for different tasks and will vary in the way they are built up. They all have in common that during the training of the model one faces a minimization problem. Training the network or subnetwork, one strives to minimize an appropriate loss function, see the section "The Network and the Training Process". For doing so, there is a set of further parameters which is set before starting the training and not optimized during the training. We refer to these parameters as hyperparameters. For a DL model, you can set a change strategy, specifying when and how you want these hyperparameters changed during the training. In this section, we explain the idea of the different hyperparameters. Note, that certain methods have additional hyperparameters, you find more information in their respective chapter.

As already mentioned, the loss compares the predictions from the network with the given information about the content of the image. The loss now penalizes deviations. Training the network means updating the filter weights in such a way, that the loss has to penalize less, thus the loss result is optimized. To do so, a certain amount of data is taken from the training dataset. For this subset the gradient of the loss is calculated and the network modified in updating its filter weights accordingly. Now this is repeated with the next subset of data till the whole training data is processed. These subsets of the training data are called batches and the size of these subsets, the *'batch_size'*, determines the number of data taken into a batch and as a consequence processed together.

A full iteration over the entire training data is called epoch. It is beneficial to iterate several times over the training data. The number of iterations is defined by `'epochs'`. Thus, `'epochs'` determines how many times the algorithm loops over the training set.

Some models (e.g., anomaly detection) train utilizing the whole dataset at once. For other models, the dataset is processed batch-wise and in order to do so, the SGD (stochastic gradient descent algorithm) or Adam (adaptive moment estimation) can be used. This involves further parameters, which are explained in the following. After every calculation of the loss gradient the filter weights are updated. For this update, there are two important hyperparameters: The *'learning_rate'* λ, determining the weight of the gradient on the updated loss function arguments (the filter weights), and the *'momentum'* μ within the interval $[0, 1)$, specifying the influence of previous updates. More information can be found in the documentation of `train_dl_model_batch`. In simple words, when we update the loss function arguments, we still remember the step we took for the last update. Now, we take a step in direction of the gradient with a length depending to the learning rate; additionally we repeat the step

we did last times, but this time only μ times as long. A visualization is given in the figure below. A too large learning rate might result in divergence of the algorithm, a very small learning rate will take unnecessarily many steps. Therefore, it is customary to start with a larger learning rate and potentially reduce it during training. With a momentum $\mu = 0$, the momentum method has no influence, so only the gradient determines the update vector.



Sketch of the *'learning_rate'* and the *'momentum'* during an actualization step. The gradient step: the learning rate λ times the gradient g (λg - dashed lines). The momentum step: the momentum μ times the previous update vector v (μv - dotted lines). Together, they form the actual step: the update vector v (v - solid lines).

To prevent the neural networks from overfitting (see the part "Risk of Underfitting and Overfitting" below), regularization can be used. With this technique an extra term is added to the loss function. One possible type of regularization is weight decay, for details see the documentation of `train_dl_model_batch`. It works by penalizing large weights, i.e., pushing the weights towards zero. Simply put, this regularization favors simpler models that are less likely to fit to noise in the training data and generalize better. It can be set by the hyperparameter *'weight_prior'*. Choosing its value is a trade-off between the model's ability to generalize, overfitting, and underfitting. If *'weight_prior'* is too small the model might overfit, if it is too large, the model might loose its ability to fit the data well because all weights are effectively zero.

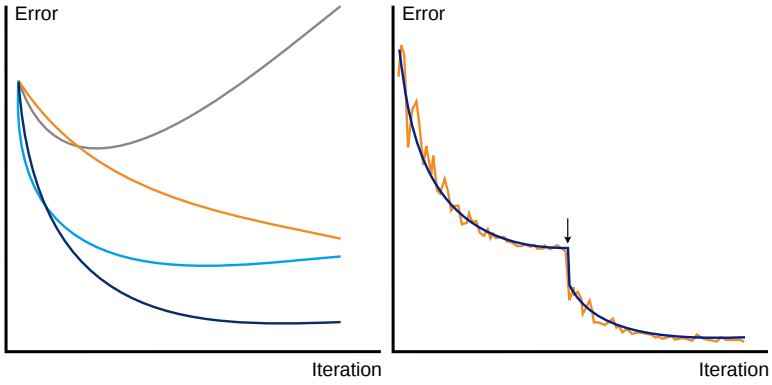
With the training data and all the hyperparameters, there are many different aspects that can have an influence on the outcome of such complex algorithms. To improve the performance of a network, generally the addition of training data also helps. Please note, whether to gather more data is a good solution always depends also on how easily one can do so. Usually, a small additional fraction will not noticeably change the total performance.

Supervising the training

The different DL methods have different results. Accordingly they also use different measures to determine 'how well' a network performs. When training a network, there are behaviors and pitfalls applying to different models, which are described here.

Validation During Training When it comes to the validation of the network performance, it is important to note that this is not a pure optimization problem (see the parts "The Network and the Training Process" and "Setting the Training Parameters" above).

In order to observe the training progress, it is usually helpful to visualize a validation measure, e.g., for the training of a classification network, the error over the samples of a batch. As the samples differ, the difficulty of the assignment task may differ. Thus it may be that the network performs better or worse for the samples of a given batch than for the samples of another batch. So it is normal that the validation measure is not changing smoothly over the iterations. But in total it should improve. Adjusting the hyperparameters *'learning_rate'* and *'momentum'* can help to improve the validation measure again. The following figures show possible scenarios.

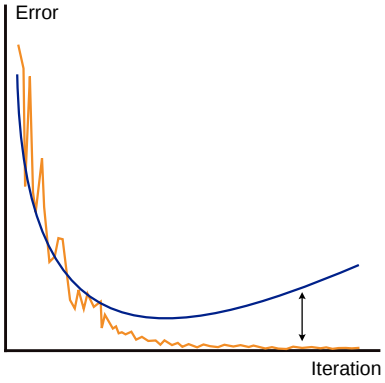


(1) (2)

Sketch of an validation measure during training, here using the error from classification as example. (1) General tendencies for possible outcomes with different *'learning_rate'* values, dark blue: good learning rate, gray: very high learning rate, light blue: high learning rate, orange: low learning rate. (2) Ideal case with a learning rate policy to reduce the *'learning_rate'* value after a given number of iterations. In orange: training error, dark blue: validation error. The arrow marks the iteration, at which the learning rate is decreased.

Risk of Underfitting and Overfitting Underfitting occurs if the model is not able to capture the complexity of the task. It is directly reflected in the validation measure on the training set which stays high.

Overfitting happens when the network starts to 'memorize' training data instead of learning how to generalize. This is shown by a validation measure on the training set which stays good or even improves while the validation measure on the validation set decreases. In such a case, regularization may help. See the explanations of the hyperparameter *'weight_prior'* in the section "Setting the Training Parameters: The Hyperparameters". Note that a similar phenomenon occurs when the model capacity is too high with respect to the data.



Sketch of a possible overfitting scenario, visible on the generalization gap (indicated with the arrow). The error from classification serves as an example for a validation measure.

Confusion Matrix A network infers for an instance a top prediction, the class for which the network deduces the highest affinity. When we know its ground truth class, we can compare the two class affiliations: the predicted one and the correct one. Thereby, the instance differs between the different types of methods, while e.g., in classification the instances are images, in semantic segmentation the instances are single pixels.

When more than two classes are distinguished, one can also reduce the comparison into binary problems. This means, for a given class you just compare if it is the same class (positive) or any other class (negative). For such binary classification problems the comparison is reduced to the following four possible entities (whereof not all are applicable for every method):

- True positives (TP: predicted positive, labeled positive),
- true negatives (TN: predicted negative, labeled negative),
- false positives (FP: predicted positive, labeled negative),
- false negatives (FN: predicted negative, labeled positive).

A confusion matrix is a table with such comparisons. This table makes it easy to see how well the network performs for each class. For every class it lists how many instances have been predicted into which class. E.g., for a classifier distinguishing the three classes 'apple', 'peach', and 'orange', the confusion matrix shows how many images with ground truth class affiliation 'apple' have been classified as 'apple' and how many have been classified as 'peach' or 'orange'. Of course, this is listed for the other classes as well. This example is shown in the figure below. In HALCON, we represent for each class the instances with this ground truth label in a column and the instances predicted to belong to this class in a row.

(1) (2)

An example for a confusion matrices from classification. We see that 68 images of an 'apple' have been classified as such (TP), 60 images showing not an 'apple' have been correctly classified as a 'peach' (30) or 'pear' (30) (TN), 0 images show a 'peach' or a 'pear' but have been classified as an 'apple' (FP) and 24 images of an 'apple' have wrongly been classified as 'peach' (21) or 'pear' (3) (FN). (1) A confusion matrix for all three distinguished classes. It appears as if the network 'confuses' apples and peaches more than all other combinations. (2) The confusion matrix of the binary problem to better visualize the 'apple' class.

Glossary

In the following, we describe the most important terms used in the context of deep learning:

Adam Adam (adaptive moment estimation) is a first-order gradient-based optimization algorithm for stochastic objective functions, which computes individual adaptive learning rates. In the deep learning methods this algorithm can be used to minimize the loss function.

anchor Anchors are fixed bounding boxes. They serve as reference boxes, with the aid of which the network proposes bounding boxes for the objects to be localized.

annotation An annotation is the ground truth information, what a given instance in the data represents, in a way recognizable for the network. This is e.g., the bounding box and the corresponding label for an instance in object detection.

anomaly An anomaly means something deviating from the norm, something unknown.

backbone A backbone is a part of a pretrained classification network. Its task is to generate various feature maps, for what reason the classifying layer has been removed.

batch size - hyperparameter *'batch_size'* The dataset is divided into smaller subsets of data, which are called batches. The batch size determines the number of images taken into a batch and thus processed simultaneously.

bounding box Bounding boxes are rectangular boxes used to define a part within an image and to specify the localization of an object within an image.

class agnostic Class agnostic means without the knowledge of the different classes.

In HALCON, we use it for reduction of overlapping predicted bounding boxes. This means, for a class agnostic bounding box suppression the suppression of overlapping instances is done ignoring the knowledge of classes, thus strongly overlapping instances get suppressed independently of their class.

change strategy A change strategy denotes the strategy, when and how hyperparameters are changed during the training of a DL model.

class Classes are discrete categories (e.g., 'apple', 'peach', 'pear') that the network distinguishes. In HALCON, the class of an instance is given by its appropriate annotation.

classifier In the context of deep learning we refer to the term classifier as follows. The classifier takes an image as input and returns the inferred confidence values, expressing how likely the image belongs to every distinguished class. E.g., the three classes 'apple', 'peach', and 'pear' are distinguished. Now we give an image of an apple to the classifier. As a result, the confidences 'apple': 0.92, 'peach': 0.07, and 'pear': 0.01 could be returned.

COCO COCO is an abbreviation for "common objects in context", a large-scale object detection, segmentation, and captioning dataset. There is a common file format for each of the different annotation types.

confidence Confidence is a number expressing the affinity of an instance to a class. In HALCON the confidence is the probability, given in the range of [0,1]. Alternative name: score

confusion matrix A confusion matrix is a table which compares the classes predicted by the network (top-1) with the ground truth class affiliations. It is often used to visualize the performance of the network on a validation or test set.

Convolutional Neural Networks (CNNs) Convolutional Neural Networks are neural networks used in deep learning, characterized by the presence of at least one convolutional layer in the network. They are particularly successful for image classification.

data We use the term data in the context of deep learning for instances to be recognized (e.g., images) and their appropriate information concerning the predictable characteristics (e.g., the labels in case of classification).

data augmentation Data augmentation is the generation of altered copies of samples within a dataset. This is done in order to augment the richness of the dataset, e.g., through flipping or rotating.

dataset: training, validation, and test set With dataset we refer to the complete set of data used for a training. The dataset is split into three, if possible disjoint, subsets:

- The training set contains the data on which the algorithm optimizes the network directly.
- The validation set contains the data to evaluate the network performance during training.
- The test set is used to test possible inferences (predictions), thus to test the performance on data without any influence on the network optimization.

deep learning The term "deep learning" was originally used to describe the training of neural networks with multiple hidden layers. Today it is rather used as a generic term for several different concepts in machine learning. In HALCON, we use the term deep learning for methods using a neural network with multiple hidden layers.

epoch In the context of deep learning, an epoch is a single training iteration over the entire training data, i.e., over all batches. Iterations over epochs should not be confused with the iterations over single batches (e.g., within an epoch).

errors In the context of deep learning, we refer to error when the inferred class of an instance does not match the real class (e.g., the ground truth label in case of classification). Within HALCON, we use the term error in deep learning when we refer to the top-1 error.

feature map A feature map is the output of a given layer

feature pyramid A feature pyramid is simply a group of feature maps, whereby every feature map origins from another level, i.e., it is smaller than its preceding levels

head Heads are subnetworks. For certain architectures they attach on selected pyramid levels. These subnetworks proceed information from previous parts of the total network in order to generate spatially resolved output, e.g., for the class predictions. Thereof they generate the output of the total network and therewith constitute the input of the losses.

hyperparameter Like every machine learning model, CNNs contain many formulas with many parameters. During training the model learns from the data in the sense of optimizing the parameters. However, such models can have other, additional parameters, which are not directly learned during the regular training. These

parameters have values set before starting the training. We refer to this last type of parameters as hyperparameters in order to distinguish them from the network parameters that are optimized during training. Or from another point of view, hyperparameters are solver-specific parameters.

Prominent examples are the initial learning rate or the batch size.

inference phase The inference phase is the stage when a trained network is applied to predict (infer) instances (which can be the total input image or just a part of it) and eventually their localization. Unlike during the training phase, the network is not changed anymore in the inference phase.

in-distribution In-distribution refers to data that comes from the same underlying distribution as the data on which a model was trained. When a model encounters in-distribution data during inference , the data is similar in terms of its statistical properties, features, and patterns to what the model has seen before during training.

intersection over union The intersection over union (IoU) is a measure to quantify the overlap of two areas. We can determine the parts common in both areas, the intersection, as well as the united areas, the union. The IoU is the ratio between the two areas intersection and union.

The application of this concept may differ between the methods.

label Labels are arbitrary strings used to define the class of an image. In HALCON these labels are given by the image name (eventually followed by a combination of underscore and digits) or by the directory name, e.g., 'apple_01.png', 'pear.png', 'peach/01.png'.

layer and hidden layer A layer is a building block in a neural network, thus performing specific tasks (e.g., convolution, pooling, etc., for further details we refer to the "`Solution Guide on Classification`"). It can be seen as a container, which receives weighted input, transforms it, and returns the output to the next layer. Input and output layers are connected to the dataset, i.e., the images or the labels, respectively. All layers in between are called hidden layers.

learning rate - hyperparameter *'learning_rate'* The learning rate is the weighting, with which the gradient is considered when updating the arguments of the loss function. In simple words, when we want to optimize a function, the gradient tells us the direction in which we shall optimize and the learning rate determines how far along this direction we step.

Alternative names: λ, step size

level The term level is used to denote within a feature pyramid network the whole group of layers, whose feature maps have the same width and height. Thereby the input image represents level 0.

loss A loss function compares the prediction from the network with the given information, what it should find in the image (and, if applicable, also where), and penalizes deviations. This loss function is the function we optimize during the training process to adapt the network to a specific task.

Alternative names: objective function, cost function, utility function

momentum - hyperparameter *'momentum'* The momentum $\mu \in [0, 1)$ is used for the optimization of the loss function arguments. When the loss function arguments are updated (after having calculated the gradient), a fraction μ of the previous update vector (of the past iteration step) is added. This has the effect of damping oscillations. We refer to the hyperparameter μ as momentum. When μ is set to 0, the momentum method has no influence. In simple words, when we update the loss function arguments, we still remember the step we did for the last update. Now we go a step in direction of the gradient with a length according to the learning rate and additionally we repeat the step we did last time, but this time only μ times as long.

non-maximum suppression In object detection, non-maximum suppression is used to suppress overlapping predicted bounding boxes. When different instances overlap more than a given threshold value, only the one with the highest confidence value is kept while the other instances, not having the maximum confidence value, are suppressed.

Out-of-Distribution Out-of-Distribution refers to data that significantly differs from the data on which a model was trained. When a model encounters out-of-distribution data during inference, the data's statistical properties, features, or patterns are unfamiliar to the model, leading to potential challenges in making accurate predictions.

overfitting Overfitting happens when the network starts to 'memorize' training data instead of learning how to find general rules for the classification. This becomes visible when the model continues to minimize error on the training set but the error on the validation set increases. Since most neural networks have a huge amount of weights, these networks are particularly prone to overfitting.

regularization - hyperparameter *'weight_prior'* Regularization is a technique to prevent neural networks from overfitting by adding an extra term to the loss function. It works by penalizing large weights, i.e., pushing the weights towards zero. Simply put, regularization favors simpler models that are less likely to fit to noise in the training data and generalize better. In HALCON, regularization is controlled via the parameter *'weight_prior'*.

Alternative names: regularization parameter, weight decay parameter, λ (note that in HALCON we use λ for the learning rate and within formulas the symbol α for the regularization parameter).

retraining We define retraining as updating the weights of an already pretrained network, i.e., during retraining the network learns the specific task.

Alternative names: fine-tuning.

solver The solver optimizes the network by updating the weights in a way to optimize (i.e., minimize) the loss.

stochastic gradient descent (SGD) SGD is an iterative optimization algorithm for differentiable functions. A key feature of the SGD is to calculate the gradient only based on a single batch containing stochastically sampled data and not all data. In the deep learning methods this algorithm can be used to calculate the gradient to optimize (i.e., minimize) the loss function.

top-k error The classifier infers for a given image class confidences of how likely the image belongs to every distinguished class. Thus, for an image we can sort the predicted classes according to the confidence value the classifier assigned. The top-k error tells the ratio of predictions where the ground truth class is not within the k predicted classes with highest probability. In the case of top-1 error, we check if the target label matches the prediction with the highest probability. In the case of top-3 error, we check if the target label matches one of the top 3 predictions (the 3 labels getting the highest probability for this image).

Alternative names: top-k score

transfer learning Transfer learning refers to the technique where a network is built upon the knowledge of an already existing network. In concrete terms this means taking an already (pre)trained network with its weights and adapt the output layer to the respective application to get your network. In HALCON, we also see the following retraining step as a part of transfer learning.

underfitting Underfitting occurs when the model over-generalizes. In other words it is not able to describe the complexity of the task. This is directly reflected in the error on the training set, which does not decrease significantly.

weights In general weights are the free parameters of the network, which are altered during the training due to the optimization of the loss. A layer with weights multiplies or adds them with its input values. In contrast to hyperparameters, weights are optimized and thus changed during the training.

Further Information

Get an introduction to deep learning or learn about datasets for deep learning and many other topics in interactive online courses at our MVTec Academy .

```
get_dl_device_param ( : : DLDeviceHandle,
    GenParamName : GenParamValue )
```

Return the parameters of a deep-learning-capable hardware device.

`get_dl_device_param` returns the parameter values `GenParamValue` of `GenParamName` for the deep-learning-capable hardware device (hereafter referred to as device) `DLDeviceHandle`. See `query_available_dl_devices` for details about deep-learning-capable hardware devices.

Supported values for `GenParamName` are:

'calibration_precisions': Specifies the unit data types that can be used for a calibration of a deep learning model.
List of values: *'int8'*.

'cast_precisions': Specifies the unit data types that can be used for a cast of a deep learning model.

When changing the data type the calibration does not require any images.

List of values: *'float32'*, *'float16'*.

'conversion_supported': Returns *'true'* if unit data types for either a calibration or a cast of a deep learning model are available. Returns *'false'* in any other case.

'id': The ID of the device. Within each inference engine, the IDs of its supported devices are unique. The same holds for devices supported through HALCON.

'inference_only': Indicates if the device can only be used to infer deep learning models (*'true'*) or also supports training or gradient-based operations (*'false'*).

'ai_accelerator_interface': AI Accelerator Interface (AI^2) on which this unit `DLDeviceHandle` is executed. In case the device is directly supported by HALCON, the value *'none'* is returned.

List of values: *'tensorrt'*, *'openvino'*, *'none'*.

'info': Dictionary containing additional information on the device.

Restriction: Only for devices that are supported via an AI2-interface.

'name': Name of the device.

'optimize_for_inference_params': Dictionary with default-defined conversion parameters for a calibration or cast operation of a deep learning model. The entries can be changed.

In case no parameter applies to the set device, an empty dictionary is returned.

Restriction: Only for devices that are supported via an AI2-interface.

'precisions': Specifies the data types that the unit supports for the weights and/or activations of a deep-learning-based model.

List of values: *'float32'*, *'float16'*, *'int8'*.

'settable_device_params': Dictionary with settable device parameters.

Restriction: Only for devices that are supported via an AI2-interface.

'type': Type of the device.

Parameters

▷ **`DLDeviceHandle`** (input_control) . . . . . dl_device ⇝ *handle*
Handle of the deep-learning-capable hardware device.

▷ **`GenParamName`** (input_control) . . . . . attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'type'
List of values: `GenParamName` ∈ {'calibration_precisions', 'cast_precisions', 'conversion_supported', 'id', 'ai_accelerator_interface', 'inference_only', 'info', 'name', 'optimize_for_inference_params', 'precisions', 'settable_device_params', 'type'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.name(-array) ⇝ *string* / real / integer
Value of the generic parameter.

Result

If the parameters are valid, the operator `get_dl_device_param` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`query_available_dl_devices`

Possible Successors

`set_dl_model_param`

Module

Foundation

```
optimize_dl_model_for_inference ( : : DLModelHandle,
    DLDeviceHandle, Precision, DLSamples,
    GenParam : DLModelHandleConverted, ConversionReport )
```

Optimize a model for inference on a device via the AI²-interface.

The operator `optimize_dl_model_for_inference` optimizes the input model `DLModelHandle` for inference on the device `DLDeviceHandle` and returns the optimized model in `DLModelHandleConverted`. This operator has two distinct functionalities: Casting the model precision to `Precision` and calibrating the model based on the given samples `DLSamples`. Additionally in either case the model architecture may be optimized for the `DLDeviceHandle`.

The parameter `DLDeviceHandle` specifies the deep learning device for which the model is optimized. Whether the device supports optimization can be determined using `get_dl_device_param` with *'conversion_supported'*. After a successful execution, `optimize_dl_model_for_inference` sets the parameter *'precision_is_converted'* to *'true'* for the output model `DLModelHandleConverted`. In addition, the device in `DLDeviceHandle` is automatically set for the model if it supports the precision set by the parameter `Precision`. Whether the device supports the requested precision can be determined using `get_dl_device_param` with *'precisions'*.

The parameter `Precision` specifies the precision to which the model should be converted to. By default, models that are delivered by HALCON have the `Precision` *'float32'*. The following values are supported for `Precision`:

- *'float32'*
- *'float16'*
- *'int8'*

The parameter `DLSamples` specifies the samples on which the calibration is based. As a consequence they should be representative. It is recommended to provide them from the training split. For most applications 10-20 samples per class are sufficient to achieve good results.

Note, the samples are not needed for a pure cast operation. In this case, an empty tuple can be passed over for `DLSamples`.

The parameter `GenParam` specifies additional, device specific parameters and their values. Which parameters to set for the given `DLDeviceHandle` in `GenParam` and their default values can be queried via the `get_dl_device_param` operator with the *'optimize_for_inference_params'* parameter.

Note, certain devices also expect only an empty dictionary.

The parameter `ConversionReport` returns a report dictionary with information about the conversion.

Attention

This operator can only be used via an AI²-interface. Furthermore, after optimization only parameters that do not change the underlying architecture of the model can be set for `DLModelHandleConverted`.

For `set_dl_model_param`, this includes the following parameters:

- 'Any': *'device'*, *'meta_data'*, *'runtime'*
- 'anomaly_detection': *'standard_deviation_factor'*
- 'classification': *'class_names'*, *'ood_threshold'*
- 'ocr_detection': *'min_character_score'*, *'min_link_score'*, *'min_word_score'*, *'orientation'*, *'sort_by_line'*, *'tiling'*, *'tiling_overlap'*
- 'ocr_recognition': *'alphabet'*, *'alphabet_internal'*, *'alphabet_mapping'*
- 'gc_anomaly_detection': *'anomaly_score_tolerance'*
- 'detection': *'class_names'*, *'max_num_detections'*, *'max_overlap'*, *'max_overlap_class_agnostic'*, *'min_confidence'*
- 'segmentation': *'class_names'*

For `set_deep_ocr_param`, this includes the following parameters:

- *'device'*, *'runtime'*

- *'detection_min_character_score'*, *'detection_min_link_score'*, *'detection_min_word_score'*,
- *'detection_orientation'*, *'detection_sort_by_line'*,
- *'detection_tiling'*, *'detection_tiling_overlap'*
- *'recognition_alphabet'*, *'recognition_alphabet_internal'*, *'recognition_alphabet_mapping'*

For `set_deep_counting_model_param`, this includes the following parameters:

- *'device'*
- *'max_overlap'*, *'min_score'*

Only the AI^2-interface that was used to optimize can be set using *'device'* or the *'runtime'*. Additional restrictions may apply to these parameters to ensure that the underlying architecture of the model does not change.

Parameters

▷ **DLModelHandle** (input_control) . . . . . dl_model ⇝ *handle*
Input model.

▷ **DLDeviceHandle** (input_control) . . . . . dl_device(-array) ⇝ *handle*
Device handle used for optimization.

▷ **Precision** (input_control) . . . . . string ⇝ *string*
Precision the model shall be converted to.

▷ **DLSamples** (input_control) . . . . . dict-array ⇝ *handle*
Samples required for optimization.

▷ **GenParam** (input_control) . . . . . dict ⇝ *handle*
Parameter dict for optimization.

▷ **DLModelHandleConverted** (output_control) . . . . . dl_model ⇝ *handle*
Output model with new precision.

▷ **ConversionReport** (output_control) . . . . . dict ⇝ *handle*
Output report for conversion.

Result

If the parameters are valid, the operator `optimize_dl_model_for_inference` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`train_dl_model_batch`, `query_available_dl_devices`

Possible Successors

`set_dl_model_param`, `apply_dl_model`

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Matching, Deep Learning Enhanced, Deep Learning Professional

query_available_dl_devices (: : GenParamName, GenParamValue : DLDeviceHandles)

Get list of deep-learning-capable hardware devices.

`query_available_dl_devices` returns a list of handles. Each handle refers to a deep-learning-capable hardware device (hereafter referred to as device) that can be used for inference or training of a deep learning model. For each returned device, every parameter mentioned in `GenParamName` must be equal to at least one

of its corresponding values that appear in GenParamValue. A parameter can have more than one value by duplicating its name in GenParamName and adding different corresponding value in GenParamValue.

A deep-learning-capable device is either supported directly through HALCON or through an AI^2-interface.

The devices that are supported directly through HALCON are equivalent to those that can be set to a deep learning model via set_dl_model_param using *'runtime'* = *'cpu'* or *'runtime'* = *'gpu'*. HALCON provides an internal implementation for the inference or training of a deep learning model for those devices. See Deep Learning for more details.

Devices that are supported through the AI^2-interface can also be set to a deep learning model using set_dl_model_param. In this case the inference is not executed by HALCON but by the device itself.

query_available_dl_devices returns a handle for each deep-learning-capable device supported through HALCON and through an inference engine.

If a device is supported through HALCON and one or several inference engines, query_available_dl_devices returns a handle for HALCON and for each inference engine.

GenParamName can be used to filter for the devices. All GenParamName that are gettable by get_dl_device_param and that do not return a handle-typed value for GenParamValue are supported for filtering. See the operator reference of get_dl_device_param for the list of gettable parameters. In addition, the following values are supported:

'runtime': The devices, which are directly supported by HALCON for this device type.

List of values: *'cpu'*, *'gpu'*.

GenParamName can have multiple entries for the same value. In this case filter combines the entries with a logical *'or'*. Please see the example code below for some examples how to use the filter.

Parameters

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: GenParamName ∈ {'calibration_precisions', 'cast_precisions', 'conversion_supported', 'id', 'ai_accelerator_interface', 'inference_only', 'name', 'optimize_for_inference_params', 'precisions', 'runtime', 'settable_device_params', 'type'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Value of the generic parameter.
Default: []
Suggested values: GenParamValue ∈ {}

▷ **DLDeviceHandles** (output_control) . . . . . dl_device(-array) ⇝ *handle*
Tuple of DLDevice handles

Example

```
* Query all deep-learning-capable hardware devices
query_available_dl_devices ([], [], DLDeviceHandles)

* Query all GPUs with ID 0 or 2
query_available_dl_devices (['type', 'id', 'id'], ['gpu', 0, 2],\
                            DLDeviceHandles)

* Query the unique GPU with ID 1 supported by HALCON
query_available_dl_devices (['runtime', 'id'], ['gpu', 1], DLDeviceHandles)
```

Result

If the parameters are valid, the operator query_available_dl_devices returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

get_dl_device_param

Module

Foundation

set_dl_device_param (: : DLDeviceHandle, GenParamName, GenParamValue :)

Set the parameters of a deep-learning-capable hardware device.

set_dl_device_param sets the parameter values GenParamValue for GenParamName for the deep-learning-capable hardware device (hereafter referred to as device) DLDeviceHandle. See query_available_dl_devices for details about deep-learning-capable hardware devices.

Parameters

▷ **DLDeviceHandle** (input_control) . . . . . dl_device ⇝ *handle*
Handle of the deep-learning-capable hardware device.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: GenParamName ∈ {'settable_device_params'}

▷ **GenParamValue** (input_control) . . . . . attribute.name(-array) ⇝ *string* / real / integer / handle
Value of the generic parameter.

Result

If the parameters are valid, the operator set_dl_device_param returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_available_dl_devices

Possible Successors

set_dl_model_param

See also

get_dl_device_param

Module

Foundation

9.1 Anomaly Detection and Global Context Anomaly Detection

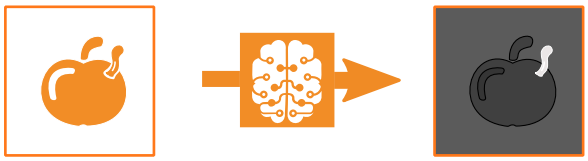
This chapter explains how to use anomaly detection and Global Context Anomaly Detection based on deep learning.

With those two methods we want to detect whether or not an image contains anomalies. An anomaly means something deviating from the norm, something unknown.

An anomaly detection or Global Context Anomaly Detection model learns common features of images without anomalies. The trained model will infer, how likely an input image contains only learned features or if the image contains something different. Latter one is interpreted as an anomaly. This inference result is returned as a gray value image. The pixel values therein indicate how likely the corresponding pixels in the input image pixels show an anomaly.

We differentiate between two model types that can be used:

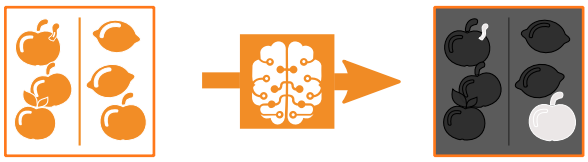
Anomaly Detection With anomaly detection (model type *'anomaly_detection'*) structural anomalies are targeted, thus any feature that was not learned during training. This can, e.g., include scratches, cracks or contamination.



A possible example for anomaly detection: Every pixel of the input image gets assigned a value that indicates how likely the pixel is to be an anomaly. The worm is not part of the worm-free apples the model has seen during training and therefore its pixels get a much higher score.

Global Context Anomaly Detection Global Context Anomaly Detection (model type *'gc_anomaly_detection'*) comprises two tasks:

- **Detecting structural anomalies**
  As described for anomaly detection above, structural anomalies primarily include unknown features, like scratches, cracks or contamination.
- **Detecting logical anomalies**
  Logical anomalies are detected if constraints regarding the image content are violated. This can, e.g., include a wrong number or wrong position of objects in an image.

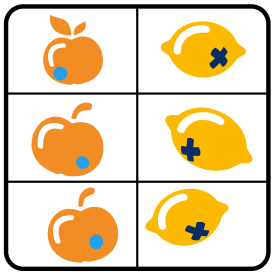


A possible example for Global Context Anomaly Detection: Every pixel of the input image gets assigned a value that indicates how likely the pixel is to be an anomaly. Thereby two different types of anomalies can be detected, structural and logical ones. Structural anomaly: One apple contains a worm, which differs from the apples the model has seen during training. Logical anomaly: One apple is sorted among lemons. Although the apple itself is intact, the logical constraint is violated, as the model has only seen images with correctly sorted fruit during training.

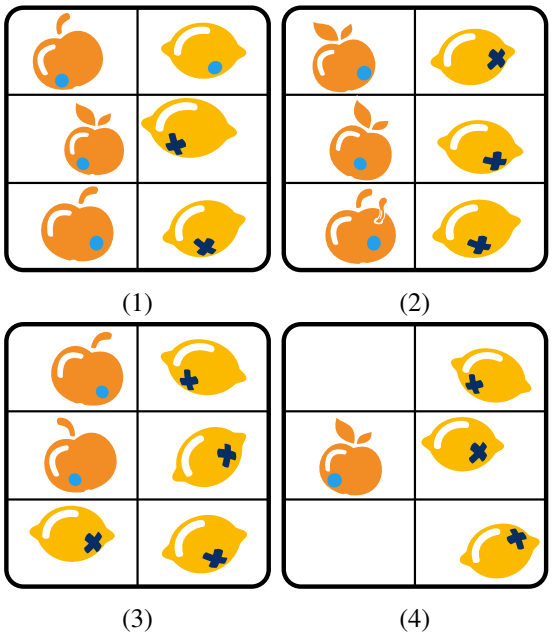
The Global Context Anomaly Detection model consists of two subnetworks. The model can be reduced to one of the subnetworks, in order to improve the runtime and memory consumption. This is recommended if a single subnetwork performs well enough. See the parameter *'gc_anomaly_networks'* in `get_dl_model_param` for details. After setting *'gc_anomaly_networks'*, the model needs to be evaluated again, since this parameter can change the Global Context Anomaly Detection performance significantly.

- **Local subnetwork**
  This subnetwork is used to detect anomalies that affect the image on a smaller, local scale. It is designed to detect structural anomalies but can find logical anomalies as well. Thus, if an anomaly can be recognized by analyzing single patches of an image, it is detected by the local component of the model. See the description of the parameter *'patch_size'* in `get_dl_model_param` for information on how to define the local scale of this subnetwork.

- **Global subnetwork**
  This subnetwork is used to detect anomalies that affect the image on a large, or global scale. It is designed to detect logical anomalies but can find structural anomalies as well. Thus, if you need to see most or all of the image to recognize an anomaly, it is detected by the global component of the model.



Training image of an exemplary task. Apples and lemons are intact, sorted correctly, and tagged with the correct sticker.



Some anomalies that can be detected with Global Context Anomaly Detection: (1) Logical anomaly, most likely detected by the local subnetwork (wrong sticker). (2) Structural anomaly, most likely detected by local subnetwork (wormy apple). (3) Logical anomaly, most likely detected by global subnetwork (wrong sorting). (4) Logical anomaly, most likely detected by global subnetwork (missing apples).

General Workflow

In this paragraph, we describe the general workflow for an anomaly detection or Global Context Anomaly Detection task based on deep learning.

Preprocess the data This part is about how to preprocess your data.

1. The information content of your dataset needs to be converted. This is done by the procedure
   - `read_dl_dataset_anomaly`.

   It creates a dictionary `DLDataset` which serves as a database and stores all necessary information about your data. For more information about the data and the way it is transferred, see the section "Data" below and the chapter Deep Learning / Model.
2. Split the dataset represented by the dictionary `DLDataset`. This can be done using the procedure
   - `split_dl_dataset`.
3. The network imposes several requirements on the images. These requirements (for example the image size and gray value range) can be retrieved with

- get_dl_model_param.

For this you need to read the model first by using

- read_dl_model.

4. Now you can preprocess your dataset. For this, you can use the procedure

   - preprocess_dl_dataset.

   In case of custom preprocessing, this procedure offers guidance on the implementation.
   To use this procedure, specify the preprocessing parameters as, e.g., the image size. Store all the parameter with their values in a dictionary DLPreprocessParam, for which you can use the procedure

   - create_dl_preprocess_param.

   We recommend to save this dictionary DLPreprocessParam in order to have access to the preprocessing parameter values later during the inference phase.

Training of the model This part explains how to train a model.

1. Set the training parameters and store them in the dictionary TrainParam. This can be done using the procedure

   - create_dl_train_param.

2. Train the model. This can be done using the procedure

   - train_dl_model.

   The procedure

   - adapts models of type *'gc_anomaly_detection'* to the image statistics of the dataset calling the procedure normalize_dl_gc_anomaly_features,
   - calls the corresponding training operator train_dl_model_anomaly_dataset (*'anomaly_detection'*) or train_dl_model_batch (*'gc_anomaly_detection'*), respectively.

   The procedure expects:

   - the model handle DLModelHandle
   - the dictionary DLDataset containing the data information
   - the dictionary TrainParam containing the training parameters

3. Normalize the network. This step is only necessary when using a Global Context Anomaly Detection model. The anomaly scores need to be normalized by applying the procedure

   - normalize_dl_gc_anomaly_scores.

   This needs to be done in order to get reasonable results when applying a threshold on the anomaly scores later (see section "Specific Parameters" below).

Evaluation of the trained model In this part, we evaluate the trained model.

1. Set the model parameters which may influence the evaluation.
2. The evaluation can be done conveniently using the procedure

   - evaluate_dl_model.

   This procedure expects a dictionary GenParam with the evaluation parameters.
3. The dictionary EvaluationResult holds the desired evaluation measures.

Inference on new images This part covers the application of an anomaly detection or Global Context Anomaly Detection model. For a trained model, perform the following steps:

1. Request the requirements the model imposes on the images using the operator

   - get_dl_model_param

   or the procedure

   - create_dl_preprocess_param_from_model.

2. Set the model parameter described in the section “Model Parameters” below, using the operator
   - `set_dl_model_param`.
3. Generate a data dictionary `DLSample` for each image. This can be done using the procedure
   - `gen_dl_samples_from_images`.
4. Every image has to be preprocessed the same way as for the training. For this, you can use the procedure
   - `preprocess_dl_samples`.

   When you saved the dictionary `DLPreprocessParam` during the preprocessing step, you can directly use it as input to specify all parameter values.
5. Apply the model using the operator
   - `apply_dl_model`.
6. Retrieve the results from the dictionary `DLResult`.

Data

We distinguish between data used for training, evaluation, and inference on new images.

As a basic concept, the model handles data by dictionaries, meaning it receives the input data from a dictionary `DLSample` and returns a dictionary `DLResult` and `DLTrainResult`, respectively. More information on the data handling can be found in the chapter Deep Learning / Model.

Classes In anomaly detection and Global Context Anomaly Detection there are exactly two classes:

- *'ok'*, meaning without anomaly, class ID *0*.
- *'nok'*, meaning with anomaly, class ID *1* (on pixel values with an ID *>0*, see the subsection “Data for evaluation” below).

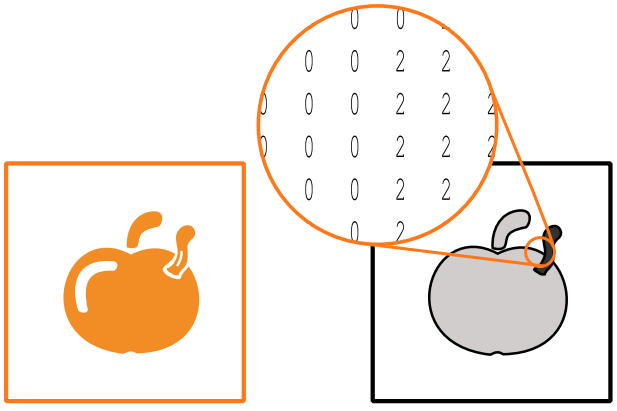
These classes apply to the whole image as well as single pixels.

Data for training This dataset consists only of images without anomalies and the corresponding information. They have to be provided in a way the model can process them. Concerning the image requirements, find more information in the section “Images” below.

The training data is used to train a model for your specific task. With the aid of this data the model can learn which features the images without anomalies have in common.

Data for evaluation This dataset should include images without anomalies but it can also contain images with anomalies. Every image within this set needs a ground truth label `image_label` specifying the class of the image (see the section above). This indicates if the image shows an anomaly (*'nok'*) or not (*'ok'*).

Evaluating the model performance on finding anomalies can visually also be done on pixel level if an image `anomaly_file_name` is included in the `DLSample` dictionary. In this image `anomaly_file_name` every pixel indicates the class ID, thus if the corresponding pixel in the input image shows an anomaly (pixel value > *0*) or not (pixel value equal to *0*).



Scheme of anomaly_file_name. For visibility, gray values are used to represent numbers. (1) Input image. (2) The corresponding anomaly_file_name providing the class annotations, *0*: *'ok'* (white and light gray), *2*: *'nok'* (dark gray).

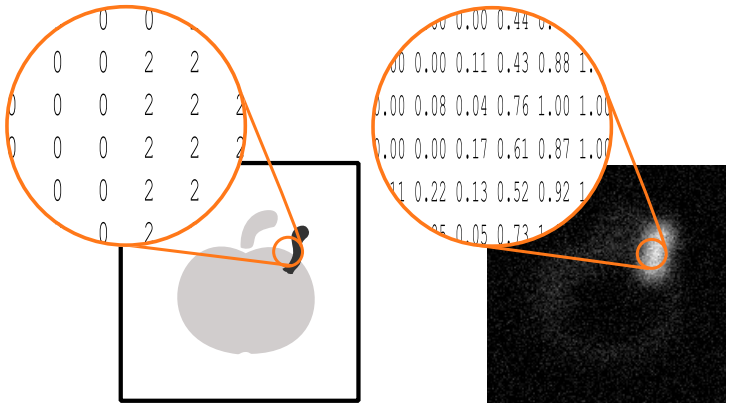
Images The model poses requirements on the images, such as the dimensions, the gray value range, and the type. The specific values depend on the model itself. See the documentation of read_dl_model for the specific values of different models. For a read model they can be queried with get_dl_model_param. In order to fulfill these requirements, you may have to preprocess your images. Standard preprocessing of an entire sample, including the image, is implemented in preprocess_dl_samples. In case of custom preprocessing these procedure offers guidance on the implementation.

Model output The training output differs depending on the used model type:

- Anomaly detection: As training output, the operator train_dl_model_anomaly_dataset will return a dictionary DLTrainResult with the best obtained error received during training and the epoch in which this error was achieved.
- Global Context Anomaly Detection: As training output, the operator train_dl_model_batch will return a dictionary DLTrainResult with the current value of the total loss as well as values for all other losses included in your model.

As inference and evaluation output, the model will return a dictionary DLResult for every sample. For anomaly detection and Global Context Anomaly Detection, this dictionary includes the following extra entries:

- anomaly_score: A score indicating how likely the entire image is to contain an anomaly. This score is based on the pixel scores given in anomaly_image.
  For Global Context Anomaly Detection, depending on the used subnetworks, the anomaly score can also be calculated by the local (anomaly_score_local) and the global (anomaly_score_global) subnetwork only. The anomaly_score is by default equal to the maximum of anomaly_image. The parameter anomaly_score_tolerance can be used to ignore a fraction of outliers in the anomaly_image when calculating the anomaly_score.
- anomaly_image: An image, where the value of each pixel indicates how likely its corresponding pixel in the input image shows an anomaly (see the illustration below). For anomaly detection the values are $\in [0, 1]$, whereas there are no constraints for Global Context Anomaly Detection. Depending on the used subnetworks, when using Global Context Anomaly Detection, an anomaly image can also be calculated by the local (anomaly_image_local) or the global (anomaly_image_global) subnetwork only.



(1) (2)

Scheme of anomaly_image. For visualization purpose, gray values are used to represent numbers. (1) The anomaly_file_name providing the class annotations, *0*: *'ok'* (white and light gray), *2*: *'nok'* (dark gray) (2) The corresponding anomaly_image.

Specific Parameters

For an anomaly detection or Global Context Anomaly Detection model, the model parameters as well as the hyperparameters are set using set_dl_model_param. The model parameters are explained in more detail in

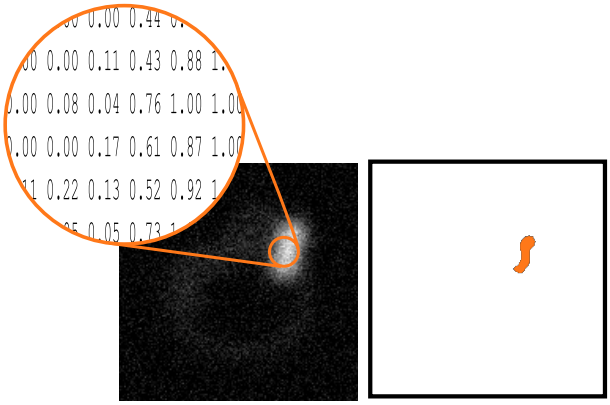
get_dl_model_param. As the training for an anomaly detection model is done utilizing the full dataset at once and not batch-wise, certain parameters as e.g., *'batch_size_multiplier'* have no influence.

The model returns scores but classifies neither pixel nor image as showing an anomaly or not. For this classification, thresholds need to be given, setting the minimum score for a pixel or image to be regarded as anomalous. You can estimate possible thresholds using the procedure compute_dl_anomaly_thresholds. Applying these thresholds can be done with the procedure threshold_dl_anomaly_results. As results the procedure adds the following (threshold depending) entries into the dictionary DLResult of a sample:

anomaly_class
: The predicted class of the entire image (for the given threshold). For Global Context Anomaly Detection, depending on the used subnetworks, the anomaly class can also be calculated by the local (anomaly_class_local) and the global (anomaly_class_global) subnetwork only.

anomaly_class_id
: ID of the predicted class of the entire image (for the given threshold). For Global Context Anomaly Detection, depending on the used subnetworks, the anomaly class ID can also be calculated by the local (anomaly_class_id_local) and the global (anomaly_class_id_global) subnetwork only.

anomaly_region
: Region consisting of all the pixels that are regarded as showing an anomaly (for the given threshold, see the illustration below). For Global Context Anomaly Detection, depending on the used subnetworks, the anomaly region can also be calculated by the local (anomaly_region_local) and the global (anomaly_region_global) subnetwork only.



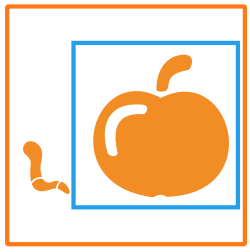
(1) (2)

Scheme of anomaly_region. For visualization purpose, gray values are used to represent numbers. (1) The anomaly_image with the obtained pixel scores. (2) The corresponding anomaly_region.

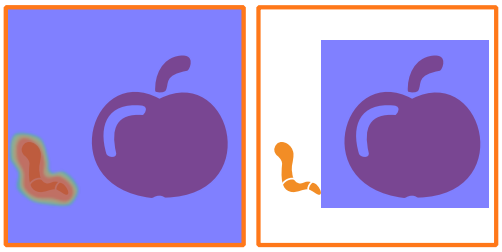
Domain Handling During Inference

A restriction of the search area can be done by reducing the domain of the input images (e.g., reduce_domain). The way preprocess_dl_samples handles the domain is set using the preprocessing parameter 'domain_handling'. The parameter 'domain_handling' should be used in a way that only essential information is passed on to the network for inference. For instance, use 'keep_domain' to exclude unwanted anomalies in the background when computing the anomaly score and image.

The following images show how an input image with reduced domain is inferred after the preprocessing step depending on the set 'domain_handling'.



Input image for inference with domain (blue).



(1) (2)

(1) anomaly_image after inference with 'full_domain' (result: *'nok'*), (2) anomaly_image after inference with 'keep_domain' (result: *'ok'*).

train_dl_model_anomaly_dataset (: : DLModelHandle, DLSamples, DLTrainParam : DLTrainResult)

Train a deep learning model for anomaly detection.

The operator train_dl_model_anomaly_dataset performs the training of a deep learning model with *'type'='anomaly_detection'* contained in DLModelHandle (for deep learning models with *'type'='gc_anomaly_detection'* see train_dl_model_batch).

This operator processes the full training dataset at once. This is in contrast to the operator train_dl_model_batch. The iterations over the dataset are performed internally by the operator. Consequently, you only need to call this operator once with the full training dataset to train your anomaly detection model.

The training dataset is handed over in the tuple of dictionaries DLSamples. See the chapter Deep Learning / Model for further information to the used dictionaries and their keys. The operator expects within the training dataset only images without anomaly to train the anomaly detection model.

The dictionary DLTrainParam can be used to change the hyperparameters. The following values are supported:

- max_num_epochs: This parameter specifies the maximum number of epochs performed during training. In case the criterion specified by error_threshold is reached in an earlier epoch, the training will terminate regardless.
  Restriction: max_num_epochs >=*1*.
  Default: max_num_epochs = *30*.
- error_threshold: This parameter is a termination criterion for the training. If the training error is less than the specified error_threshold, the training terminates successfully.
  Restriction:
  0.0 <= error_threshold <= *1.0*.
  Default: error_threshold = *0.001*.
- domain_ratio: This parameter determines the percentage of information of each image used for training. Since images tend to contain an abundance of information, it is advisable to reduce its amount. Additionally, reducing domain_ratio can decrease the time needed for training. Please note, however, sufficient information needs to remain and therefore this value should not be set too small either. Otherwise the training result might not be satisfactory or the training itself might even fail.
  Restriction: *0.0* < domain_ratio <= *1.0*.
  Default: domain_ratio = *0.1*.
- regularization_noise: This parameter can be set to regularize the training in order to improve robustness.
  Restriction: regularization_noise >=*0.0*.
  Default: regularization_noise = *0.0*.

The output dictionary DLTrainResult contains the following values:

- final_error: The best error received during training.
- final_epoch The epoch in which the error final_error was achieved.

Attention

The operator `train_dl_model_anomaly_dataset` internally calls functions that might not be deterministic. Therefore, results from multiple calls of `train_dl_model_anomaly_dataset` can slightly differ, although the same input values have been used.

System requirements: To run this operator on GPU by setting *'runtime'* to *'gpu'* (see `get_dl_model_param`), cuDNN and cuBLAS are required. For further details, please refer to the "`Installation Guide`", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods". Alternatively, this operator can also be run on CPU by setting *'runtime'* to *'cpu'*.

Parameters

- ▷ **`DLModelHandle`** (input_control) ..... dl_model ⇝ *handle*
  Deep learning model handle.
- ▷ **`DLSamples`** (input_control) ..... dict-array ⇝ *handle*
  Tuple of Dictionaries with input images and corresponding information.
- ▷ **`DLTrainParam`** (input_control) ..... dict ⇝ *handle*
  Parameter for training the anomaly detection model.
  Default: []
- ▷ **`DLTrainResult`** (output_control) ..... dict ⇝ *handle*
  Dictionary with the train result data.

Result

If the parameters are valid, the operator `train_dl_model_anomaly_dataset` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`read_dl_model`, `set_dl_model_param`, `get_dl_model_param`

Possible Successors

`apply_dl_model`

See also

`apply_dl_model`

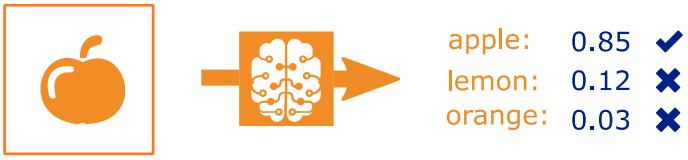
Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
Deep Learning Professional

9.2 Classification

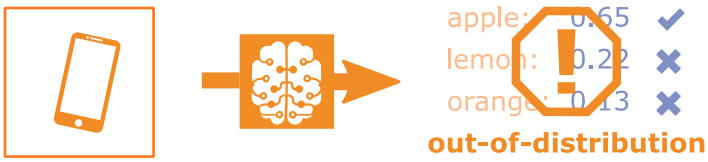
This chapter explains how to use classification based on deep learning, both for the training and inference phases.

Classification based on deep learning is a method, in which an image gets a set of confidence values assigned. These confidence values indicate how likely the image belongs to each of the distinguished classes. Thus, if we regard only the top prediction, classification means to assign a specific class out of a given set of classes to an image. This is illustrated in the following schema.



A possible classification example, in which the network distinguishes three classes. The input image gets confidence values assigned for each of the three distinguished classes: 'apple' 0.85, 'lemon' 0.03, and 'orange' 0.12. The top prediction tells us, the image is recognized as 'apple'.

Out-of-Distribution Detection for classification is a method for identifying inputs which differ significantly from the classes the model was trained on. It is crucial for ensuring model safety and robustness. Out-of-Distribution Detection helps to filter potentially problematic cases for further review. This is illustrated in the following schema.



A possible example of classification with the addition of Out-of-Distribution Detection. The object in the inference image differs significantly from the data used to train the network. In addition to the confidence values for the three classes to be distinguished ('apple' 0.65, 'lemon' 0.22, and 'orange' 0.13), the network also indicates that the image does not belong to any of the three trained classes (Out-of-Distribution).

In order to do your specific task, thus to classify your data into the classes you want to have distinguished, the classifier has to be trained accordingly. In HALCON, we use a technique called transfer learning (see also the chapter Deep Learning). Hence, we provide pretrained networks, representing classifiers which have been trained on huge amounts of labeled image data. These classifiers have been trained and tested to perform well on industrial image classification tasks. One of these classifiers, already trained for general classifications, is now retrained for your specific task. For this, the classifier needs to know, which classes are to be distinguished and how such examples look like. This is represented by your dataset, i.e., your images with the corresponding ground truth labels. More information on the data requirements can be found in the section "Data".

In HALCON, classification with deep learning is implemented within the more general deep learning model. For more information to the latter one, see the chapter Deep Learning / Model. For the specific system requirements in order to apply deep learning, please refer to the HALCON "`Installation Guide`".

The following sections are introductions to the general workflow needed for classification, information related to the involved data and parameters, and explanations to the evaluation measures.

General Workflow

In this paragraph, we describe the general workflow for a classification task based on deep learning. It is subdivided into the four parts preprocessing of the data, training of the model, evaluation of the trained model, and inference on new images. Thereby we assume, your dataset is already labeled, see also the section "Data" below. Have a look at the HDevelop example series `classify_pill_defects_deep_learning` for an application.

Preprocess the data This part is about how to preprocess your data. The single steps are also shown in the HDevelop example `classify_pill_defects_deep_learning_1_preprocess.hdev`.

1. The information what is to be found in which image of your training dataset needs to be transferred. This is done by the procedure
   - `read_dl_dataset_classification`.

   Thereby a dictionary `DLDataset` is created, which serves as a database and stores all necessary information about your data. For more information about the data and the way it is transferred, see the section "Data" below and the chapter Deep Learning / Model.
2. Split the dataset represented by the dictionary `DLDataset`. This can be done using the procedure
   - `split_dl_dataset`.

   The resulting split will be saved over the key `split` in each sample entry of `DLDataset`.
3. Read in a pretrained network using the operator
   - `read_dl_model`.

   This operator is likewise used when you want to read your own trained networks, after you saved them with `write_dl_model`.

   The network will impose several requirements on the images, as the image dimensions and the gray value range. The default values are listed in `read_dl_model`. These are the values with which the networks have been pretrained. The network architectures allow different image dimensions, which can be set with `set_dl_model_param`, but depending on the network a change may make a retraining necessary. The actually set values can be retrieved with

- get_dl_model_param.

4. Now you can preprocess your dataset. For this, you can use the procedure
   - preprocess_dl_dataset.

   In case of custom preprocessing, this procedure offers guidance on the implementation.

   To use this procedure, specify the preprocessing parameters as e.g., the image size. Store all the parameters with their values in a dictionary DLPreprocessParam, wherefore you can use the procedure
   - create_dl_preprocess_param.

   We recommend to save this dictionary DLPreprocessParam in order to have access to the preprocessing parameter values later during the inference phase.

Training of the model This part is about how to train a classifier. The single steps are also shown in the HDevelop example classify_pill_defects_deep_learning_2_train.hdev.

1. Set the training parameters and store them in the dictionary TrainParam. These parameters include:
   - the hyperparameters, for an overview see the chapter Deep Learning.
   - parameters for possible data augmentation (optional).
   - parameters for the evaluation during training.
   - parameters for the visualization of training results.
   - parameters for serialization.

   This can be done using the procedure
   - create_dl_train_param.
2. Train the model. This can be done using the procedure
   - train_dl_model.

   The procedure expects:
   - the model handle DLModelHandle
   - the dictionary with the data information DLDataset
   - the dictionary with the training parameter TrainParam
   - the information, over how many epochs the training shall run.

   In case the procedure train_dl_model is used, the total loss as well as optional evaluation measures are visualized.

Evaluation of the trained model In this part we evaluate the trained classifier. The single steps are also shown in the HDevelop example classify_pill_defects_deep_learning_3_evaluate.hdev.

1. The evaluation can conveniently be done using the procedure
   - evaluate_dl_model.
2. The dictionary EvaluationResult holds the asked evaluation measures. You can visualize your evaluation results using the procedure
   - dev_display_classification_evaluation.
3. A heatmap can be generated for specified samples using
   (a) the operator gen_dl_model_heatmap
   (b) the procedure gen_dl_model_classification_heatmap

Fit model to Out-of-Distribution Detection (optional) In this part, we extend the trained classifier so it can detect out-of-distribution data. The single steps are also shown in the HDevelop example detect_out_of_distribution_samples_for_classification.hdev.

1. Fitting the Out-of-Distribution Detection using
   - fit_dl_out_of_distribution.
2. *Optional step:* Add out-of-distribution data to the dataset for evaluation, using the procedure
   - add_dl_out_of_distribution_data.

3. Rerun the evaluation using the procedure
   - `evaluate_dl_model`.
4. The dictionary `EvaluationResult` holds the asked evaluation measures. You can visualize your evaluation results using the procedure
   - `dev_display_classification_evaluation`.

Inference on new images This part covers the application of a deep-learning-based classification model. The single steps are also shown in the HDevelop example `classify_pill_defects_deep_learning_4_infer.hdev`.

1. Set the parameters as e.g., *'batch_size'* using the operator
   - `set_dl_model_param`.
2. Generate a data dictionary `DLSample` for each image. This can be done using the procedure
   - `gen_dl_samples_from_images`.
3. Preprocess the images as done for the training. We recommend to do this using the procedure
   - `preprocess_dl_samples`.

   When you saved the dictionary `DLPreprocessParam` during the preprocessing step, you can directly use it as input to specify all parameter values.
4. Apply the model using the operator
   - `apply_dl_model`.
5. Retrieve the results from the dictionary *'DLResultBatch'*.

Data

We distinguish between data used for training and data for inference. Latter one consists of bare images. But for the former one you already know to which class the images belong and provide this information over the corresponding labels.

As a basic concept, the model handles data over dictionaries, meaning it receives the input data over a dictionary `DLSample` and returns a dictionary `DLResult` and `DLTrainResult`, respectively. More information on the data handling can be found in the chapter Deep Learning / Model.

Data for training and evaluation The dataset consists of images and corresponding information. They have to be provided in a way the model can process them. Concerning the image requirements, find more information in the section "Images" below.

The training data is used to train and evaluate a network for your specific task. With the aid of this data the classifier can learn which classes are to be distinguished and how their representatives look like. In classification, the image is classified as a whole. Therefore, the training data consists of images and their ground truth labels, thus the class you say this image belongs to. Note that the images should be as representative as possible for your task. There are different ways possible, how to store and retrieve this information. How the data has to be formatted in HALCON for a DL model is explained in the chapter Deep Learning / Model. In short, a dictionary `DLDataset` serves as a database for the information needed by the training and evaluation procedures. The procedure `read_dl_dataset_classification` supports the following sources of the ground truth label for an image:

- The last directory name containing the image
- The file name.

For training a classifier, we use a technique called transfer learning (see the chapter Deep Learning). For this, you need less resources, but still a suitable set of data. While in general the network should be more reliable when trained on a larger dataset, the amount of data needed for training also depends on the complexity of the task. You also want enough training data to split it into three subsets, used for training, validation, and testing the network. These subsets are preferably independent and identically distributed, see the section "Data" in the chapter Deep Learning.

Images Regardless of the application, the network poses requirements on the images regarding e.g., the image dimensions. The specific values depend on the network itself and can be queried with `get_dl_model_param`. In order to fulfill these requirements, you may have to preprocess your images. Standard preprocessing is implemented in `preprocess_dl_dataset` and in `preprocess_dl_samples` for a single sample, respectively. In case of custom preprocessing these procedures offer guidance on the implementation.

Network output The network output depends on the task:

training As output, the operator will return a dictionary `DLTrainResult` with the current value of the total loss as well as values for all other losses included in your model.

inference and evaluation As output, the network will return a dictionary `DLResult` for every sample. For classification, this dictionary will include for each input image a tuple with the confidence values for every class to be distinguished in decreasing order and a second tuple with the corresponding class IDs.

Interpreting the Classification Results

When we classify an image, we obtain a set of confidence values, telling us the affinity of the image to every class. It is also possible to compute the following values.

Confusion Matrix, Precision, Recall, and F-score In classification whole images are classified. As a consequence, the instances of a confusion matrix are images. See the chapter Deep Learning for explanations on confusion matrices.

You can generate a confusion matrix with the aid of the procedures `gen_confusion_matrix` and `gen_interactive_confusion_matrix`. Thereby, the interactive procedure gives you the possibility to select examples of a specific category, but it does not work with exported code.

From such a confusion matrix you can derive various values. The precision is the proportion of all correct predicted positives to all predicted positives (true and false ones). Thus, it is a measure of how many positive predictions really belong to the selected class.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

The recall, also called the "true positive rate", is the proportion of all correct predicted positives to all real positives. Thus, it is a measure of how many samples belonging to the selected class were predicted correctly as positives.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

A classifier with high recall but low precision finds most members of positives (thus members of the class), but at the cost of also classifying many negatives as member of the class. A classifier with high precision but low recall is just the opposite, classifying only few samples as positives, but most of these predictions are correct. An ideal classifier with high precision and high recall will classify many samples as positive with a high accuracy.

To represent this with a single number, we compute the F1-score, the harmonic mean of precision and recall. Thus, it is a measure of the classifier's accuracy.

$$\text{F1-score} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

For the example from the confusion matrix shown in Deep Learning we get for the class 'apple' the values precision: 1.00 (= 68/(68+0+0)), recall: 0.74 (= 68/(68+21+3)), and F1-score: 0.85 (=2*(1.00*0.74)/(1.00+0.74)).

fit_dl_out_of_distribution (: : DLModelHandle, DLDataset, GenParam :)

Extend a deep learning model for Out-of-Distribution Detection.

fit_dl_out_of_distribution extends a trained deep learning model DLModelHandle of *'type'* = *'classification'* for Out-of-Distribution Detection. This functionality allows the model to detect samples that differ significantly from the classes it was trained on, known as "Out-of-Distribution" (OOD) samples.

When apply_dl_model is called subsequently, the results will include the following additional entries related to Out-of-Distribution Detection:

'ood_result': Indicates whether the sample is predicted as out-of-distribution.

'ood_score': Indicates how much the sample differs from the trained classes. The higher this score, the more likely it is that the sample is out-of-distribution.

'ood_threshold': If *'ood_score'* exceeds this threshold, the sample is predicted as out-of-distribution. The out-of-distribution threshold is computed during the execution of fit_dl_out_of_distribution and stored within the model handle DLModelHandle as 'ood_threshold'. If required, the threshold can be adjusted manually using the operator set_dl_model_param.

For fit_dl_out_of_distribution to work properly, it is important that DLDataset is the same dataset with the same split and preprocessing parameters, as the one used for training DLModelHandle. It is crucial that the provided dataset DLDataset contains diverse and sufficient samples for each class to ensure reliable Out-of-Distribution Detection. If the dataset is too small or lacks variation, fit_dl_out_of_distribution may return an error. In such cases, additional training data should be added to the dataset.

fit_dl_out_of_distribution can be applied to any classification model supported by HALCON. For models created using Deep Learning / Framework operators or read from an ONNX model file, Out-of-Distribution Detection compatibility may vary depending on the architecture.

The performance of the model for Out-of-Distribution Detection can be evaluated using the procedure evaluate_dl_model. To evaluate the model on out-of-distribution data, these can be added to the DLDataset using the procedure add_dl_out_of_distribution_data, allowing for testing whether the model can accurately separate in-distribution from out-of-distribution data. Adjustments to the *'ood_threshold'* will affect evaluation results. Therefore, it is recommended to re-evaluate the model after making such changes.

GenParam is a dictionary for setting generic parameters. Currently no generic parameters are supported.

Attention

If fit_dl_out_of_distribution is called for a model that has already been extended with Out-of-Distribution Detection, the previous internal calculations are discarded and the model is adapted anew

Certain modifications to the model, such as changing the number of classes or continuing training of the model, cannot be performed once the model has been extended for Out-of-Distribution Detection. To make such changes possible, the model internal Out-of-Distribution Detection must first be removed from the model using the parameter *'clear_ood'* in set_dl_model_param. Once removed, fit_dl_out_of_distribution can be called again to re-enable Out-of-Distribution Detection.

Parameters

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Handle of a deep learning classification model.

▷ **DLDataset** (input_control) ..... dict ⇝ *handle*
Dataset, which was used for training the model.

▷ **GenParam** (input_control) ..... dict ⇝ *handle*
Dictionary for generic parameters.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_dl_model

Module

Deep Learning Professional

9.3 Framework

create_dl_layer_activation (: : DLLayerInput, LayerName, ActivationType, GenParamName, GenParamValue : DLLayerActivation)

Create an activation layer.

The operator create_dl_layer_activation creates an activation layer whose handle is returned in DLLayerActivation.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter ActivationType sets the type of the activation. Supported activation types are:

'relu': Rectified linear unit (ReLU) activation. By setting a specific ReLU parameter, another type can be specified instead of the standard ReLU:

- Standard ReLU, defined as follows:

$$ReLU(x) := \max(0, x)$$

- Bounded ReLU, defined as follows:

$$ReLU(x) := \begin{cases} 0 & \text{if } x \leq 0, \\ x & \text{if } 0 < x \leq \beta, \\ \beta & \text{else.} \end{cases}$$

  Setting the generic parameter *'upper_bound'* will result in a bounded ReLU and determines the value of β.

- Leaky ReLU:, defined as follows:

$$ReLU(x) := \begin{cases} \alpha x & \text{if } x \leq 0, \\ x & \text{else.} \end{cases}$$

  Setting the generic parameter *'leaky_relu_alpha'* results in a leaky ReLU and determines the value α.

'sigmoid': Sigmoid activation, which is defined as follows.

$$Sigmoid(x) := \frac{1}{1 + e^{-x}}$$

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

'upper_bound': Float value defining an upper bound for a rectified linear unit. If the activation layer is part of a model which has been created using create_dl_model, the upper bound can be unset. To do so, use set_dl_model_layer_param and set an empty tuple for *'upper_bound'*.
Default: *[]*

'leaky_relu_alpha': Float value defining the alpha parameter of a leaky ReLU.

Restriction: The value of *'leaky_relu_alpha'* must be positive or zero.

Default: *0.0*

Certain parameters of layers created using this operator `create_dl_layer_activation` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer Parameters	set	get
'activation_type' (`ActivationType`)	x	x
'input_layer' (`DLLayerInput`)		x
'name' (`LayerName`)	x	x
'output_layer' (`DLLayerActivation`)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'leaky_relu_alpha'	x	x
'num_trainable_params'		x
'upper_bound'	x	x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **ActivationType** (input_control) . . . . . string ⇝ *string*
Activation type.
Default: 'relu'
List of values: `ActivationType` ∈ {'relu', 'sigmoid'}

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'is_inference_output', 'upper_bound', 'leaky_relu_alpha'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {'true', 'false'}

▷ **DLLayerActivation** (output_control) . . . . . dl_layer ⇝ *handle*
Activation layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_batch_normalization (: : DLLayerInput, LayerName, Momentum, Epsilon, Activation, GenParamName, GenParamValue : DLLayerBatchNorm)

Create a batch normalization layer.

The operator create_dl_layer_batch_normalization creates a batch normalization layer whose handle is returned in DLLayerBatchNorm. Batch normalization is used to improve the performance and stability of a neural network during training. The mean and variance of each input activation are calculated for each batch and the input values are transformed to have zero mean and unit variance. Moreover, a linear scale and shift transformation is learned. During training, to take all samples into account, the batch-wise calculated mean and variance values are combined with a Momentum into running mean and running variance, where i denotes the iteration index:

$$\begin{aligned} running\_mean(i) &= (1 - \texttt{Momentum}) * mean(i) + \texttt{Momentum} * running\_mean(i-1), \\ running\_variance(i) &= (1 - \texttt{Momentum}) * variance(i) + \texttt{Momentum} * running\_variance(i-1). \end{aligned}$$

To affect the mean and variance values you can set the following options for Momentum:

Given number: For example: *0.9*. This is the default and recommended option.
Restriction: $0 \leq \texttt{Momentum} < 1$

'auto': Combines mean and variance values by a cumulative moving average. This is only recommended in case the parameters of all previous layers in the network are frozen, i.e., have a learning rate of *0*.

'freeze': Stops the adjustment of the mean and variance and their values stay fixed. In this case, the mean and variance are used during training for normalizing a batch, analogously to how the batch normalization operates during inference. The parameters of the linear scale and shift transformation, however, remain learnable.

Epsilon is a small offset to the variance and used to control the numerical stability. Usually its default value should be adequate.

The parameter DLLayerInput determines the feeding input layer.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter Activation determines whether an activation is performed after the batch normalization in order to optimize the runtime performance.

- *'relu'*: perform a ReLU activation after the batch normalization.
  It is possible to specify an upper bound to the ReLU operation (see create_dl_layer_activation) via the generic parameter *'upper_bound'*.
- *'none'*: no activation operation is performed.

It is not possible to specify a leaky ReLU or a sigmoid activation function. Use create_dl_layer_activation instead.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'bias_filler': See create_dl_layer_convolution for a detailed explanation of this parameter and its values.
List of values: *'xavier'*, *'msra'*, *'const'*.
Default: *'const'*

'bias_filler_const_val': Constant value.
Restriction: *'bias_filler'* must be set to *'const'*.
Default: *0*

'bias_filler_variance_norm': See create_dl_layer_convolution for a detailed explanation of this parameter and its values.
List of values: *'norm_out'*, *'norm_in'*, *'norm_average'*, or constant value (in combination with *'bias_filler'* = *'msra'*).
Default: *'norm_out'*

'bias_term': Determines whether the created batch normalization layer has a bias term (*'true'*) or not (*'false'*).

Default: *'true'*

'is_inference_output': Determines whether `apply_dl_model` will include the output of this layer in the dictionary `DLResultBatch` even without specifying this layer in `Outputs` (*'true'*) or not (*'false'*).

Default: *'false'*

'learning_rate_multiplier': Multiplier for the learning rate for this layer that is used during training. If *'learning_rate_multiplier'* is set to *0.0*, the layer is skipped during training.

Default: *1.0*

'learning_rate_multiplier_bias': Multiplier for the learning rate of the bias term. The total bias learning rate is the product of *'learning_rate_multiplier_bias'* and *'learning_rate_multiplier'*.

Default: *1.0*

'upper_bound': Float value defining an upper bound for a rectified linear unit. If the activation layer is part of a model, which has been created using `create_dl_model`, the upper bound can be unset. To do so, use `set_dl_model_layer_param` and set an empty tuple for *'upper_bound'*.

Default: *[]*

'weight_filler': See `create_dl_layer_convolution` for a detailed explanation of this parameter and its values.

List of values: *'xavier'*, *'msra'*, *'const'*.

Default: *'const'*

'weight_filler_const_val': See `create_dl_layer_convolution` for a detailed explanation of this parameter and its values.

Default: *1.0*

'weight_filler_variance_norm': See `create_dl_layer_convolution` for a detailed explanation of this parameter and its values.

List of values: *'norm_in'*, *'norm_out'*, *'norm_average'*, or constant value (in combination with *'weight_filler'* = *'msra'*).

Default: *'norm_in'*

Certain parameters of layers created using this operator `create_dl_layer_batch_normalization` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer Parameters	set	get
'activation_mode' (`Activation`)		x
'epsilon' (`Epsilon`)		x
'input_layer' (`DLLayerInput`)		x
'momentum' (`Momentum`)	x	x
'name' (`LayerName`)	x	x
'output_layer' (`DLLayerBatchNorm`)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'bias_filler'	x	x
'bias_filler_const_val'	x	x
'bias_filler_variance_norm'	x	x
'bias_term'		x
'is_inference_output'	x	x
'learning_rate_multiplier'	x	x
'learning_rate_multiplier_bias'	x	x
'num_trainable_params'		x
'upper_bound'	x	x
'weight_filler'	x	x
'weight_filler_const_val'	x	x
'weight_filler_variance_norm'	x	x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **Momentum** (input_control) . . . . . string ⇝ *string* / real
Momentum.
Default: 0.9
List of values: Momentum ∈ {0.9, 0.99, 0.999, 'auto', 'freeze'}

▷ **Epsilon** (input_control) . . . . . number ⇝ *real*
Variance offset.
Default: 0.0001

▷ **Activation** (input_control) . . . . . string ⇝ *string*
Optional activation function.
Default: 'none'
List of values: Activation ∈ {'none', 'relu'}

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'bias_filler', 'bias_filler_variance_norm', 'bias_filler_const_val', 'bias_term', 'is_inference_output', 'learning_rate_multiplier', 'learning_rate_multiplier_bias', 'upper_bound', 'weight_filler', 'weight_filler_variance_norm', 'weight_filler_const_val'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'xavier', 'msra', 'const', 'nearest_neighbor', 'bilinear', 'norm_in', 'norm_out', 'norm_average', 'true', 'false', 1.0, 0.9, 0.0}

▷ **DLLayerBatchNorm** (output_control) . . . . . dl_layer ⇝ *handle*
Batch normalization layer.

Example

```
create_dl_layer_input ('input', [224,224,3], [], [], DLLayerInput)
* In practice, one typically sets ['bias_term'], ['false'] for a convolution
* that is directly followed by a batch normalization layer.
create_dl_layer_convolution (DLLayerInput, 'conv1', 3, 1, 1, 64, 1, \
                             'none', 'none', ['bias_term'], ['false'], \
                             DLLayerConvolution)
create_dl_layer_batch_normalization (DLLayerConvolution, 'bn1', 0.9, \
                                     0.0001, 'none', [], [], \
                                     DLLayerBatchNorm)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_dl_layer_convolution

Possible Successors

create_dl_layer_activation, create_dl_layer_convolution

References

Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Proceedings of the 32nd International Conference on Machine Learning, (ICML) 2015, Lille, France, 6-11 July 2015, pp. 448–456

Module

Deep Learning Professional

create_dl_layer_class_id_conversion (: : DLLayerInput, LayerName, ConversionMode, GenParamName, GenParamValue : DLLayerClassIdConversion)

Create a class ID conversion layer.

The operator create_dl_layer_class_id_conversion creates a class ID conversion layer whose handle is returned in DLLayerClassIdConversion. The layer converts between the IDs used internally by the network and the target / output class IDs.

The network internally uses consecutive integer values starting from 0 as IDs (the number of values depends on the model type). In case the target / output class IDs differ from the internal IDs, this layer can be used to convert between them. The target / output class IDs are stored in the model parameter *'class_ids'* (see get_dl_model_param for more information on this parameter). If no *'class_ids'* are set, this layer copies the input to the output.

The parameter ConversionMode specifies the conversion direction and accepts the following values:

- *'from_class_id'*: Convert target / output class IDs into internal IDs. This mode is typically used after a target input layer.
- *'to_class_id'*: Convert internal IDs into target / output class IDs. This mode is typically used after an inference output layer.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_class_id_conversion can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer' (DLLayerInput)		x
'name' (LayerName)	x	x
'output_layer' (DLLayerClassIdConversion)		x
'shape'		x
'to_class_id' (ConversionMode)		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) ..... dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **ConversionMode** (input_control) ..... string ⇝ *string*
Direction of the class ID conversion.
Default: 'from_class_id'
List of values: ConversionMode ∈ {'from_class_id', 'to_class_id'}

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerClassIdConversion** (output_control) ..... dl_layer ⇝ *handle*
Class IDs conversion layer.

Example

```
* Example demonstrating the usage of
* create_dl_layer_class_id_conversion.
*
dev_update_off ()
set_system ('seed_rand', 42)
*
* Create simple segmentation model.
NumClasses := 3
InputShape := [32, 32, 3]
*
* Input feeding layers.
create_dl_layer_input ('image', InputShape, [], [], DLLayerInput)
create_dl_layer_input ('target', [InputShape[0],InputShape[1],1], [], [], \
                       DLLayerTarget)
create_dl_layer_class_id_conversion (DLLayerTarget, 'target_internal', \
                                     'from_class_id', [], [], \
                                     DLLayerTargetInternal)
* Feature extraction layers.
create_dl_layer_convolution (DLLayerInput, 'conv1', 3, 1, 1, 32, 1, \
                             'half_kernel_size', 'relu', [], [], \
                             DLLayerConv1)
create_dl_layer_convolution (DLLayerConv1, 'conv2', 3, 1, 1, 32, 1, \
```

```
                              'half_kernel_size', 'relu', [], [], \
                              DLLayerConv2)
* Output generation layers.
create_dl_layer_convolution (DLLayerConv2, 'conv_final', 1, 1, 1, \
                              NumClasses, 1, 'none', 'none', [], [], \
                              DLLayerConvFinal)
create_dl_layer_softmax (DLLayerConvFinal, 'softmax', [], [], \
                         DLLayerSoftMax)
create_dl_layer_depth_max (DLLayerSoftMax, 'output_internal', \
                           'argmax', [], [], DLLayerOutputInternal, _)
create_dl_layer_class_id_conversion (DLLayerOutputInternal, 'output', \
                                     'to_class_id', [], [], DLLayerOutput)
* Loss layer.
create_dl_layer_loss_cross_entropy (DLLayerSoftMax, DLLayerTargetInternal, \
                                    [], 'loss', 1.0, [], [], DLLayerLoss)
*
* Create the model.
create_dl_model ([DLLayerOutput, DLLayerLoss], DLModelHandle)
set_dl_model_param (DLModelHandle, 'type', 'segmentation')
set_dl_model_param (DLModelHandle, 'runtime', 'cpu')
*
* Test model on dummy example data.
read_image (Image, 'claudia')
zoom_image_size (Image, Image, InputShape[0], InputShape[1], 'constant')
convert_image_type (Image, Image, 'real')
*
* Fill target image with specific target class IDs.
ClassIDs := [42, 17, 5]
gen_image_const (Target, 'real', InputShape[0], InputShape[1])
paint_region (Target, Target, Target, ClassIDs[0], 'fill')
gen_rectangle1 (RectClass1, 1, 3, 16, 27)
paint_region (RectClass1, Target, Target, ClassIDs[1], 'fill')
gen_rectangle1 (RectClass2, 19, 1, 30, 30)
paint_region (RectClass2, Target, Target, ClassIDs[2], 'fill')
*
* Set class IDs in the model.
set_dl_model_param (DLModelHandle, 'class_ids', ClassIDs)
*
* Create test sample.
create_dict (DLSample)
set_dict_object (Image, DLSample, 'image')
set_dict_object (Target, DLSample, 'target')
*
* Train model for a few iterations. Note that training would not
* work without the first class ID conversion layer 'target_internal'.
for Idx := 1 to 100 by 1
    train_dl_model_batch (DLModelHandle, DLSample, DLTrainResult)
endfor
*
* Apply model on test image. With the second class ID conversion
* layer 'output', the image now contains values according to the
* target IDs in segmentation_image.
apply_dl_model (DLModelHandle, DLSample, [], DLApplyResult)
get_dict_object (SegmentationImage, DLApplyResult, 'output')
dev_display (SegmentationImage)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_concat (: : DLLayerInputs, LayerName, Axis, GenParamName, GenParamValue : DLLayerConcat)

Create a concatenation layer.

The operator create_dl_layer_concat creates a concatenation layer whose handle is returned in DLLayerConcat.

The parameter DLLayerInputs determines the feeding input layers. This layer expects multiple layers as input.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

A concatenation layer concatenates the data tensors of the input layers in DLLayerInputs and returns a single data tensor DLLayerConcat. The parameter Axis specifies along which dimension the inputs should be concatenated. The supported options for Axis are:

'batch': Concatenation is applied along the batch-dimension.
Example: if you concatenate two inputs *A* and *B* of shape (h, w, d, b) = (1, 1, 1, 2), where *A* = [*A0*, *A1*] and *B* = [*B0*, *B1*], you obtain the output [*A0*, *A1*, *B0*, *B1*] with shape (1, 1, 1, 4).

'batch_interleaved': Concatenation is applied along the depth-dimension, but the output is reshaped as if the data was concatenated along the batch-dimension. For this dimension, all inputs need to have exactly the same shape.
Note that when the input batch_size is *1*, the concatenation is identical for *'batch'* and *'batch_interleaved'*.
Example: if you concatenate two inputs *A* and *B* of shape (h, w, d, b) = (1, 1, 1, 2), where *A* = [*A0*, *A1*] and *B* = [*B0*, *B1*], you obtain the output [*A0*, *B0*, *A1*, *B1*] with shape (1, 1, 1, 4).

'depth': Concatenation is applied along the depth-dimension.
Example: if you concatenate two inputs *A* and *B* of shape (h, w, d, b) = (1, 1, 1, 2), where *A* = [*A0*, *A1*] and *B* = [*B0*, *B1*], you obtain the output [*A0*, *A1*, *B0*, *B1*] with shape (1, 1, 2, 2).

'height': Concatenation is applied along the height-dimension.

'width': Concatenation is applied along the width-dimension.

Note that all non-concatenated dimensions must be equal for all input data tensors.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_concat can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer' (DLLayerInputs)		x
'name' (LayerName)	x	x
'output_layer' (DLLayerConcat)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInputs** (input_control) ..... dl_layer(-array) ⇝ *handle*
Feeding input layers.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **Axis** (input_control) ..... string ⇝ *string*
Dimension along which the input layers are concatenated.
Default: 'depth'
List of values: Axis ∈ {'batch', 'batch_interleaved', 'depth', 'height', 'width'}

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerConcat** (output_control) ..... dl_layer ⇝ *handle*
Concatenation layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_convolution (: : DLLayerInput, LayerName, KernelSize, Dilation, Stride, NumKernel, Groups, Padding, Activation, GenParamName, GenParamValue : DLLayerConvolution)

Create a convolutional layer.

The operator create_dl_layer_convolution creates a convolutional layer with NumKernel kernels in Groups filter groups whose handle is returned in DLLayerConvolution.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter KernelSize specifies the filter kernel in the dimensions width and height.

The parameter Dilation specifies the factor of filter dilation in the dimensions width and height.

The parameter Stride specifies how the filter is shifted.

The values for KernelSize, Dilation, and Stride can be set as

- a single value which is used for both dimensions
- a tuple [width, height] and [column, row], respectively.

The parameter Groups specifies the number of filter groups.

The parameter NumKernel specifies the number of filter kernels. NumKernel must be a multiple of Groups.

The parameter Padding determines the padding, thus how many pixels with value 0 are appended on the border of the processed input image. Supported values are:

- *'half_kernel_size'*: The number of appended pixels depends on the specified KernelSize. More precisely, it is calculated as $\lfloor \texttt{KernelSize}/2 \rfloor$, where for the padding on the left / right border the value of KernelSize in dimension width is regarded and for the padding on the upper / lower border the value of KernelSize in height.
- *'none'*: No pixels are appended.
- Number of pixels: Specify the number of pixels appended on each border. To do so, the following tuple lengths are supported:
  - Single number: Padding in all four directions left/right/top/bottom.
  - Two numbers: Padding in left/right and top/bottom: [l/r, t/b].
  - Four numbers: Padding on left, right, top, bottom side: [l,r,t,b].
    Restriction: *'runtime' 'gpu'* does not support asymmetric padding, i.e., the padding values for the left and right side must be equal, as well as the padding values for the top and bottom side.

  Restriction: The integer padding values must be smaller than the value set for KernelSize in the corresponding dimension.

The output dimensions of the convolution layer are given by

$$output\_dim = \left\lfloor \frac{input\_dim + 2(padding\_begin + padding\_end)}{\texttt{Stride}} - \frac{\texttt{KernelSize} + (\texttt{KernelSize} - 1)(\texttt{Dilation} - 1)}{\texttt{Stride}} \right\rfloor + 1$$

Thereby we use the following values: $output\_dim$: output width/height, $input\_dim$: input width/height, $padding\_begin$: number of pixels added to the left/top of the input image, and $padding\_end$: number of pixels added to the right/bottom of the input image.

The parameter Activation determines whether an activation is performed after the convolution in order to optimize the runtime performance. The following values are supported:

- *'relu'*: perform a ReLU activation after the convolution.
  It is possible to specify an upper bound to the ReLU operation (see create_dl_layer_activation) via the generic parameter *'upper_bound'*. Note: It is not possible to specify a leaky ReLU.
- *'none'*: no activation operation is performed.

We refer to the "Solution Guide on Classification" for more general information about the convolution layer and the reference given below for more detailed information about the arithmetic of the layer.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'bias_filler': See *'weight_filler'* for an explanation of the values.
List of values: *'xavier'*, *'msra'*, *'const'*.
Default: *'const'*

'bias_filler_variance_norm': See *'weight_filler_variance_norm'* for an explanation of the values.
List of values: *'norm_in'*, *'norm_out'*, *'norm_average'*.
Default: *'norm_out'*

'bias_filler_const_val': Specifies the constant bias term initialization value if *'bias_filler'* has been set to *'const'*.
Restriction: Ignored for other values of *'bias_filler'*.
Default: *0*

'bias_term': Determines whether the created convolutional layer has a bias term (*'true'*) or not (*'false'*).
Default: *'true'*

'is_inference_output': Determines whether `apply_dl_model` will include the output of this layer in the dictionary `DLResultBatch` even without specifying this layer in `Outputs` (*'true'*) or not (*'false'*).

Default: *'false'*

'learning_rate_multiplier': Multiplier for the learning rate for this layer that is used during training. If *'learning_rate_multiplier'* is set to *0.0*, the layer is skipped during training.

Default: *1.0*

'learning_rate_multiplier_bias': Multiplier for the learning rate of the bias term. The total bias learning rate is the product of *'learning_rate_multiplier_bias'* and *'learning_rate_multiplier'*.

Default: *1.0*

'upper_bound': Float value, which defines the upper bound for ReLU. To unset the upper bound, set *'upper_bound'* to an empty tuple.

Default: *[]*

'weight_filler': This parameter defines the mode how the weights are initialized. The following values are supported:

- *'const'*: The weights are filled with constant values.
- *'msra'*: The weights are drawn from a Gaussian distribution.
- *'xavier'*: The weights are drawn from a uniform distribution.

Default: *'xavier'*

'weight_filler_const_val': Specifies the constant weight initialization value.

Restriction: Only applied if *'weight_filler'* = *'const'*.

Default: *0.5*

'weight_filler_variance_norm': This parameter determines the value range for *'weight_filler'*. The following values are supported:

- *'norm_average'*: the values are based on the average of the input and output size
- *'norm_in'*: the values are based on the input size
- *'norm_out'*: the values are based on the output size.

Default: *'norm_in'*

Certain parameters of layers created using `create_dl_layer_convolution` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer Parameters	set	get
'activation_mode' (`Activation`)		x
'dilation' (`Dilation`)		x
'groups' (`Groups`)		x
'input_depth'		x
'input_layer' (`DLLayerInput`)		x
'kernel_size' (`KernelSize`)		x
'name' (`LayerName`)	x	x
'num_kernels' (`NumKernel`)		x
'output_layer' (`DLLayerConvolution`)		x
'padding' (`Padding`)		x
'padding_type' (`Padding`)		x
'shape'		x
'stride' (`Stride`)		x
'type'		x

Generic Layer Parameters	set	get
'bias_filler'	x	x
'bias_filler_const_val'	x	x
'bias_filler_variance_norm'	x	x
'bias_term'		x
'is_inference_output'	x	x
'learning_rate_multiplier'	x	x
'learning_rate_multiplier_bias'	x	x
'num_trainable_params'		x
'upper_bound'	x	x
'weight_filler'	x	x
'weight_filler_const_val'	x	x
'weight_filler_variance_norm'	x	x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **KernelSize** (input_control) . . . . . number(-array) ⇝ *integer*
Width and height of the filter kernels.
Default: 3

▷ **Dilation** (input_control) . . . . . number(-array) ⇝ *integer*
Amount of filter dilation for width and height.
Default: 1

▷ **Stride** (input_control) . . . . . number(-array) ⇝ *integer*
Amount of filter shift in width and height direction.
Default: 1

▷ **NumKernel** (input_control) . . . . . number ⇝ *integer*
Number of filter kernels.
Default: 64

▷ **Groups** (input_control) . . . . . number ⇝ *integer*
Number of filter groups.
Default: 1

▷ **Padding** (input_control) . . . . . number(-array) ⇝ *string* / integer
Padding type or specific padding size.
Default: 'none'
List of values: `Padding` ∈ {'none', 'half_kernel_size', [all], [width,height], [left,right,top,bottom]}
Suggested values: `Padding` ∈ {'none', 'half_kernel_size'}

▷ **Activation** (input_control) . . . . . number ⇝ *string*
Enable optional ReLU or sigmoid activations.
Default: 'none'
List of values: `Activation` ∈ {'none', 'relu', 'sigmoid'}

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'weight_filler', 'weight_filler_variance_norm', 'weight_filler_const_val', 'bias_filler', 'bias_filler_variance_norm', 'bias_filler_const_val', 'bias_term', 'is_inference_output', 'learning_rate_multiplier', 'learning_rate_multiplier_bias', 'upper_bound'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {'xavier', 'msra', 'const', 'nearest_neighbor', 'bilinear', 'norm_in', 'norm_out', 'norm_average', 'true', 'false', 1.0, 0.9, 0.0}

▷ **DLLayerConvolution** (output_control) . . . . . dl_layer ⇝ *handle*
Convolutional layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

References

V. Dumoulin, F. Visin: "A guide to convolution arithmetic for deep learning", 2018, http://arxiv.org/abs/1603.07285

Module

Deep Learning Professional

create_dl_layer_dense (: : DLLayerInput, LayerName, NumOut, GenParamName, GenParamValue : DLLayerDense)

Create a dense layer.

The operator create_dl_layer_dense creates a dense or fully connected layer (sometimes also called gemm) with NumOut output neurons whose handle is returned in DLLayerDense.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'bias_filler': See create_dl_layer_convolution for a detailed explanation of this parameter and its values.
List of values: *'xavier'*, *'msra'*, *'const'*.
Default: *'const'*

'bias_filler_const_val': Constant value if *'bias_filler'* = *'const'*.
Default: *0*

'bias_filler_variance_norm': See create_dl_layer_convolution for a detailed explanation of this parameter and its values.
List of values: *'norm_out'*, *'norm_in'*, *'norm_average'*, or constant value (in combination with *'bias_filler'* = *'msra'*).
Default: *'norm_out'*

'bias_term': Determines whether the created dense layer has a bias term (*'true'*) or not (*'false'*).
Default: *'true'*

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

'learning_rate_multiplier': Multiplier for the learning rate for this layer that is used during training. If *'learning_rate_multiplier'* is set to *0.0*, the layer is skipped during training.
Default: *1.0*

'learning_rate_multiplier_bias': Multiplier for the learning rate of the bias term. The total bias learning rate is the product of *'learning_rate_multiplier_bias'* and *'learning_rate_multiplier'*.
Default: *1.0*

'weight_filler': See create_dl_layer_convolution for a detailed explanation of this parameter and its values.
List of values: *'xavier'*, *'msra'*, *'const'*.
Default: *'xavier'*

'weight_filler_const_val': See create_dl_layer_convolution for a detailed explanation of this parameter and its values.

Default: *0.5*

'weight_filler_variance_norm': See create_dl_layer_convolution for a detailed explanation of this parameter and its values.

List of values: *'norm_in'*, *'norm_out'*, *'norm_average'*, or constant value (in combination with *'bias_filler'* = *'msra'*).

Default: *'norm_in'*

Certain parameters of layers created using create_dl_layer_dense can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer' (DLLayerInput)		x
'name' (LayerName)	x	x
'neurons_in'		x
'neurons_out' (NumOut)		x
'output_layer' (DLLayerDense)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'bias_filler'	x	x
'bias_filler_const_val'	x	x
'bias_filler_variance_norm'	x	x
'bias_term'		x
'is_inference_output'	x	x
'learning_rate_multiplier'	x	x
'learning_rate_multiplier_bias'	x	x
'num_trainable_params'		x
'weight_filler'	x	x
'weight_filler_const_val'	x	x
'weight_filler_variance_norm'	x	x

Parameters

▷ **DLLayerInput** (input_control) ..... dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **NumOut** (input_control) ..... number ⇝ *integer*
Number of output neurons.
Default: 100

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'weight_filler', 'weight_filler_variance_norm', 'weight_filler_const_val', 'bias_filler', 'bias_filler_variance_norm', 'bias_filler_const_val', 'bias_term', 'is_inference_output', 'learning_rate_multiplier', 'learning_rate_multiplier_bias'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'xavier', 'msra', 'const', 'nearest_neighbor', 'bilinear', 'norm_in', 'norm_out', 'norm_average', 'true', 'false', 1.0, 0.9, 0.0}

▷ **DLLayerDense** (output_control) . . . . . dl_layer ⇝ *handle*
Dense layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_depth_max (: : DLLayerInput, LayerName, DepthMaxMode, GenParamName, GenParamValue : DLLayerDepthMaxArg, DLLayerDepthMaxValue)

Create a depth max layer.

The operator create_dl_layer_depth_max creates a depth max layer.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

There are two possible output layers depending on DepthMaxMode:

- DLLayerDepthMaxArg: Handle to a depth max layer with mode *'argmax'*.
- DLLayerDepthMaxValue: Handle to a depth max layer with mode *'value'*.

Note, these parameters only need to be set in case such an output layer is requested (see DepthMaxMode).

The parameter LayerName defines the name of the output layer(s) depending on DepthMaxMode:

- *'argmax'*: name of DLLayerDepthMaxArg.
- *'value'*: name of DLLayerDepthMaxValue.
- *'argmax_and_value'*: name of DLLayerDepthMaxArg, while the layer DLLayerDepthMaxValue receives the same name with the suffix string *'_value'* appended to it.

Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The mode DepthMaxMode indicates which depth max value is actually returned as output. The following values are supported:

'argmax': *newline* The depth index of the maximal value is returned in DLLayerDepthMaxArg.

'value': *newline* The maximal value itself is returned in DLLayerDepthMaxValue.

'argmax_and_value': *newline* Both are returned, the depth index of the maximal value in the output layer DLLayerDepthMaxArg, and the maximal value itself in the output layer DLLayerDepthMaxValue.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_depth_max can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer' (DLLayerInput)		x
'mode' (DepthMaxMode)	x	x
'name' (LayerName)	x	x
'output_layer' (DLLayerDepthMaxArg and/or DLLayerDepthMaxValue)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'		x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **DepthMaxMode** (input_control) . . . . . string ⇝ *string*
Mode to indicate type of return value.
Default: 'argmax'
List of values: DepthMaxMode ∈ {'argmax', 'value', 'argmax_and_value'}

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerDepthMaxArg** (output_control) . . . . . dl_layer(-array) ⇝ *handle*
Optional, depth max layer with mode *'argmax'*.

▷ **DLLayerDepthMaxValue** (output_control) . . . . . dl_layer(-array) ⇝ *handle*
Optional, depth max layer with mode *'value'*.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

```
create_dl_layer_depth_to_space ( : : DLLayerInput, LayerName,
    BlockSize, Mode, GenParamName,
    GenParamValue : DLLayerDepthToSpace )
```

Create a depth to space layer.

The operator create_dl_layer_depth_to_space creates a depth to space layer whose handle is returned in DLLayerDepthToSpace.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

This layer rearranges the elements of the feeding tensor of shape $(N, C*r^2, H, W)$ to a tensor of shape $(N, C, H*r, W*r)$. Thereby r can be considered an upscale factor, which is set with BlockSize.

The output element $(depth, row, col)$ is mapped from the input element $(depth * r^2 + (row\%r) * r + col\%r, row/r, col/r)$.

With Mode the ordering in the output tensor is set. Currently only the *'column_row_depth'* order described above is available.

Certain parameters of layers created using this operator create_dl_layer_depth_to_space can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer' (DLLayerInput)		x
'name' (LayerName)	x	x
'block_size' (BlockSize)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

- ▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
  Feeding layer.
- ▷ **LayerName** (input_control) . . . . . string ⇝ *string*
  Name of the output layer.
- ▷ **BlockSize** (input_control) . . . . . number ⇝ *integer*
  Block size (i.e., upscale factor).
  Default: 3
- ▷ **Mode** (input_control) . . . . . string ⇝ *string*
  Ordering mode.
  Default: 'column_row_depth'
- ▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
  Generic input parameter names.
  Default: []
  List of values: GenParamName ∈ {'is_inference_output'}
- ▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
  Generic input parameter values.
  Default: []
  Suggested values: GenParamValue ∈ {'true', 'false'}
- ▷ **DLLayerDepthToSpace** (output_control) . . . . . dl_layer ⇝ *handle*
  Depth to space layer.

Example

```
InputShape := [16, 16, 3]
Upscale := 2
*
create_dl_layer_input ('input', InputShape, [], [], DLLayerInput)
* Create a convolutional layer, that generates Upscale^2*NumChannel feature maps.
create_dl_layer_convolution (DLLayerInput, 'conv1', 3, 1, 1,\
                             Upscale * Upscale * InputShape[2],\
                             1, 'half_kernel_size', 'none',\
                             [],  [], DLLayerConvolution)
* Use a depth to space layer to combine Upscale^2 feature maps to upscale.
create_dl_layer_depth_to_space (DLLayerConvolution, 'upscaled', Upscale,\
                                'column_row_depth',[], [],\
                                DLLayerDepthToSpace)
* The output shape of DLLayerDepthToSpace is now [16*Upscale, 16*Upscale, 3].
create_dl_model (DLLayerDepthToSpace, DLModel)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_dl_layer_input, create_dl_layer_concat, create_dl_layer_reshape

Possible Successors

create_dl_layer_convolution, create_dl_layer_dense, create_dl_layer_reshape

See also

create_dl_layer_reshape

Module

Deep Learning Professional

create_dl_layer_dropout (: : DLLayerInput, LayerName, Probability, GenParamName, GenParamValue : DLLayerDropOut)

Create a DropOut layer.

The operator create_dl_layer_dropout creates a DropOut layer with probability Probability and returns the handle DLLayerDropOut.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

During training, activations within DLLayerInput are set to zero with probability Probability. All other activations are rescaled with (1 - Probability).

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_dropout can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer' (DLLayerInput)		x
'name' (LayerName)	x	x
'output_layer' (DLLayerDropOut)		x
'probability' (Probability)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **Probability** (input_control) . . . . . number ⇝ *real*
Probability.
Default: 0.5

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerDropOut** (output_control) . . . . . dl_layer ⇝ *handle*
DropOut layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

```
create_dl_layer_elementwise ( : : DLLayerInputs, LayerName,
    Operation, Coefficients, GenParamName,
    GenParamValue : DLLayerElementWise )
```

Create an elementwise layer.

The operator create_dl_layer_elementwise creates an element-wise layer whose handle is returned in DLLayerElementWise.

An elementwise layer applies a certain operation to every data tensor of the input layers handles and to each element of the data tensor. As a consequence, all input data tensors should be of the same shape and the output tensor has the same shape as the first input tensor.

The parameter DLLayerInputs determines the feeding input layers. This layer expects multiple layers as input. For Operation = *'division'* exactly two input layers are expected.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter Operation specifies the operation that is applied. Depending on Operation, the layer supports implicit broadcasting. I.e., if one of the shape dimensions (batch_size, depth, height, width) of the second or any of the following input tensors is 1, the values are implicitly multiplied along that dimension to match the shape of the first input. The supported values are:

- *'division'*: Element-wise division. Broadcasting is fully supported.
- *'maximum'*: Element-wise maximum. Broadcasting is fully supported.
- *'minimum'*: Element-wise minimum. Broadcasting is fully supported.
- *'product'*: Element-wise product. Broadcasting is supported, but all inputs following the second input must have the same shape as the second input.
- *'sum'*: Element-wise summation. Broadcasting is not supported.

The optional parameter Coefficients determines a weighting coefficient for every input tensor. The number of values in Coefficients must match the number of feeding layers in DLLayerInputs. Set Coefficients equal to *[]* if no coefficients shall be used in the element-wise operation.

Restriction: No coefficients can be set for Operation = *'product'*.

Example: for Operation = *'sum'*, the i-th element of the output data tensor is given by

$$output[i] = \sum_{n=0}^{N-1} \texttt{Coefficients}[n] \cdot \texttt{DLLayerInputs}_n[i],$$

where N is the number of input data tensors.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'div_eps': Small scalar value that is added to the elements of the denominator to avoid a division by zero (for Operation = *'division'*).
Default: *1e-10*

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_elementwise can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'coefficients' (Coefficients)		x
'input_layer' (DLLayerInputs)		x
'name' (LayerName)	x	x
'operation' (Operation)		x
'output_layer' (DLLayerElementWise)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'div_eps'	x	x
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInputs** (input_control) . . . . . dl_layer(-array) ⇝ *handle*
Feeding input layers.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **Operation** (input_control) . . . . . string ⇝ *string*
Element-wise operations.
Default: 'sum'
List of values: `Operation` ∈ {'division', 'maximum', 'minimum', 'product', 'sum'}

▷ **Coefficients** (input_control) . . . . . number(-array) ⇝ *real*
Optional input tensor coefficients.
Default: []

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {'true', 'false'}

▷ **DLLayerElementWise** (output_control) . . . . . dl_layer ⇝ *handle*
Elementwise layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_identity (: : DLLayerInput, LayerName, GenParamName, GenParamValue : DLLayerIdentity)

Create an identity layer.

The operator `create_dl_layer_identity` creates an identity layer whose handle is returned in `DLLayerIdentity`.

The parameter `DLLayerInput` determines the feeding input layer and expects the layer handle as value.

The parameter `LayerName` sets an individual layer name. Note that if creating a model using `create_dl_model` each layer of the created network must have a unique name.

The following generic parameters `GenParamName` and the corresponding values `GenParamValue` are supported:

'is_inference_output': Determines whether `apply_dl_model` will include the output of this layer in the dictionary `DLResultBatch` even without specifying this layer in `Outputs` (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator `create_dl_layer_identity` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer Parameters	set	get
'input_layer' (DLLayerInput)		x
'name' (LayerName)	x	x
'output_layer' (DLLayerIdentity)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerIdentity** (output_control) . . . . . dl_layer ⇝ *handle*
Identity layer.

Example

```
* Create a model that concatinates the output of a convolution layer.
create_dl_layer_input ('input', [10,10,3], [], [], DLLayerInput)
create_dl_layer_convolution (DLLayerInput, 'conv', 3, 1, 1, 8, 1, 'none', \
                             'none', [], [], DLLayerConvolution)
* Using the same layer multiple times as input does not work, so make a copy.
create_dl_layer_identity (DLLayerConvolution, 'conv_copy', [], [], \
                          DLLayerIdentity)
create_dl_layer_concat ([DLLayerConvolution, DLLayerIdentity], 'concat', \
                        'depth', [], [], DLLayerConcat)
create_dl_model (DLLayerConcat, DLModelHandle)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

create_dl_layer_elementwise, create_dl_layer_concat

Module

Deep Learning Professional

create_dl_layer_input (: : LayerName, Shape, GenParamName, GenParamValue : DLLayerInput)

Create an input layer.

The operator create_dl_layer_input creates an input layer with spatial dimensions given by Shape whose handle is returned in DLLayerInput.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

When the created model is applied using e.g., apply_dl_model or train_dl_model_batch, it must be possible to map an input with its corresponding input layer. Operators applying a model expect a feeding dictionary DLSample, see Deep Learning / Model. The mentioned mapping is done using dictionary entries, where the key matches the input layer name. Thus, for an input of this layer a sample dictionary will need an entry with the key LayerName (except if the *'input_type'* is set to *'constant'*, see below).

The parameter Shape defines the shape of the input values (the values given in the feeding dictionary DLSample) and must be a tuple of length three, containing width, height, and depth of the input. The tuple values must be given as integer values and have have different meaning depending on the input type:

- for an input image the layer Shape defines the image size. Images shall be given with type real (for information on image types see Image).
- for an input tuple its length will need to match the product of the individual values in Shape, i.e., width × height × depth.

  Tuple values are distributed along the column- (width), row- (height), and depth-axes in this order. Input tuple values can be given either as integer or real.

The batch size has to be set later with set_dl_model_param, once the model has been created by create_dl_model.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'allow_smaller_tuple': For tuple inputs, setting *'allow_smaller_tuple'* to *'true'* allows to have an input tuple with less values than the total dimension given by Shape. E.g., this can be the case if an input corresponds to the number of objects within one image and the number of objects changes from image to image. If fewer than the maximum number of values given by the total dimension of Shape are present, the remaining values are set to zero.

Shape should be set such that it fits the maximum expected length. For the example above this would be the maximum number of objects within one image present in the whole dataset.

Default: *'false'*.

'const_val': Constant output value.

Restriction:

Only an integer or float is settable. This value is only settable or gettable if *'input_type'* is set to *'constant'*.

Default: *0.0*.

'input_type': Defines the type of input that is expected. The following values are possible:

'default': The layer expects a number of input images corresponding to the batch size.

'region_to_bin': The layer expects a tuple of regions as input and internally converts it to a binary image where each region is encoded in one depth channel. Regions reaching out of the given dimensions are clipped to the width and height given by Shape. The maximum number of regions is defined by the depth of Shape. If fewer than the maximum number of regions are given, the output is filled up with empty (zero) images. For example, this can be the case if the regions are corresponding to objects within an image and the number of objects changes from image to image.

'constant': The layer does not expect any key value pair in the input dictionary. Instead all entries within the output of this layer are filled with the value given by *'const_val'*.

Default: *'default'*.

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).

Default: *'false'*

Certain parameters of layers created using create_dl_layer_input can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer'		x
'name' (LayerName)	x	x
'output_layer' (DLLayerInput)		x
'shape' (Shape)		x
'type'		x

Generic Layer Parameters	set	get
'allow_smaller_tuple'		x
'const_val'	(x)	(x)
'input_type'		x
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **Shape** (input_control) . . . . . number-array ⇝ *integer*
Dimensions of the input (width, height, depth).
Default: [224,224,3]

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'allow_smaller_tuple', 'const_val', 'input_type', 'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {0.0, 'constant', 'default', 'false', 'region_to_bin', 'true'}

▷ **DLLayerInput** (output_control) . . . . . dl_layer ⇝ *handle*
Input layer.

Example

```
* Create a model for summation.
create_dl_layer_input ('input_a', [2, 3, 4], [], [], DLLayerInputA)
create_dl_layer_input ('input_b', [2, 3, 4], [], [], DLLayerInputB)
create_dl_layer_elementwise ([DLLayerInputA, DLLayerInputB], 'sum', \
                             'sum', [], [], [], DLLayerElementWise)
create_dl_model (DLLayerElementWise, DLModel)
set_dl_model_param (DLModel, 'runtime', 'cpu')
*
* Add 'input_a' as an inference model output.
set_dl_model_layer_param (DLModel, 'input_a', 'is_inference_output', 'true')
*
* Feed input data as tuple (a) or image (b).
create_dict (Sample)
set_dict_tuple (Sample, 'input_a', [1:(2*3*4)])
gen_empty_obj (InputB)
for I := 1 to 4 by 1
    gen_image_const (Channel, 'real', 2, 3)
    get_region_points (Channel, Rows, Cols)
    set_grayval (Channel, Rows, Cols, gen_tuple_const(|Rows|, I))
    append_channel (InputB, Channel, InputB)
endfor
set_dict_object (InputB, Sample, 'input_b')
*
```

```
* Apply the model for summation and get results.
set_dl_model_param (DLModel, 'batch_size', 2)
apply_dl_model (DLModel, [Sample,Sample], [], Result)
get_dict_object (Sum, Result[0], 'sum')
get_dict_object (TupleInputA, Result[1], 'input_a')
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

create_dl_layer_activation, create_dl_layer_batch_normalization, create_dl_layer_class_id_conversion, create_dl_layer_class_id_conversion, create_dl_layer_concat, create_dl_layer_convolution, create_dl_layer_dense, create_dl_layer_depth_max, create_dl_layer_dropout, create_dl_layer_elementwise, create_dl_layer_loss_cross_entropy, create_dl_layer_loss_ctc, create_dl_layer_loss_distance, create_dl_layer_loss_focal, create_dl_layer_loss_huber, create_dl_layer_lrn, create_dl_layer_pooling, create_dl_layer_reduce, create_dl_layer_reshape, create_dl_layer_softmax, create_dl_layer_transposed_convolution, create_dl_layer_zoom_factor, create_dl_layer_zoom_size, create_dl_layer_zoom_to_layer_size

Module

Deep Learning Professional

create_dl_layer_loss_cross_entropy (: : DLLayerInput, DLLayerTarget, DLLayerWeights, LayerName, LossWeight, GenParamName, GenParamValue : DLLayerLossCrossEntropy)

Create a cross entropy loss layer.

The operator create_dl_layer_loss_cross_entropy creates a cross entropy loss layer whose handle is returned in DLLayerLossCrossEntropy. This layer computes the two dimensional cross entropy loss on the input (provided by DLLayerInput) given the corresponding target (provided by DLLayerTarget) and weight (provided by DLLayerWeights).

Cross entropy is commonly used to measure the similarity between two vectors.

Example: Illustrative example, where we have a pixel-level classification problem with three classes.

The input vector for a single pixel is $\mathbf{x} = [0.7, 0.1, 0.2]$ (e.g., the output of a softmax layer) which means that the predicted value (e.g., probability) is 0.7 for the class at index 0, 0.1 for the class at index 1 and 0.2 for the class at index 2.

The target vector is $\mathbf{t} = [1.0, 0.0, 0.0]$ with a probability of 1.0 for the actual class and 0.0 else. Entropy is calculated by the dot product of these two vectors. Since the target vector has only one non-zero entry, it can be given by the index of the actual class instead of a vector, in this case $t = 0$.

The cross entropy is then simply the value of the input vector at the target class index, hence $\mathbf{x}[t] = 0.7$. Using this simplification, the cross entropy loss function over an input image can be defined by

$$L_{cross\_entropy}(\mathbf{x}, \mathbf{t}, \mathbf{w}) \quad := \quad -\frac{1}{W} \sum_{i=0}^{N-1} w_i \cdot \mathbf{x}_i[t_i],$$

where the input $\mathbf{x}$ consists of one prediction vector $\mathbf{x}_i$ for each pixel, the target $\mathbf{t}$ and weight $\mathbf{w}$ consist of one value t_i and w_i for each input pixel, N is the number of pixels and $W = \sum_{i=0}^{N-1} w_i$ is the sum over all weights.

Hence, this layer expects multiple incoming layers:

- DLLayerInput: Specifies the prediction (e.g., a softmax layer, commonly with logarithmized results).
- DLLayerTarget: Specifies the target sequences (originating from the ground truth information).
- DLLayerWeights: Specifies the weight sequences. This parameter is optional. If an empty tuple *[]* is passed for all values the weighting factor *1.0* is used.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter LossWeight determines the scalar weight factor with which the loss, calculated in this layer, is multiplied. This parameter can be used to specify the contribution of the cross entropy loss to the overall network loss in case multiple loss layers are used.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_loss_cross_entropy can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer' (DLLayerInput, DLLayerTarget, and/or DLLayerWeights)		x
'loss_weight' (LossWeight)	x	x
'name' (LayerName)	x	x
'output_layer' (DLLayerLossCrossEntropy)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) ..... dl_layer ⇝ *handle*
Input layer.

▷ **DLLayerTarget** (input_control) ..... dl_layer ⇝ *handle*
Target layer.

▷ **DLLayerWeights** (input_control) ..... dl_layer ⇝ *handle*
Weights layer.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **LossWeight** (input_control) ..... number ⇝ *real*
Overall loss weight if there are multiple losses in the network.
Default: 1.0

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerLossCrossEntropy** (output_control) . . . . . dl_layer ⇝ *handle*
Cross entropy loss layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_loss_ctc (: : DLLayerInput, DLLayerInputLengths, DLLayerTarget, DLLayerTargetLengths, LayerName, GenParamName, GenParamValue : DLLayerLossCTC)

Create a CTC loss layer.

The operator create_dl_layer_loss_ctc creates a Connectionist Temporal Classification (CTC) loss layer whose handle is returned in DLLayerLossCTC. See the reference cited below for information about the CTC loss.

With this loss layer it is possible to train sequence to sequence models (Seq2Seq). E.g., it can be used to train a model that is able to read text in an image. In order to do so, the sequences are compared, thus the determined network prediction DLLayerInput with sequence length DLLayerInputLengths to the given DLLayerTarget with sequence length DLLayerTargetLengths.

The following variables are important to understand the input shapes:

- *T:* Maximum input sequence length (i.e., width of DLLayerInput)
- *S:* Maximum output sequence length (i.e., width of DLLayerTarget)
- *C:* Number of classes including 0 as the blank class ID (i.e., depth of DLLayerInput)

This layer expects multiple layers as input:

- DLLayerInput: Specifies the network prediction.
  Shape: [*T*,1,*C*]
- DLLayerInputLengths: Specifies the input sequence length of each item in the batch.
  Shape: [1,1,1]
- DLLayerTarget: Specifies the target sequences.
  Shape: [*S*,1,1]
- DLLayerTargetLengths: Input layer which specifies the target sequence length of each item in the batch.
  Shape: [1,1,1]

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The CTC loss is typically applied in a CNN as follows. The input sequence is expected to be encoded in some CNN layer with the output shape [width: *T*, height: 1, depth: *C*]. Typically the end of a large fully convolutional classifier is pooled in height down to 1 with an average pooling layer. It is important that the last layer is wide enough to hold enough information. In order to obtain the sequence prediction in the output depth a 1x1 convolutional layer is added after the pooling with the number of kernels set to *C*. In this use case the CTC loss obtains this convolutional layer as input layer DLLayerInput. The width of the input layer determines the maximum output sequence of the model.

The CTC loss can be applied to a batch of input items with differing input and target sequence lengths. T and S are the maximum lengths. In DLLayerInputLengths and DLLayerTargetLengths the individual length of each item in a batch needs to be specified.

Restrictions

- A model containing this layer cannot be trained on a CPU.
- A model containing this layer cannot be trained with a *'batch_size_multiplier'* != 1.0.
- The input layer DLLayerInput must not be a softmax layer. The softmax calculation is done internally in this layer. For inference, there should be an extra softmax layer connected to the DLLayerInput (see create_dl_layer_softmax).

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_loss_ctc can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer' (DLLayerInput, DLLayerInputLengths, DLLayerTarget, and/or DLLayerTargetLengths)		x
'name' (LayerName)	x	x
'output_layer' (DLLayerLossCTC)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) ..... dl_layer ⇝ *handle*
Input layer with network predictions.

▷ **DLLayerInputLengths** (input_control) ..... dl_layer ⇝ *handle*
Input layer which specifies the input sequence length of each item in the batch.

▷ **DLLayerTarget** (input_control) ..... dl_layer ⇝ *handle*
Input layer which specifies the target sequences. If the input dimensions of the CNN are changed the width of this layer is automatically resized to the same width as the DLLayerInput layer.

▷ **DLLayerTargetLengths** (input_control) ..... dl_layer ⇝ *handle*
Input layer which specifies the target sequence length of each item in the batch.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerLossCTC** (output_control) ..... dl_layer ⇝ *handle*
CTC loss layer.

Example

```
* Create a simple Seq2Seq model which overfits to a single output sequence.

* Input sequence length
T := 6
* Number of classes including blank (blank is always class_id: 0)
C := 3
* Batch Size
N := 1
* Maximum length of target sequences
S := 3

* Model creation
create_dl_layer_input ('input', [T,1,1], [], [], Input)
create_dl_layer_dense (Input, 'dense', T*C, [], [], DLLayerDense)
create_dl_layer_reshape (DLLayerDense, 'dense_reshape', [T,1,C], [], [],\
                         ConvFinal)

* Training part

* Specify the shapes without batch-size
* (batch-size will be specified in the model).
create_dl_layer_input ('ctc_input_lengths', [1,1,1], [], [],\
                       DLLayerInputLengths)
create_dl_layer_input ('ctc_target', [S,1,1], [], [], DLLayerTarget)
create_dl_layer_input ('ctc_target_lengths', [1,1,1], [], [],\
                       DLLayerTargetLengths)
* Create the loss layer
create_dl_layer_loss_ctc (ConvFinal, DLLayerInputLengths, DLLayerTarget,\
                          DLLayerTargetLengths, 'ctc_loss', [], [],\
                          DLLayerLossCTC)

* Get all names so that users can set values
get_dl_layer_param (ConvFinal, 'name', CTCInputName)
get_dl_layer_param (DLLayerInputLengths, 'name', CTCInputLengthsName)
get_dl_layer_param (DLLayerTarget, 'name', CTCTargetName)
get_dl_layer_param (DLLayerTargetLengths, 'name', CTCTargetLengthsName)

* Inference part
create_dl_layer_softmax (ConvFinal, 'softmax', [], [], DLLayerSoftMax)
create_dl_layer_depth_max (DLLayerSoftMax, 'prediction', 'argmax', [], [],\
                           DLLayerDepthMaxArg, _)

* Setting a seed because the weights of the network are randomly initialized
set_system ('seed_rand', 35)

create_dl_model ([DLLayerLossCTC,DLLayerDepthMaxArg], DLModel)

set_dl_model_param (DLModel, 'batch_size', N)
set_dl_model_param (DLModel, 'runtime', 'gpu')
set_dl_model_param (DLModel, 'learning_rate', 1)
```

```
* Create input sample for training
InputSequence := [0,1,2,3,4,5]
TargetSequence := [1,2,1]
create_dict (InputSample)
set_dict_tuple (InputSample, 'input', InputSequence)
set_dict_tuple (InputSample, 'ctc_input_lengths', |InputSequence|)
set_dict_tuple (InputSample, 'ctc_target', TargetSequence)
set_dict_tuple (InputSample, 'ctc_target_lengths', |TargetSequence|)
Eps := 0.01

PredictedSequence := []
dev_inspect_ctrl ([InputSequence, TargetSequence, CTCLoss, PredictedValues,\
                   PredictedSequence])
MaxIterations:= 15
for I := 0 to MaxIterations by 1
  apply_dl_model (DLModel, InputSample, ['prediction','softmax'], \
                  DLResultBatch)
  get_dict_object (Softmax, DLResultBatch, 'softmax')
  get_dict_object (Prediction, DLResultBatch, 'prediction')
  PredictedValues := []
  for t := 0 to T-1 by 1
      get_grayval (Prediction, 0, t, PredictionValue)
      PredictedValues := [PredictedValues, PredictionValue]
  endfor
  train_dl_model_batch (DLModel, InputSample, DLTrainResult)

  get_dict_tuple (DLTrainResult, 'ctc_loss', CTCLoss)
  if (CTCLoss < Eps)
      break
  endif
  stop()
endfor

* Rudimentary implementation of fastest path prediction
PredictedSequence := []
LastV := -1
for I := 0 to |PredictedValues|-1 by 1
  V := PredictedValues[I]
  if (V == 0)
      LastV := -1
      continue
  endif
  if (|PredictedSequence| > 0 and V == LastV)
      continue
  endif
  PredictedSequence := [PredictedSequence, V]
  LastV :=  PredictedSequence[|PredictedSequence|-1]
endfor
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

References

Graves Alex et al., "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks." Proceedings of the 23rd international conference on Machine learning. 2006.

Module

Deep Learning Professional

create_dl_layer_loss_distance (: : DLLayerInput, DLLayerTarget, DLLayerWeights, LayerName, DistanceType, LossWeight, GenParamName, GenParamValue : DLLayerLossDistance)

Create a distance loss layer.

The operator create_dl_layer_loss_distance creates a distance loss layer whose handle is returned in DLLayerLossDistance.

This layer expects multiple layers as input:

- DLLayerInput: Specifies the prediction (e.g., a softmax layer).
- DLLayerTarget: Specifies the target sequences (originating from the ground truth information).
- DLLayerWeights: Specifies the weight sequences. This parameter is optional. If an empty tuple *[]* is passed for all values the weighting factor *1.0* is used.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter LossWeight is an overall loss weight if there are multiple losses in the network.

The parameter DistanceType determines which distance measure is applied. Currently, *'l2'* and *'l1'* are implemented. Depending on the generic parameter *'reduce'* this results in

- L2 loss distance as a tensor:

$$\begin{aligned}\texttt{DLLayerLossDistance}[i] &= 0.5 \cdot \texttt{LossWeight} \cdot \texttt{DLLayerWeights}[i] \\ &\cdot(\texttt{DLLayerInput}[i] - \texttt{DLLayerTarget}[i])^2,\end{aligned}$$

  in this case loss is the tensor of the same size as DLLayerInput.

- L2 loss distance as a scalar:

$$\texttt{DLLayerLossDistance} = \frac{\sum_{i=0}^{N-1} \texttt{DLLayerLossDistance}[i]}{\sum_{i=0}^{N-1} \texttt{DLLayerWeights}[i]},$$

  where N is a number of elements in DLLayerInput.

- L1 loss distance as a tensor:

$$\begin{aligned}\texttt{DLLayerLossDistance}[i] &= \texttt{LossWeight} \cdot \texttt{DLLayerWeights}[i] \\ &\cdot|\texttt{DLLayerInput}[i] - \texttt{DLLayerTarget}[i]|,\end{aligned}$$

  in this case loss is the tensor of the same size as DLLayerInput.

- L1 loss distance as a scalar:

$$\texttt{DLLayerLossDistance} = \frac{\sum_{i=0}^{N-1} \texttt{DLLayerLossDistance}[i]}{\sum_{i=0}^{N-1} \texttt{DLLayerWeights}[i]},$$

  where N is a number of elements in DLLayerInput.

Thus DLLayerInput, DLLayerTarget and DLLayerWeights should have the same size. Setting the weights in DLLayerWeights to *1* will result in a loss normalized over the number of elements.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether `apply_dl_model` will include the output of this layer in the dictionary `DLResultBatch` even without specifying this layer in `Outputs` (*'true'*) or not (*'false'*).
Default: *'false'*

'reduce': Determines whether the output of the layer is reduced:

- *'true'*: The output is reduced to a scalar.
- *'false'*: The output of the layer is a tensor, where each element is a 'per-pixel' loss (squared differences).

Default: *'true'*.

Certain parameters of layers created using this operator `create_dl_layer_loss_distance` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer Parameters	set	get
'input_layer' (`DLLayerInput`, `DLLayerTarget`, and/or `DLLayerWeights`)		x
'loss_weight' (`LossWeight`)	x	x
'name' (`LayerName`)	x	x
'output_layer' (`DLLayerLossDistance`)		x
'shape'		x
'type'		x
'distance_type' (`DistanceType`)		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x
'reduce'	x	x

Parameters

▷ **DLLayerInput** (input_control) ..... dl_layer ⇝ *handle*
Input layer.

▷ **DLLayerTarget** (input_control) ..... dl_layer ⇝ *handle*
Target layer.

▷ **DLLayerWeights** (input_control) ..... dl_layer ⇝ *handle*
Weights layer.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **DistanceType** (input_control) ..... string ⇝ *string*
Type of distance.
Default: 'l2'
List of values: `DistanceType` ∈ {'l2', 'l1'}

▷ **LossWeight** (input_control) ..... number ⇝ *real*
Loss weight. Applies to all losses, if several losses occur in the network.
Default: 1.0

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'is_inference_output', 'reduce'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *string*
Generic input parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {'true', 'false'}

▷ **DLLayerLossDistance** (output_control) ..... dl_layer ⇝ *handle*
Distance loss layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_loss_focal (: : DLLayerInput, DLLayerTarget, DLLayerWeights, DLLayerNormalization, LayerName, LossWeight, Gamma, ClassWeights, Type, GenParamName, GenParamValue : DLLayerLossFocal)

Create a focal loss layer.

The operator create_dl_layer_loss_focal creates a focal loss layer whose handle is returned in DLLayerLossFocal. See the reference cited below for further information about its definition and parameter meanings.

This layer expects multiple layers as input:

- DLLayerInput: Specifies the prediction (e.g., a sigmoid or softmax layer).
- DLLayerTarget: Specifies the target sequences (originating from the ground truth information).
- DLLayerWeights: Specifies the weight sequences. This parameter is optional. If an empty tuple *[]* is passed for all values the weighting factor *1.0* is used.
- DLLayerNormalization: Specifies the factor to normalize the loss. This parameter is optional, it can be given by the layer handle as value or ignored handing over an empty tuple *[]*.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter LossWeight is a overall loss weight if there are multiple losses in the network.

The parameter Gamma is the exponent of the focal factor.

The parameter ClassWeights defines class specific weights. All loss contributions of foreground samples of a class are weighted with the given factor. The background samples are weighted by 1 - ClassWeights. Typically, this is set to *1.0/(Number of samples of the class)*. Note, the length of this array has to be either 1, then its broadcasted to the number of classes, or it has to correspond to the number of classes. The default value *[]* corresponds to a factor of *0.5* for all classes. Note, if the number of classes are changed on a network then the number of class specific weights are also adapted and reset with the default value *0.5* for each class.

The parameter Type sets the focal loss options:

'focal_binary': Focal loss.
'sigmoid_focal_binary': Focal loss fused with sigmoid.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_loss_focal can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'focal_type' (Type)		x
'gamma' (Gamma)	x	x
'input_layer' (DLLayerInput, DLLayerTarget, DLLayerWeights, and/or DLLayerNormalization)		x
'loss_weight' (LossWeight)	x	x
'name' (LayerName)	x	x
'output_layer' (DLLayerLossFocal)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Input layer.

▷ **DLLayerTarget** (input_control) . . . . . dl_layer ⇝ *handle*
Target layer.

▷ **DLLayerWeights** (input_control) . . . . . dl_layer ⇝ *handle*
Weights layer.

▷ **DLLayerNormalization** (input_control) . . . . . dl_layer ⇝ *handle*
Normalization layer.
Default: []

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **LossWeight** (input_control) . . . . . number ⇝ *real* / integer
Overall loss weight if there are multiple losses in the network.
Default: 1.0

▷ **Gamma** (input_control) . . . . . number ⇝ *real* / integer
Exponent of the focal factor.
Default: 2.0

▷ **ClassWeights** (input_control) . . . . . number(-array) ⇝ *real* / integer
Class specific weight.
Default: []

▷ **Type** (input_control) . . . . . string ⇝ *string*
Focal loss type.
Default: 'focal_binary'
List of values: Type ∈ {'focal_binary', 'sigmoid_focal_binary'}

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string*
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerLossFocal** (output_control) . . . . . dl_layer ⇝ *handle*
Focal loss layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

References

T. Lin, P. Goyal, R. Girshick, K. He and P. Dollar, "Focal Loss for Dense Object Detection," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 2, pp. 318-327, 1 Feb. 2020, doi: 10.1109/TPAMI.2018.2858826.

Module

Deep Learning Professional

create_dl_layer_loss_huber (: : DLLayerInput, DLLayerTarget, DLLayerWeights, DLLayerNormalization, LayerName, LossWeight, Beta, GenParamName, GenParamValue : DLLayerLossHuber)

Create a Huber loss layer.

The operator create_dl_layer_loss_huber creates a Huber loss layer whose handle is returned in DLLayerLossHuber. The Huber loss is defined by

$$\begin{aligned} L_{Huber}(x,t,w,n) &:= \frac{\alpha}{n}\sum_{i=0}^{N-1} w_i l(x_i - t_i), \quad \text{with} \\ l(y) &:= \begin{cases} 0.5y^2/\beta & \text{if } |y| < \beta \\ |y| - 0.5\beta & \text{else.} \end{cases} \end{aligned}$$

This layer expects multiple layers as input:

- DLLayerInput: Specifies x (usually a softmax layer).
- DLLayerTarget: Specifies the targets t.
- DLLayerWeights: Specifies the weights w.

The underlying data tensors are assumed to be of the same shape with a total number of N elements.

The parameter DLLayerNormalization can be used to determine the normalization factor n. If DLLayerNormalization is set to an empty tuple, the sum over all weights is used for the normalization n.

The parameter LossWeight determines the scalar weight factor α.

The parameter Beta sets the value for β in the formula. If Beta is set to *0*, the Huber loss is equal to an L1-loss.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_loss_huber can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'beta' (Beta)	x	x
'input_layer' (DLLayerInput, DLLayerTarget, DLLayerWeights, and/or DLLayerNormalization)		x
'loss_weight' (LossWeight)	x	x
'name' (LayerName)	x	x
'output_layer' (DLLayerLossHuber)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Input layer.

▷ **DLLayerTarget** (input_control) . . . . . dl_layer ⇝ *handle*
Target layer.

▷ **DLLayerWeights** (input_control) . . . . . dl_layer ⇝ *handle*
Weights layer.

▷ **DLLayerNormalization** (input_control) . . . . . dl_layer ⇝ *handle*
Normalization layer.
Default: []

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **LossWeight** (input_control) . . . . . number ⇝ *real*
Scalar weight factor.
Default: 1.0

▷ **Beta** (input_control) . . . . . number ⇝ *real*
Beta value in the loss-defining formula.
Default: 1.1
Restriction: Beta >= 0

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerLossHuber** (output_control) . . . . . dl_layer ⇝ *handle*
Huber loss layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_lrn (: : DLLayerInput, LayerName, LocalSize, Alpha, Beta, K, NormRegion, GenParamName, GenParamValue : DLLayerLRN)

Create a LRN layer.

The operator create_dl_layer_lrn creates a local response normalization layer which performs normalization over a local window and whose handle is returned in DLLayerLRN. Currently, for NormRegion only *'across_channels'* can be set, which results in a normalization across the channel dimension. More detailed, a value x_c located in a channel with index c is normalized with a scale factor depending on a local window,

$$LRN(x_c) = x_c \cdot \left(\mathtt{K} + \frac{\mathtt{Alpha}}{n} \sum_{c'=\max(0,c-n/2)}^{\min(N-1,c+n/2)} x_{c'}^2 \right)^{-\mathtt{Beta}},$$

where n is the size of the local window given by LocalSize, N is the total number of channels, Alpha is the scaling parameter (used as a normalization constant), Beta is the exponent used as a contrast constant, and K is a constant summand, which is used to avoid any singularities.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_lrn can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'alpha' (Alpha)		x
'beta' (Beta)		x
'input_layer' (DLLayerInput)		x
'k' (K)		x
'local_size' (LocalSize)		x
'name' (LayerName)	x	x
'norm_region' (NormRegion)		x
'output_layer' (DLLayerLRN)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **LocalSize** (input_control) . . . . . number ⇝ *integer*
Size of the local window.
Default: 5

▷ **Alpha** (input_control) . . . . . number ⇝ *real*
Scaling factor in the LRN formula.
Default: 0.0001

▷ **Beta** (input_control) . . . . . number ⇝ *real*
Exponent in the LRN formula.
Default: 0.75

▷ **K** (input_control) . . . . . number ⇝ *real*
Constant summand in the LRN formula.
Default: 1.0

▷ **NormRegion** (input_control) . . . . . string ⇝ *string*
Normalization dimension.
Default: 'across_channels'

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerLRN** (output_control) . . . . . dl_layer ⇝ *handle*
LRN layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_matmul (: : DLLayerA, DLLayerB, LayerName, GenParamName, GenParamValue : DLLayerMatMul)

Create a MatMul layer.

The operator create_dl_layer_matmul creates a MatMul layer whose handle is returned in DLLayerMatMul.

A MatMul layer multiplies the 2D matrices, given in the latter two dimensions (H, W) of input DLLayerA, with the corresponding 2D matrices of input DLLayerB, also given in the latter two dimensions (H, W). The output in DLLayerMatMul is hence given by $C = A \cdot B$.

The MatMul layer supports broadcasting for the first input DLLayerA. That means, if the batch size or the number of channels in DLLayerA equals one then the first batch item or channel of DLLayerA is multiplied with all batch items or channels of DLLayerB, respectively.

To make the multiplication work, the width of DLLayerA must be equal to the height of DLLayerB.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether `apply_dl_model` will include the output of this layer in the dictionary `DLResultBatch` even without specifying this layer in `Outputs` (*'true'*) or not (*'false'*).
Default: *'false'*

'num_trainable_params': Number of trainable parameters (weights and biases) of the layer.

'transpose_a': Matrices of input `DLLayerA` are transposed: $C = A^T \cdot B$.
Default: *'false'*

'transpose_b': Matrices of input `DLLayerB` are transposed: $C = A \cdot B^T$.
Default: *'false'*

Certain parameters of layers created using this operator `create_dl_layer_matmul` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer Parameters	set	get
'input_layer'		x
'name' (`LayerName`)	x	x
'output_layer' (`DLLayerMatMul`)		x
'shape'		x
'transpose_a'		x
'transpose_b'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerA** (input_control) . . . . . dl_layer ⇝ *handle*
Input layer A.

▷ **DLLayerB** (input_control) . . . . . dl_layer ⇝ *handle*
Input layer B.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'is_inference_output', 'num_trainable_params', 'transpose_a', 'transpose_b'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {'true', 'false'}

▷ **DLLayerMatMul** (output_control) . . . . . dl_layer ⇝ *handle*
MatMul layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_permutation (: : DLLayerInput, LayerName, Permutation, GenParamName, GenParamValue : DLLayerPermutation)

Create a permutation layer.

The operator create_dl_layer_permutation creates a permutation layer whose handle is returned in DLLayerPermutation.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter Permutation determines the new order of the axes of DLLayerInput, to which the input axes should be permuted.

Permutation has the form [index width, index height, index depth, index batch], where the indices are corresponding to the dimensions of the input. For example, *[0, 1, 3, 2]* leads to swapping the depth and the batch axes. Therefore, each index must be unique and be taken from the set $0, 1, 2, 3$.

Using a CPU device, for some values of Permutation the internal code can not be optimized which can lead to an increased runtime. In this case, the layer parameter *'fall_back_to_baseline'* is set to *'true'*.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_permutation can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'fall_back_to_baseline'		x
'input_layer' (DLLayerInput)		x
'name' (LayerName)	x	x
'permutation' (Permutation)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) ..... dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **Permutation** (input_control) ..... number-array ⇝ *integer*
Order of the permuted axes.
Default: [0,1,2,3]

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerPermutation** (output_control) . . . . . dl_layer ⇝ *handle*
Permutation layer.

Example

```
* Swap the batch and depth axes with a permutation layer.
create_dl_layer_input ('input_a', [1, 1, 4], ['input_type', 'const_val'], \
                       ['constant', 1.0], DLLayerInputA)
create_dl_layer_input ('input_b', [1, 1, 4], ['input_type', 'const_val'], \
                       ['constant', 2.0], DLLayerInputB)
create_dl_layer_concat ([DLLayerInputA, DLLayerInputB], 'concat', 'batch', \
                        [], [], DLLayerConcat)
create_dl_layer_permutation (DLLayerConcat, 'permute', [0,1,3,2], \
                             [], [], DLLayerPermute)
create_dl_layer_depth_max (DLLayerPermute, 'depth_max', 'value', \
                           [], [], _, DLLayerDepthMaxValue)
create_dl_model (DLLayerDepthMaxValue, DLModel)
* The expected output values in DLResultBatch.depth_max are [2.0,2.0,2.0,2.0]
query_available_dl_devices (['runtime'], ['cpu'], DLDeviceHandles)
set_dl_model_param (DLModel, 'device', DLDeviceHandles[0])
apply_dl_model (DLModel, dict{}, [], DLResultBatch)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_dl_layer_input, create_dl_layer_concat, create_dl_layer_reshape

Possible Successors

create_dl_layer_convolution, create_dl_layer_dense, create_dl_layer_reshape

See also

create_dl_layer_reshape

Module

Deep Learning Professional

create_dl_layer_pooling (: : DLLayerInput, LayerName, KernelSize, Stride, Padding, Mode, GenParamName, GenParamValue : DLLayerPooling)

Create a pooling layer.

The operator create_dl_layer_pooling creates a pooling layer whose handle is returned in DLLayerPooling.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter KernelSize specifies the filter kernel in the dimensions width and height.

The parameter Stride specifies how the filter is shifted.

The values for KernelSize and Stride can be set as

- a single value which is used for both dimensions
- a tuple [width, height] and [column, row], respectively.

The parameter Padding determines the padding, thus how many pixels with value 0 are appended on the border of the processed input image. Supported values are:

- *'half_kernel_size'*: The number of appended pixels depends on the specifies KernelSize. More precisely, it is calculated as $\lfloor \texttt{KernelSize}/2 \rfloor$, where for the padding on the left / right border the value of KernelSize in dimension width is regarded and for the padding on the upper / lower border the value of KernelSize in height.
- *'implicit'*: No pixels are appended on the left or on the top of the input image. The number of pixels appended on the right or lower border of the input image is $\texttt{Stride} - (input\_dim - \texttt{KernelSize}) \% \texttt{Stride}$, or zero if the kernel size is a divisor of the input dimension. $input\_dim$ stands for the input width or height.
- *'none'*: No pixels are appended.
- Number of pixels: Specify the number of pixels appended on each border. To do so, the following tuple lengths are supported:
  - Single number: Padding in all four directions left/right/top/bottom.
  - Two numbers: Padding in left/right and top/bottom: [l/r, t/b].
  - Four numbers: Padding on left, right, top, bottom side: [l,r,t,b].
    Restriction: *'runtime' 'gpu'* does not support asymmetric padding, i.e., the padding values for the left and right side must be equal, as well as the padding values for the top and bottom side.

  Restriction: The integer padding values must be smaller than the value set for KernelSize in the corresponding dimension.

The output dimensions of the pooling layer are given by

$$output\_dim = \left\lfloor \frac{input\_dim + padding\_begin + padding\_end - \texttt{KernelSize}}{\texttt{Stride}} \right\rfloor + 1$$

Thereby we use the following values: $output\_dim$: output width, $input\_dim$: input width, $padding\_begin$: number of pixels added to the left/top of the input image, and $padding\_end$: number of pixels added to the right/bottom of the input image.

The parameter Mode specifies the mode of the pooling operation. Supported modes are:

'average': The resulting pixel value is the average of all pixel values in the filter.

'maximum': The resulting pixel value is the maximum of all pixel values in the filter.

'global_average': Same as mode *'average'*, but without the knowledge of the spatial dimensions of the input, it is possible to define the desired output dimensions via the parameter KernelSize. E.g., if the average over all pixel values of the input shall be returned, set the KernelSize to *1* and the output width and height is equal to 1. The internally used kernel size and stride are calculated as follows:

- If KernelSize is a divisor of the input dimensions: The internally used kernel size and stride are both set to the value $input\_dim / \texttt{KernelSize}$.
- If KernelSize is not a divisor of the input dimension: The calculation of the internally used kernel size and stride depend on the generic parameter *'global_pooling_mode'*:

  'overlapping': The internally used $stride$ is set to $\lfloor input\_dim / \texttt{KernelSize} \rfloor$. The internally used kernel size is then computed as $input\_dim - (\texttt{KernelSize} - 1) \cdot stride$. This leads to overlapping kernels but the whole input image is taken into account for the computation of the output.

  'non_overlapping': The internally used kernel size and stride are set to the same value $\lfloor input\_dim / \texttt{KernelSize} \rfloor$. This leads to non-overlapping pooling kernels, but parts of the input image at the right or bottom border might not be considered when computing the output. In this mode, due to rounding the output size is not always equal to the size given by KernelSize.

'adaptive': In this mode, for each pixel (k, l) of the output, the size of the corresponding pooling area within the input is computed adaptively, where k are the row and l are the column indices of the output. The row indices of the pooling area for pixels of the k-th output row are given by $[\lfloor k \cdot input\_dim/\texttt{KernelSize} \rfloor, \lceil (k+1) \cdot input\_dim/\texttt{KernelSize} \rceil)$, where in this case the height of the `KernelSize` is used. The computation of the column coordinates is done analogously. This means that neighboring pooling areas can have a different size which can lead to a less efficient implementation. However, the pooling areas are only overlapping by one pixel which is generally less overlap than for *'global_pooling_mode' 'overlapping'*. The whole input image is taken into account for the computation of the output. For this mode, the parameter `Padding` must be set to *'none'*.

For this mode the parameter `Stride` is ignored and calculated internally as described above.

'global_maximum': Same as mode *'global_average'*, but the maximum is calculated instead of the average.

For more information about the pooling layer see the "`Solution Guide on Classification`".

The following generic parameters `GenParamName` and the corresponding values `GenParamValue` are supported:

'global_pooling_mode': Mode for calculation of the internally used kernel size and stride in case of global pooling (`Mode` *'global_average'* or *'global_maximum'*). See description above. In case of a non-global pooling the parameter is set to the value *'undefined'*.

Default: *'overlapping'*

'is_inference_output': Determines whether `apply_dl_model` will include the output of this layer in the dictionary `DLResultBatch` even without specifying this layer in `Outputs` (*'true'*) or not (*'false'*).

Default: *'false'*

Certain parameters of layers created using this operator `create_dl_layer_pooling` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer Parameters	set	get
'global'		x
'global_pooling_mode'		x
'input_layer' (`DLLayerInput`)		x
'kernel_size' (`KernelSize`)		x
'name' (`LayerName`)	x	x
'output_layer' (`DLLayerPooling`)		x
'padding' (`Padding`)		x
'padding_type' (`Padding`)		x
'pooling_mode' (`Mode`)		x
'shape'		x
'stride' (`Stride`)		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **KernelSize** (input_control) . . . . . number-array ⇝ *integer*
Width and height of the filter kernels.
Default: [2,2]

▷ **Stride** (input_control) . . . . . number-array ⇝ *integer*
Bi-dimensional amount of filter shift.
Default: [2,2]

▷ **Padding** (input_control) . . . . . number(-array) ⇝ *string* / integer
Padding type or specific padding size.
Default: 'none'
Suggested values: Padding ∈ {'none', 'half_kernel_size', 'implicit'}

▷ **Mode** (input_control) . . . . . number ⇝ *string*
Mode of pooling operation.
Default: 'maximum'
List of values: Mode ∈ {'maximum', 'average', 'global_maximum', 'global_average'}

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'global_pooling_mode', 'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'adaptive', 'non_overlapping', 'overlapping', 'true', 'false', 1.0, 0.9, 0.0}

▷ **DLLayerPooling** (output_control) . . . . . dl_layer ⇝ *handle*
Pooling layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_reduce (: : DLLayerInput, LayerName, Operation, Axes, GenParamName, GenParamValue : DLLayerReduce)

Create a reduce layer.

The operator create_dl_layer_reduce creates a reduce layer whose handle is returned in DLLayerReduce.

A reduce layer applies a given operation to the input data tensor to reduce it along one or multiple axes to a single value. Hence, the output tensor has the same shape as the input tensor, but at the axes given by Axes the dimension equals one.

The parameter DLLayerInput determines the feeding input layer. This layer expects a single layer as input.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter Operation specifies the operation that is applied. The operation is applied to the values along the axes given by Axes of the input tensor and the result is written to the corresponding position in the output tensor. The supported values for Operation are:

- *'norm_l2'*: Computes the L2 norm of the input values.
- *'sum'*: Computes the sum of the input values.

The following generic parameters `GenParamName` and the corresponding values `GenParamValue` are supported:

'is_inference_output': Determines whether `apply_dl_model` will include the output of this layer in the dictionary `DLResultBatch` even without specifying this layer in `Outputs` (*'true'*) or not (*'false'*).
Default: *'false'*

'div_eps': Small scalar value that is used to stabilize the training. I.e., in case of a division, the value is added to the denominator to prevent a division by zero.
Default: *1e-10*

Certain parameters of layers created using this operator `create_dl_layer_reduce` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer-Parameter	set	get
'axes' (`Axes`)		x
'input_layer' (`DLLayerInput`)		x
'name' (`LayerName`)	x	x
'operation' (`Operation`)		x
'output_layer' (`DLLayerReduce`)		x
'shape'		x
'type'		x

Generische Layer-Parameter	set	get
'is_inference_output'	x	x
'num_trainable_params'		x
'div_eps'		x

Parameters

▷ **`DLLayerInput`** (input_control) ..... dl_layer ⇝ *handle*
Feeding input layer.

▷ **`LayerName`** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **`Operation`** (input_control) ..... string ⇝ *string*
Reduce operation.
Default: 'norm_l2'
List of values: `Operation` ∈ {'norm_l2', 'sum'}

▷ **`Axes`** (input_control) ..... integer(-array) ⇝ *integer* / string
Axes to which the reduce operation is applied.
Default: [2,3]
List of values: `Axes` ∈ {1, 2, 3, 'width', 'height', 'depth'}

▷ **`GenParamName`** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'div_eps', 'is_inference_output'}

▷ **`GenParamValue`** (input_control) ..... attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {1e-10, 'true', 'false'}

▷ **`DLLayerReduce`** (output_control) ..... dl_layer ⇝ *handle*
Reduce layer.

Example

```
* Minimal example for reduce-layer.
create_dl_layer_input ('input', [64, 32, 10], [], [], DLLayerInput)
create_dl_layer_reduce (DLLayerInput, 'reduce_width', 'sum', 'width', [], [], \
                        DLLayerReduceWidth)
create_dl_layer_reduce (DLLayerReduceWidth, 'reduce_height_depth', 'norm_l2',  [1,2], []
                        [], DLLayerReduceHeightDepth)
* Create a model and change the batch-size.
create_dl_model (DLLayerReduceHeightDepth, DLModel)
set_dl_model_param (DLModel, 'batch_size', 2)
get_dl_model_layer_param (DLModel, 'reduce_height_depth', 'shape', ShapeReduceHeightWidt
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_dl_layer_input`

Possible Successors

`create_dl_model`

Module

Deep Learning Professional

create_dl_layer_reshape (: : DLLayerInput, LayerName, Shape, GenParamName, GenParamValue : DLLayerReshape)

Create a reshape layer.

The operator `create_dl_layer_reshape` creates a reshape layer whose handle is returned in `DLLayerReshape`.

The parameter `DLLayerInput` determines the feeding input layer and expects the layer handle as value.

The parameter `LayerName` sets an individual layer name. Note that if creating a model using `create_dl_model` each layer of the created network must have a unique name.

The parameter `Shape` determines the output shape, into which the input data is converted.

The value of `Shape` has to be given in the form [`width`, `height`, `depth`, `batch_size`], where the fourth value for the batch size is optional (see below). The overall size of the data has to remain constant, i.e., `width_out * height_out * depth_out * batch_size_out = width_in * height_in * depth_in * batch_size_in`.

The following options are available for setting the values of `Shape`:

- Setting a value for each of the four dimensions,
- One or several values are set to *0* in order to keep the value of the input dimension,
- By setting a maximum of one value to *-1*, this value will be determined automatically. It will be calculated in a way that the overall size remains constant. Note that this is only possible if the computed value is an integer.

For a model that was created using `create_dl_model` the model's batch size should always be settable with `set_dl_model_param`. Hence, either the output batch size of the reshape layer equals the batch size of the model (batch size in `Shape` set to *0*), or at least one reshape dimension should be calculated automatically (one value in `Shape` set to *-1*).

If the batch size is specified and it is not set to *0*, at least one dimension of Shape must be set to *-1*. This is necessary, because for a model created with create_dl_model, the model's batch size should always be settable with set_dl_model_param. Hence, either the output batch size of the reshape layer equals the batch size of the model (batch size in Shape set to *0*), or at least one reshape dimension should be calculated automatically (one value in Shape set to *-1*). In case the batch size is not specified it is set to *0*, which leads to an output batch size equal to the input one.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_reshape can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'input_layer' (DLLayerInput)		x
'name' (LayerName)	x	x
'output_depth' (Shape)		x
'output_height' (Shape)		x
'output_layer' (Shape)		x
'output_width' (Shape)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **Shape** (input_control) . . . . . number-array ⇝ *integer*
Shape of the output graph layer data.
Default: [224,224,3]

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerReshape** (output_control) . . . . . dl_layer ⇝ *handle*
Reshape layer.

Example

```
* Minimal example for reshape-layer.
create_dl_layer_input ('input', [64, 32, 10], [], [], DLLayerInput)
create_dl_layer_reshape (DLLayerInput, 'reshape_wh', [32, 64, 0], [], [], \
                         DLLayerReshapeWH)
create_dl_layer_reshape (DLLayerInput, 'reshape_bs', [64, 32, 1, -1], [], \
                         [], DLLayerReshapeBS)
* DLLayerReshapeBS has batch size 10 and depth 1.
get_dl_layer_param (DLLayerReshapeBS, 'shape', ShapeReshapeBS)
* Create a model and change the batch-size.
create_dl_model (DLLayerReshapeBS, DLModel)
set_dl_model_param (DLModel, 'batch_size', 2)
* DLLayerReshapeBS has batch size 20 now.
get_dl_model_layer_param (DLModel, 'reshape_bs', 'shape', ShapeReshapeBS)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_dl_layer_input, create_dl_layer_concat

Possible Successors

create_dl_layer_convolution, create_dl_layer_dense

Module

Deep Learning Professional

create_dl_layer_softmax (: : DLLayerInput, LayerName, GenParamName, GenParamValue : DLLayerSoftMax)

Create a softmax layer.

The operator create_dl_layer_softmax creates a softmax layer whose handle is returned in DLLayerSoftMax.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The softmax layer applies the softmax function which is defined for each input x_i as follows:

$$Softmax(x_i) = \frac{\exp(x_i)}{\sum_{j=0}^{N-1} \exp(x_j)}$$

where N is the number of inputs. During training, the result of the softmax function is transformed by a logarithm function, such that the values are suitable as input to e.g., a cross entropy loss layer. This behavior can be changed by setting the generic parameter *'output_mode'*, see below.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'output_mode': This parameter determines if and in which case the output is transformed by a logarithm function:

- *'default'*: During inference, the result of the softmax function is returned as output while during training, the softmax is further transformed by a logarithm function.
- *'no_log_training'*: During training the result of the softmax function is not transformed by a logarithm function.

- *'log_inference'*: The logarithm of the softmax is calculated during inference in the same way as during training.

Default: *'default'*.

'is_inference_output': Determines whether `apply_dl_model` will include the output of this layer in the dictionary `DLResultBatch` even without specifying this layer in `Outputs` (*'true'*) or not (*'false'*).

Default: *'false'*

Certain parameters of layers created using this operator `create_dl_layer_softmax` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer Parameters	set	get
'input_layer' (`DLLayerInput`)		x
'name' (`LayerName`)	x	x
'output_layer' (`DLLayerSoftMax`)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x
'output_mode'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'output_mode', 'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {'default', 'no_log_training', 'log_inference', 'true', 'false'}

▷ **DLLayerSoftMax** (output_control) . . . . . dl_layer ⇝ *handle*
Softmax layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_transposed_convolution (: : DLLayerInput, LayerName, KernelSize, Stride, KernelDepth, Groups, Padding, GenParamName, GenParamValue : DLLayerTransposedConvolution)

Create a transposed convolution layer.

The operator create_dl_layer_transposed_convolution creates a transposed convolution layer whose handle is returned in DLLayerTransposedConvolution.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter KernelSize specifies the filter kernel in the dimensions width and height. So far, only quadratic kernels are supported.

Restriction: This value must be a tuple of length 1.

The parameter Stride determines how the filter is shifted in row and column direction.

Restriction: This value must be a tuple of length 1.

The parameter KernelDepth defines the depth of the output feature maps.

Restriction: This value must be a tuple of length 1.

The parameter Groups determines the amount of filter groups. So far, only a single filter group is supported.

Restriction: This value must be a tuple of length 1.

The parameter Padding effectively appends $\texttt{KernelSize} - 1 - \texttt{Padding}$ pixels with value 0 to each border of the input. This is set so that a convolutional layer and a transposed convolution layer with the same KernelSize, Stride and Padding values are inverses of each other regarding their input and output shapes. Supported Padding values are:

- *'half_kernel_size'*: The integer value of Padding in the formula above depends on the specified KernelSize. More precisely, it is calculated as $\lfloor \texttt{KernelSize}/2 \rfloor$.
- *'none'*: The value of Padding in the formula above is 0.
- Number of pixels: Specify the integer value of Padding in the formula above for each border. To do so, the following tuple lengths are supported:
  - Single number: Padding value for all four directions left/right/top/bottom.
  - Two numbers: Padding value for left/right and top/bottom: [l/r, t/b].
  - Four numbers: Padding value for left, right, top, bottom side: [l,r,t,b].
    Restriction: *'runtime' 'gpu'* does not support asymmetric padding, i.e., the padding values for the left and right side must be equal, as well as the padding values for the top and bottom side.

  Restriction: The integer padding values must be smaller than the value set for KernelSize.

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'bias_term': Determines, whether the layer has bias terms.
Default: *'true'*

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

'learning_rate_multiplier': Learning rate multiplier for this layer that is used during training. If *'learning_rate_multiplier'* is set to *0.0*, the layer is skipped during training.
Default: *1.0*

'output_padding': Can be used to resolve ambiguities in the output shape for KernelSize and Stride larger than *1*. As mentioned above in the description of the parameter Padding, with respect to the input and output shapes a transposed convolution layer can be seen as the inverse of a convolution layer if they have the same settings for KernelSize, Stride, and Padding (and dilation). However, a convolution layer can map several shapes to the same output spatial dimensions. E.g., for KernelSize *3*, Padding *1*, and Stride *2*, both inputs with spatial dimensions (H, W) = *'(4, 4)'* and (H, W) = *'(3, 3)'* are mapped to an output shape with (H, W) = *'(2, 2)'*. To get back to *'(4, 4)'* using a transposed convolution (with the same settings for KernelSize, Stride, and Padding) *'output_padding'* must be set to *1*. *'output_padding'* must not be larger than the bottom and right Padding and can only be set larger than *0* if Stride is larger than *1*.
Default: *0*

'weight_filler': Defines the mode how the weights are initialized. See create_dl_layer_convolution for a detailed explanation of this parameter and its values.
List of values: *'xavier'*, *'msra'*, *'const'*
Default: *'xavier'*

'weight_filler_const_val': See create_dl_layer_convolution for a detailed explanation of this parameter and its values.
Default: *0.5*

'weight_filler_variance_norm': Value range for *'weight_filler'*. See create_dl_layer_convolution for a detailed explanation of this parameter and its values.
List of values: *'norm_average'*, *'norm_in'*, *'norm_out'*, constant value (in combination with *'weight_filler'* = *'msra'*)
Default: *'norm_in'*

Certain parameters of layers created using this operator create_dl_layer_transposed_convolution can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'groups' (Groups)		x
'input_depth'		x
'input_layer' (DLLayerInput)		x
'kernel_depth' (KernelDepth)		x
'kernel_size' (KernelSize)		x
'name' (LayerName)	x	x
'output_layer' (DLLayerTransposedConvolution)		x
'padding_type' (Padding)		x
'shape'		x
'stride' (Stride)		x
'type'		x

Generic Layer Parameters	set	get
'bias_term'		x
'is_inference_output'	x	x
'learning_rate_multiplier'	x	x
'num_trainable_params'		x
'output_padding'		x
'weight_filler'	x	x
'weight_filler_const_val'	x	x
'weight_filler_variance_norm'	x	x

Parameters

▷ **DLLayerInput** (input_control) ..... dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **KernelSize** (input_control) ..... number ⇝ *integer*
Width and height of the filter kernels.
Default: 3

▷ **Stride** (input_control) ..... number ⇝ *integer*
Amount of filter shift.
Default: 1

▷ **KernelDepth** (input_control) ..... number ⇝ *integer*
Depth of filter kernels.
Default: 64

▷ **Groups** (input_control) ..... number ⇝ *integer*
Number of filter groups.
Default: 1

▷ **Padding** (input_control) ..... number(-array) ⇝ *string* / integer
Type of the padding.
Default: 'none'
List of values: `Padding` ∈ {'none', 'half_kernel_size', [all], [width,height], [left,right,top,bottom]}
Suggested values: `Padding` ∈ {'none', 'half_kernel_size'}

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'bias_term', 'is_inference_output', 'learning_rate_multiplier', 'output_padding', 'weight_filler', 'weight_filler_const_val', 'weight_filler_variance_norm'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {'xavier', 'msra', 'const', 'norm_in', 'norm_out', 'norm_average', 'true', 'false'}

▷ **DLLayerTransposedConvolution** (output_control) ..... dl_layer ⇝ *handle*
Transposed convolutional layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

create_dl_layer_zoom_factor (: : DLLayerInput, LayerName, ScaleWidth, ScaleHeight, Interpolation, AlignCorners, GenParamName, GenParamValue : DLLayerZoom)

Create a zoom layer using size factors.

The operator `create_dl_layer_zoom_factor` creates a zoom layer using size factors and returns the layer handle in `DLLayerZoom`.

The parameter `DLLayerInput` determines the feeding input layer and expects the layer handle as value.

The parameter `LayerName` sets an individual layer name. Note that if creating a model using `create_dl_model` each layer of the created network must have a unique name.

The parameters `ScaleWidth` and `ScaleHeight` specify the ratio between the output and the corresponding input dimension. Together they define the output size of the zoom layer `DLLayerZoom`.

The parameter `Interpolation` defines the interpolation mode. Currently only the mode *'bilinear'* is supported.

The parameter `AlignCorners` defines how coordinates are transformed from the output to the input image:

'true': The transformation is applied in the HALCON Non-Standard Cartesian coordinate system (edge-centered, with the origin in the upper left corner, see chapter Transformations / 2D Transformations). Using the x axis as an example, this leads to:

$$x_{input} = x_{output} * (length_{input} - 1)/(length_{output} - 1)$$

'false': The transformation is applied in the HALCON standard coordinate system (pixel centered, with the origin in the center of the upper left pixel, see chapter Transformations / 2D Transformations). Using the x axis as an example, this leads to:

$$x_{input} = (x_{output} + 0.5) * length_{input}/length_{output} - 0.5$$

The following generic parameters `GenParamName` and the corresponding values `GenParamValue` are supported:

'is_inference_output': Determines whether `apply_dl_model` will include the output of this layer in the dictionary `DLResultBatch` even without specifying this layer in `Outputs` (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator `create_dl_layer_zoom_factor` can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using `set_dl_model_layer_param` and which ones can be retrieved using `get_dl_model_layer_param` or `get_dl_layer_param`. Note, the operators `set_dl_model_layer_param` and `get_dl_model_layer_param` require a model created by `create_dl_model`.

Layer Parameters	set	get
'align_corners' (`AlignCorners`)	x	x
'input_layer' (`DLLayerInput`)		x
'interpolation_mode' (`Interpolation`)		x
'name' (`LayerName`)	x	x
'output_layer' (`DLLayerZoom`)		x
'scale_params' (`ScaleWidth` and `ScaleHeight`)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **`DLLayerInput`** (input_control) ..... dl_layer ⇝ *handle*
Feeding layer.

▷ **`LayerName`** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **`ScaleWidth`** (input_control) ..... number ⇝ *real* / integer
Ratio output/input width of the layer.
Default: 2.0

▷ **`ScaleHeight`** (input_control) ..... number ⇝ *real* / integer
Ratio output/input height of the layer.
Default: 2.0

▷ **`Interpolation`** (input_control) ..... string ⇝ *string*
Mode of interpolation.
Default: 'bilinear'
List of values: `Interpolation` ∈ {'bilinear'}

▷ **`AlignCorners`** (input_control) ..... string ⇝ *string*
Type of coordinate transformation between output/input images.
Default: 'false'
List of values: `AlignCorners` ∈ {'true', 'false'}

▷ **`GenParamName`** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerZoom** (output_control) . . . . . dl_layer ⇝ *handle*
Zoom layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

create_dl_layer_zoom_size, create_dl_layer_zoom_to_layer_size

Module

Deep Learning Professional

create_dl_layer_zoom_size (: : DLLayerInput, LayerName, Width, Height, Interpolation, AlignCorners, GenParamName, GenParamValue : DLLayerZoom)

Create a zoom layer using an absolute output size.

The operator create_dl_layer_zoom_size creates a zoom layer using an absolute output size and returns the layer handle in DLLayerZoom.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameters Width and Height define the absolute output size of the zoom layer DLLayerZoom.

The parameter Interpolation defines the interpolation mode. Currently only the mode *'bilinear'* is supported.

The parameter AlignCorners defines how coordinates are transformed from the output to the input image:

'true': The transformation is applied in the HALCON Non-Standard Cartesian coordinate system (edge-centered, with the origin in the upper left corner, see chapter Transformations / 2D Transformations). Using the x axis as an example, this leads to:

$$x_{input} = x_{output} * (length_{input} - 1)/(length_{output} - 1)$$

'false': The transformation is applied in the HALCON standard coordinate system (pixel centered, with the origin in the center of the upper left pixel, see chapter Transformations / 2D Transformations). Using the x axis as an example, this leads to:

$$x_{input} = (x_{output} + 0.5) * length_{input}/length_{output} - 0.5$$

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_zoom_size can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'align_corners' (AlignCorners)	x	x
'input_layer' (DLLayerInput)		x
'interpolation_mode' (Interpolation)		x
'name' (LayerName)	x	x
'output_layer' (DLLayerZoom)		x
'scale_params' (Width and Height)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) ..... dl_layer ⇝ *handle*
Feeding layer.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **Width** (input_control) ..... number ⇝ *integer*
Absolute width of the output layer.
Default: 100

▷ **Height** (input_control) ..... number ⇝ *integer*
Absolute height of the output layer.
Default: 100

▷ **Interpolation** (input_control) ..... string ⇝ *string*
Mode of interpolation.
Default: 'bilinear'
List of values: Interpolation ∈ {'bilinear'}

▷ **AlignCorners** (input_control) ..... string ⇝ *string*
Type of coordinate transformation between output/input images.
Default: 'false'
List of values: AlignCorners ∈ {'true', 'false'}

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: GenParamName ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **DLLayerZoom** (output_control) ..... dl_layer ⇝ *handle*
Zoom layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

create_dl_layer_zoom_factor, create_dl_layer_zoom_to_layer_size

Module

Deep Learning Professional

create_dl_layer_zoom_to_layer_size (: : DLLayerInput, DLLayerReference, LayerName, Interpolation, AlignCorners, GenParamName, GenParamValue : DLLayerZoom)

Create a zoom layer using the output size of a reference layer.

The operator create_dl_layer_zoom_to_layer_size creates a zoom layer using the output size of a reference layer and returns the layer handle in DLLayerZoom.

The parameter DLLayerInput determines the feeding input layer and expects the layer handle as value.

The parameter DLLayerReference is used to define the output size of the zoom layer DLLayerZoom: the size is adapted to the output size of DLLayerReference.

The parameter LayerName sets an individual layer name. Note that if creating a model using create_dl_model each layer of the created network must have a unique name.

The parameter Interpolation defines the interpolation mode. Currently only the mode *'bilinear'* is supported.

The parameter AlignCorners defines how coordinates are transformed from the output to the input image:

'true': The transformation is applied in the HALCON Non-Standard Cartesian coordinate system (edge-centered, with the origin in the upper left corner, see chapter Transformations / 2D Transformations). Using the x axis as an example, this leads to:

$$x_{input} = x_{output} * (length_{input} - 1)/(length_{output} - 1)$$

'false': The transformation is applied in the HALCON standard coordinate system (pixel centered, with the origin in the center of the upper left pixel, see chapter Transformations / 2D Transformations). Using the x axis as an example, this leads to:

$$x_{input} = (x_{output} + 0.5) * length_{input}/length_{output} - 0.5$$

The following generic parameters GenParamName and the corresponding values GenParamValue are supported:

'is_inference_output': Determines whether apply_dl_model will include the output of this layer in the dictionary DLResultBatch even without specifying this layer in Outputs (*'true'*) or not (*'false'*).
Default: *'false'*

Certain parameters of layers created using this operator create_dl_layer_zoom_to_layer_size can be set and retrieved using further operators. The following tables give an overview, which parameters can be set using set_dl_model_layer_param and which ones can be retrieved using get_dl_model_layer_param or get_dl_layer_param. Note, the operators set_dl_model_layer_param and get_dl_model_layer_param require a model created by create_dl_model.

Layer Parameters	set	get
'align_corners' (AlignCorners)	x	x
'input_layer' (DLLayerInput)		x
'interpolation_mode' (Interpolation)		x
'name' (LayerName)	x	x
'output_layer' (DLLayerZoom)		x
'scale_params' (DLLayerReference)		x
'shape'		x
'type'		x

Generic Layer Parameters	set	get
'is_inference_output'	x	x
'num_trainable_params'		x

Parameters

▷ **DLLayerInput** (input_control) . . . . . dl_layer ⇝ *handle*
Feeding layer.

▷ **DLLayerReference** (input_control) . . . . . dl_layer ⇝ *handle*
Reference layer to define the output size.

▷ **LayerName** (input_control) . . . . . string ⇝ *string*
Name of the output layer.

▷ **Interpolation** (input_control) . . . . . string ⇝ *string*
Mode of interpolation.
Default: 'bilinear'
List of values: `Interpolation` ∈ {'bilinear'}

▷ **AlignCorners** (input_control) . . . . . string ⇝ *string*
Type of coordinate transformation between output/input images.
Default: 'false'
List of values: `AlignCorners` ∈ {'true', 'false'}

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Generic input parameter names.
Default: []
List of values: `GenParamName` ∈ {'is_inference_output'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Generic input parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {'true', 'false'}

▷ **DLLayerZoom** (output_control) . . . . . dl_layer ⇝ *handle*
Zoom layer.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`create_dl_layer_zoom_size`, `create_dl_layer_zoom_factor`

Module

Deep Learning Professional

create_dl_model (: : OutputLayers : DLModelHandle)

Create a deep learning model.

The operator `create_dl_model` creates a deep learning model from a graph and returns its handle in `DLModelHandle`.

A deep learning model in HALCON mainly consists of a directed acyclic graph that defines the networks architecture. Further components of a deep learning model in HALCON are parameters as *'class_names'*, *'class_ids'*, and many others, or hyperparameters that are needed to train a model, as for example the *'learning_rate'*. While parameters and hyperparameters can be set after creation of the model using `set_dl_model_param`, the model itself can only be created using `create_dl_model` if its network architecture is given in form of a graph.

To build a graph that defines the models network architecture, one needs to put together the networks layers. In general, a graph starts with an input layer. A subsequent layer that follows after the input layer uses the input layer as feeding layer, and the new layer itself might be used as a feeding layer for the next layer, and so on. This is repeated until the graphs output layers (e.g., softmax or loss layers) are appended to the graph. To create a layer, use its specified creation operator, e.g., an input layer is created using `create_dl_layer_input`, a convolution layer is created using `create_dl_layer_convolution`, and so on.

When the graph is defined, a model can be created using `create_dl_model` by passing over the graphs output layer handles in `OutputLayers`. Note that the output layer handles save all other layers that directly or indirectly serve as feeding input layers for the output layers during their creation. This means that the output layer handles keep the whole network architecture necessary for the creation of the model using `create_dl_model`.

The type of the created model, hence the task the model is designed for (classification, object detection, segmentation), is only given by the networks architecture. However, if the networks architecture allows it, the type of the model, *'type'*, can be set using `set_dl_model_param`. A specified model type allows a more user friendly usage in the HALCON deep learning workflow. Supported types are:

'generic': This is the default model type. The task the model's neuronal network can solve is defined by its architecture. When `apply_dl_model` is applied for inference, the operator returns the activations of the output layers. To train the model using `train_dl_model_batch`, the underlying graph requires loss layers.

'classification': The model is specified for classification and all layers required for training the model are adapted to the model. When `apply_dl_model` is applied for inference, the output is adapted according to the type, see `apply_dl_model` for more details. See Deep Learning / Classification for further information.

In addition, the operator `gen_dl_model_heatmap` can be used to display the models heatmap.

'detection': The model is specified for object detection and instance segmentation and all layers and anchors required for training the model are adapted to the model. When `apply_dl_model` is applied for inference, the output is adapted according to the type, see `apply_dl_model` for more details. See Deep Learning / Object Detection and Instance Segmentation for further information.

'multi_label_classification': The model is specified for multi-label classification and all layers required for training the model are adapted to the model. When `apply_dl_model` is applied for inference, the output is adapted according to the type, see `apply_dl_model` for more details. See Deep Learning / Multi-Label Classification for further information.

'segmentation': The model is specified for semantic segmentation or edge extraction respectively and all layers required for training the model are adapted to the model. When `apply_dl_model` is applied for inference, the output is adapted according to the type, see `apply_dl_model` for more details. See Deep Learning / Semantic Segmentation and Edge Extraction for further information.

Furthermore, many deep learning procedures provide more functionality for the model if its type is set. As an example, `dev_display_dl_data` can be used to display the inferred results more nicely.

Note that setting a model type requires that the graph fulfills certain structure conditions. We recommend to follow the architecture of our delivered neuronal networks if the model type should be set to one of these types.

Parameters

▷ **`OutputLayers`** (input_control) ..... dl_layer(-array) ⇝ *handle*
Output layers of the graph.

▷ **`DLModelHandle`** (output_control) ..... dl_model ⇝ *handle*
Handle of the deep learning model.

Result

If the parameters are valid, the operator `create_dl_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`create_dl_layer_softmax`, `create_dl_layer_loss_cross_entropy`, `create_dl_layer_loss_focal`, `create_dl_layer_loss_huber`

Possible Successors

`set_dl_model_param`

Module

Deep Learning Professional

get_dl_layer_param (: : DLLayer, GenParamName : GenParamValue)

Return the parameters of a deep learning layer.

The operator get_dl_layer_param returns the parameter GenParamName of the deep learning layer DLLayer in GenParamValue.

Depending on the type of the layer, different parameter names are valid. Which generic and layer-specific parameters can be queried is described in the specific references of the operators used for layer creation (create_dl_layer_*).

Parameters

▷ **DLLayer** (input_control) . . . . . dl_layer ⇝ *handle*
Layer.

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Parameter to query.
Default: 'shape'
List of values: GenParamName ∈ {'input_layer', 'name', 'shape', 'type'}

▷ **GenParamValue** (output_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string / handle
Value of the queried parameter.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

get_dl_model_layer (: : DLModelHandle, LayerNames : DLLayers)

Create a deep copy of the layers and all of their graph ancestors in a given deep learning model.

The operator get_dl_model_layer creates a deep copy of every layer named in LayerNames and all their graph ancestors in the deep learning model DLModelHandle. You can retrieve the unique layer names using get_dl_model_param with its option *'summary'*.

You might use the output layers returned in DLLayers as inputs to the create_dl_layer_* and create_dl_model operators in order to create novel model architectures based on existing models.

If you want to get multiple layers of a single model, these layers have to be specified as a LayerNames tuple in a single call to get_dl_model_layer. Doing so, you avoid multiple deep copies of graph ancestors that are potentially shared by the layers.

Example:

```
get_dl_model_layer(DLModelHandleOrig, ['layer_name_3',
'layer_name_6'], DLLayersOutput)
create_dl_model([DLLayersOutput], DLModelHandle)
```

Please note, that the output layers are copies. They contain the same weights and settings as in the given input model but they are unique copies. You cannot alter the existing model by changing the output layers.

Parameters

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Deep learning model.

▷ **LayerNames** (input_control) ..... string(-array) ⇝ *string*
Names of the layers to be copied.

▷ **DLLayers** (output_control) ..... dl_layer(-array) ⇝ *handle*
Copies of layers and all of their ancestors.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_dl_model

Possible Successors

create_dl_model, create_dl_layer_activation, create_dl_layer_batch_normalization, create_dl_layer_class_id_conversion, create_dl_layer_class_id_conversion, create_dl_layer_concat, create_dl_layer_convolution, create_dl_layer_dense, create_dl_layer_depth_max, create_dl_layer_dropout, create_dl_layer_elementwise, create_dl_layer_loss_cross_entropy, create_dl_layer_loss_ctc, create_dl_layer_loss_distance, create_dl_layer_loss_focal, create_dl_layer_loss_huber, create_dl_layer_lrn, create_dl_layer_pooling, create_dl_layer_reduce, create_dl_layer_reshape, create_dl_layer_softmax, create_dl_layer_transposed_convolution, create_dl_layer_zoom_factor, create_dl_layer_zoom_size, create_dl_layer_zoom_to_layer_size

Module

Deep Learning Professional

get_dl_model_layer_activations (: Activations : DLModelHandle, LayerName :)

Get the activations of a Deep Learning model layer.

The operator get_dl_model_layer_activations returns in Activations the activations of the specified LayerName of the model DLModelHandle.

Activations is a tuple of batch_size many objects, where every object is an image having the size (width, height, depth) of the given LayerName.

Parameters

▷ **Activations** (output_object) ..... image(-array) ⇝ *object :* real
Output activations.

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Handle of the deep learning model.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the layer to be queried.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Module

Deep Learning Professional

get_dl_model_layer_gradients (: Gradients : DLModelHandle, LayerName :)

Get the gradients of a Deep Learning model layer.

The operator get_dl_model_layer_gradients returns in Gradients the gradients of the specified LayerName of the model DLModelHandle.

Gradients is a tuple of batch_size many objects, where every object is an image having the size (width, height, depth) of the given LayerName.

Parameters

▷ **Gradients** (output_object) ..... image(-array) ⇝ *object :* real
Output gradients.

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Handle of the deep learning model.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the layer to be queried.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

get_dl_model_layer_param (: : DLModelHandle, LayerName, ParamName : ParamValue)

Retrieve parameter values for a given layer.

The operator get_dl_model_layer_param returns for a layer the value of the parameter ParamName in ParamValue. The layer is referred by its name LayerName in the model DLModelHandle. You can retrieve the layer names using get_dl_model_param with its option *'layer_names'* or *'summary'*.

Which generic and layer-specific parameters can be queried is described in the specific operator references (create_dl_layer_*).

Parameters

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Deep learning model.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **ParamName** (input_control) ..... string ⇝ *string*
Name of the queried parameter.
Default: 'type'
List of values: ParamName ∈ {'bias_filler', 'bias_filler_variance_norm', 'bias_filler_const_val', 'bias_term', 'input_layer', 'is_inference_output', 'leaky_relu_alpha', 'learning_rate_multiplier', 'learning_rate_multiplier_bias', 'name', 'num_trainable_params', 'output_layer', 'shape', 'type', 'upper_bound', 'weight_filler', 'weight_filler_const_val', 'weight_filler_variance_norm'}
Restriction: length(ParamName) > 0

▷ **ParamValue** (output_control) . . . . . tuple(-array) ⇝ *string* / real / integer
Value of the queried parameter.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

get_dl_model_param, set_dl_model_layer_param

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Deep Learning Professional

get_dl_model_layer_weights (: Weights : DLModelHandle, LayerName, WeightsType :)

Get the weights (or values) of a Deep Learning model layer.

The operator get_dl_model_layer_weights returns in Weights the values of a LayerName of the model DLModelHandle.

The parameter WeightsType determines which type of layer values are retrieved. The following values are supported for WeightsType:

- *'batchnorm_mean'*: Batch-wise calculated mean values to normalize the inputs. For further information, please refer to create_dl_layer_batch_normalization.
  Restriction: This value is only supported if the layer is of type *'batchnorm'*.
- *'batchnorm_mean_avg'*: Average of the batch-wise calculated mean values to normalize the inputs. For further information, please refer to create_dl_layer_batch_normalization.
  Restriction: This value is only supported if the layer is of type *'batchnorm'*.
- *'batchnorm_variance'*: Batch-wise calculated variance values to normalize the inputs. For further information, please refer to create_dl_layer_batch_normalization.
  Restriction: This value is only supported if the layer is of type *'batchnorm'*.
- *'batchnorm_variance_avg'*: Average of the batch-wise calculated variance values to normalize the inputs. For further information, please refer to create_dl_layer_batch_normalization.
  Restriction: This value is only supported if the layer is of type *'batchnorm'*.
- *'bias'*: Biases of the layer.
- *'bias_gradient'*: Gradients of the biases of the layer.
- *'bias_gradient_norm_l2'*: Gradients of the biases of the layer in terms of L2 norm.
- *'bias_norm_l2'*: Biases of the layer in terms of L2 norm.
- *'bias_update'*: Update of the biases of the layer. This is used in e.g., a solver which uses the last update.
- *'bias_update_norm_l2'*: Update of the biases of the layer in terms of L2 norm. This is used in a solver which uses the last update.
- *'weights'*: Weights of the layer.
- *'weights_gradient'*: Gradients of the weights of the layer.
- *'weights_gradient_norm_l2'*: Gradients of the weights of the layer in terms of L2 norm.
- *'weights_norm_l2'*: Weights of the layer in terms of L2 norm.
- *'weights_update'*: Update of the weights of the layer. This is used in a solver which uses the last update.
- *'weights_update_norm_l2'*: Update of the weights of the layer in terms of L2 norm. This is used in a solver which uses the last update.

The following tables give an overview, which parameters for WeightsType can be set using set_dl_model_layer_weights and which ones can be retrieved using get_dl_model_layer_weights.

Layer Parameters	set	get
'batchnorm_mean'	x	x
'batchnorm_mean_avg'	x	x
'batchnorm_variance'	x	x
'batchnorm_variance_avg'	x	x
'bias'	x	x
'bias_gradient'		x
'bias_gradient_norm_l2'		x
'bias_norm_l2'		x
'bias_update'		x
'bias_update_norm_l2'		x
'weights'	x	x
'weights_gradient'		x
'weights_gradient_norm_l2'		x
'weights_norm_l2'		x
'weights_update'		x
'weights_update_norm_l2'		x

Attention

The operator get_dl_model_layer_weights is only applicable to self-created networks. For networks delivered by HALCON, the operator returns an empty tuple.

Parameters

▷ **Weights** (output_object) ..... image(-array) ⇝ *object :* real
Output weights.

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Handle of the deep learning model.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the layer to be queried.

▷ **WeightsType** (input_control) ..... string ⇝ *string*
Selected type of layer values to be returned.
Default: 'weights'
List of values: WeightsType ∈ {'weights', 'weights_norm_l2', 'weights_update', 'weights_update_norm_l2', 'weights_gradient', 'weights_gradient_norm_l2', 'bias', 'bias_norm_l2', 'bias_update', 'bias_update_norm_l2', 'bias_gradient', 'bias_gradient_norm_l2', 'batchnorm_mean', 'batchnorm_variance', 'batchnorm_mean_avg', 'batchnorm_variance_avg'}

Example

```
set_system ('seed_rand', 42)
* Create a small model network.
create_dl_layer_input ('input', [InputImageSize[0],InputImageSize[1],1], [],\
                       [], DLGraphNodeInput)
create_dl_layer_convolution (DLGraphNodeInput, 'conv', 3, 1, 1, 2, 1, 'none',\
                             'none', [], [], DLGraphNodeConvolution)
create_dl_layer_activation (DLGraphNodeConvolution, 'relu', 'relu', [], [],\
                            DLGraphNodeActivation)
create_dl_layer_dense (DLGraphNodeActivation, 'dense', 3, [], [],\
                       DLGraphNodeDense)
create_dl_layer_softmax (DLGraphNodeDense, 'softmax', [], [],\
                         DLGraphNodeSoftMax)
create_dl_model (DLGraphNodeSoftMax, DLModelHandle)
```

```
*
set_dl_model_param (DLModelHandle, 'type', 'classification')
set_dl_model_param (DLModelHandle, 'batch_size', 1)
set_dl_model_param (DLModelHandle, 'runtime', 'gpu')
set_dl_model_param (DLModelHandle, 'runtime_init', 'immediately')
*
* Train for 5 iterations.
for TrainIterations := 1 to NumTrainIterations by 1
    train_dl_model_batch (DLModelHandle, DLSample, DLTrainResult)
endfor
*
* Get the gradients, weights, and activations.
get_dl_model_layer_gradients (GradientsSoftmax, DLModelHandle, 'softmax')
get_dl_model_layer_gradients (GradientsDense, DLModelHandle, 'dense')
get_dl_model_layer_gradients (GradientsConv, DLModelHandle, 'conv')
*
get_dl_model_layer_weights (WeightsDense, DLModelHandle, 'dense',\
                        'weights_gradient')
get_dl_model_layer_weights (WeightsConv, DLModelHandle, 'conv',\
                        'weights_gradient')
*
get_dl_model_layer_activations (ActivationsDense,  DLModelHandle, 'dense')
get_dl_model_layer_activations (ActivationsConv,  DLModelHandle, 'conv')
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_dl_model, train_dl_classifier_batch, set_dl_model_layer_weights

Possible Successors

set_dl_model_layer_weights

Alternatives

get_dl_model_layer_activations, get_dl_model_layer_gradients

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
Deep Learning Professional

load_dl_model_weights (: : DLModelHandleSource, DLModelHandleTarget : ChangesByLayer)

Load the weights of a source model into a target model.

The operator load_dl_model_weights loads weights of a source model DLModelHandleSource into a target model DLModelHandleTarget. Thereby applies for every layer in the target model: Its weights are only changed if there is a layer in the source model having the same name and the same weight-shape. Note that DLModelHandleSource must be different from DLModelHandleTarget, i.e., you cannot use the same model handle as source and target.

ChangesByLayer is a tuple indicating for every target layer how many weights changed. Its entries are sorted by ascending layer IDs. The layer IDs can be queried via the operator get_dl_model_param with the parameter *'summary'*.

Note, that 'weights' means all weights and biases for all layers which can have such values (e.g., convolutional layer, batch normalization layer, etc.).

Parameters

▷ **DLModelHandleSource** (input_control) ..... dl_model ⇝ *handle*
Handle of the source deep learning model.

▷ **DLModelHandleTarget** (input_control) ..... dl_model ⇝ *handle*
Handle of the target deep learning model.

▷ **ChangesByLayer** (output_control) ..... integer(-array) ⇝ *integer*
Indicates for every target layer how many weights changed.

Result

If the parameters are valid, the operator `load_dl_model_weights` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Deep Learning Professional

set_dl_model_layer_param (: : DLModelHandle, LayerName, ParamName, ParamValue :)

Set parameter values of a given layer.

The operator `set_dl_model_layer_param` sets the value `ParamValue` of the parameter `ParamName` for a layer. The layer is referred to by its name `LayerName` in the model `DLModelHandle`. You can retrieve the layer names using `get_dl_model_param` with its option *'layer_names'* or *'summary'*.

Which generic and layer-specific parameters can be set is described in the entries of the operators (`create_dl_layer_*`), which are used for creating the layer.

Parameters

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Deep learning model.

▷ **LayerName** (input_control) ..... string ⇝ *string*
Name of the output layer.

▷ **ParamName** (input_control) ..... attribute.name ⇝ *string*
Name of the set parameter.
Default: []
List of values: `ParamName` ∈ {'bias_filler', 'bias_filler_variance_norm', 'bias_filler_const_val', 'is_inference_output', 'leaky_relu_alpha', 'learning_rate_multiplier', 'learning_rate_multiplier_bias', 'name', 'upper_bound', 'weight_filler', 'weight_filler_const_val', 'weight_filler_variance_norm'}

▷ **ParamValue** (input_control) ..... attribute.value(-array) ⇝ *string* / integer / real
Value of the set parameter.
Default: []
List of values: `ParamValue` ∈ {'xavier', 'msra', 'const', 'norm_in', 'norm_out', 'norm_average', 'true', 'false'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`set_dl_model_param`, `get_dl_model_param`, `get_dl_model_layer_param`

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Deep Learning Professional

set_dl_model_layer_weights (Weights : : DLModelHandle, LayerName, WeightsType :)

Set the weights (or values) of a Deep Learning model layer.

The operator set_dl_model_layer_weights sets for the model DLModelHandle the given Weights in the specified LayerName.

The parameter WeightsType determines which type of layer values are set. Which values can be set, please refer to the get_dl_model_layer_weights documentation.

Attention

The operator set_dl_model_layer_weights is only applicable to self-created networks. For networks delivered by HALCON, the operator does have no impact.

Parameters

- ▷ **Weights** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* real
  Input weights.
- ▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
  Handle of the deep learning model.
- ▷ **LayerName** (input_control) ..... string ⇝ *string*
  Name of the layer, whose weights are to be set.
- ▷ **WeightsType** (input_control) ..... string ⇝ *string*
  Selected type of layer values to be set.
  Default: 'weights'
  List of values: WeightsType ∈ {'weights', 'bias', 'batchnorm_mean', 'batchnorm_variance', 'batchnorm_mean_avg', 'batchnorm_variance_avg'}

Example

```
* Create weights for a convolution layer.
gen_image_const (Weights, 'real', 1, 1)
paint_region (Weights, Weights, Weights, 1, 'fill')
gen_empty_obj (WeightsArray)
for Index := 0 to 10 by 1
    concat_obj (WeightsArray, Weights, WeightsArray)
endfor
*
* Input image with rows consisting of 1s to 10s.
gen_image_const (Image, 'real', 10, 10)
for Index := 0 to 9 by 1
    gen_rectangle1 (Rectangle, Index, 0, Index, 9)
    paint_region (Rectangle, Image, Image, Index + 1, 'fill')
endfor
*
* Create a small model network.
create_dl_layer_input ('image', [10, 2, 1], [], [], ImageNode)
create_dl_layer_convolution (ImageNode, 'conv', 1, 1, 2, 11, 1, 'none', \
                             'none', [], [], ConvNode)
create_dl_layer_zoom_factor (ConvNode, 'zoom', 2, 2, 'bilinear', 'true', [],\
                             [], ZoomNode)
create_dl_model (ZoomNode, DLModelHandle)
set_dl_model_param (DLModelHandle, 'runtime', 'cpu')
*
```

```
* Set the weights to the convolution layer.
set_dl_model_layer_weights (WeightsArray, DLModelHandle, 'conv', 'weights')
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_dl_model`, `get_dl_model_layer_weights`

Possible Successors

`get_dl_model_layer_weights`

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
Deep Learning Professional

9.4 Model

This chapter explains the general concept of the deep learning (DL) model in HALCON and the data handling.

By concept, a deep learning model in HALCON is an internal representation of a deep neural network. Each deep neural network has an architecture defining its function, i.e., the tasks it can be used for. There can be several possible network architectures for one functionality. Currently, networks for the following functionalities are implemented in HALCON as model:

- 3D Gripping Point Detection, see 3D Matching / 3D Gripping Point Detection.
- Anomaly detection and Global Context Anomaly Detection, see Deep Learning / Anomaly Detection and Global Context Anomaly Detection.
- Classification, see Deep Learning / Classification.
- Deep OCR, see OCR / Deep OCR.
- Multi-Label Classification, see Deep Learning / Multi-Label Classification.
- Object detection and instance segmentation see Deep Learning / Object Detection and Instance Segmentation.
- Semantic segmentation and edge extraction, see Deep Learning / Semantic Segmentation and Edge Extraction.

Each functionality is identified by its unique model type. For the implemented methods you can find further information about the specific workflow, data requirements, and validation measures in the corresponding chapters. Information to deep learning in general are given in the chapter Deep Learning.

In this chapter you find the information, which data a DL model needs and returns as well as how this data is transferred.

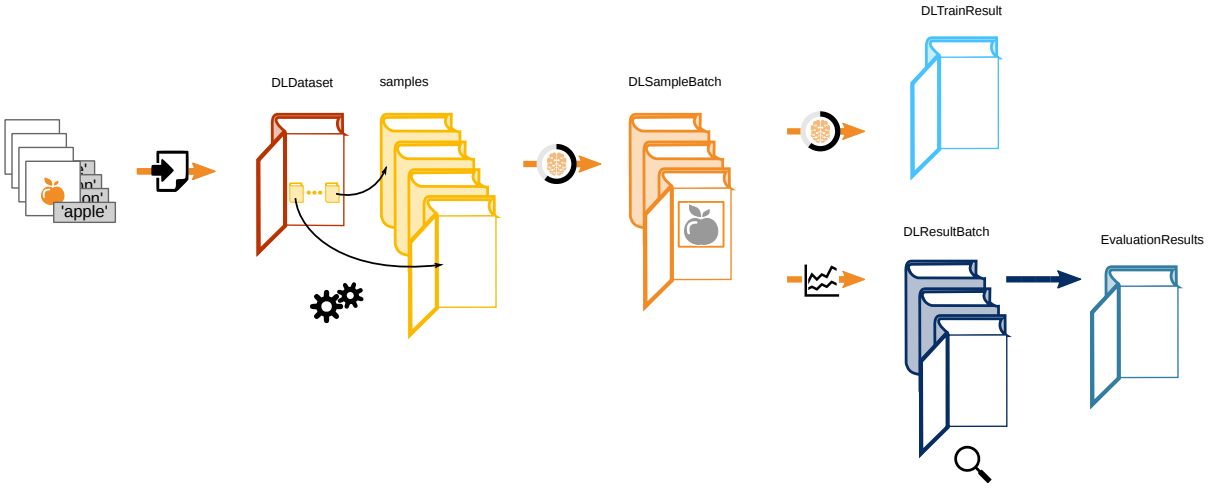
Data

Deep Learning applications have different types of data to be distinguished. Roughly spoken these are: The raw images with possible annotations, data preprocessed in a way suitable for the model, and output data.

Before the different types of data and the entries of the specific dictionaries are explained, we will have a look how the data is connected. Thereby, symbols and colors refer to the schematic overviews given below.

In brief, the data structure for training or evaluation starts with the raw images and their ground truth annotations (gray frames). With the read data the following dictionaries are created: A dictionary `DLDataset` (red), which serves as database and refers to a specific dictionary (yellow) for every input image. The dictionary `DLSample` (orange) contains the data for a sample in the way the network can process it. A batch of `DLSample` is handed to the model in `DLSampleBatch`. For evaluation, `DLResultBatch` is returned, a tuple of dictionaries `DLResult` (dark blue), one for every sample. They are needed to obtain the evaluation results `EvaluationResult`. For training, the training results (e.g., loss values) are returned in the dictionary `DLTrainResult` (light blue). The most important steps concerning modifying or creating a dictionary:

- reading the raw data (symbol: paper with arrow)
- preprocessing the data (symbol: cogs)
- training (symbol: transparent brain in an arc)
- evaluation of the model (symbol: graph)
- evaluation of a sample (symbol: magnifying glass)



Schematic overview of the data structure during training and evaluation.

For inference no annotations are needed. Thus, the data structure starts with the raw images (gray frames). The dictionary `DLSample` (orange) contains the data for a sample in the way the network can process it. The results for a sample are returned in a dictionary `DLResult` (dark blue). The most important steps concerning modifying or creating a dictionary:

- reading the raw data (symbol: paper with arrow)
- preprocessing the data (symbol: cogs)
- inference (symbol: brain in a circle)
- evaluation of a sample (symbol: magnifying glass)

Schematic overview of the data connection during inference.

In order for the model to process the data, the data needs to follow certain conventions about what is needed and how it is given to the model. As visible from the figures above, in HALCON the data is transferred using dictionaries.

In the following we explain the involved dictionaries, how they can be created, and their entries. Thereby, we group them according to the main step of a deep learning application they are created in and whether they serve as input or output data. The following abbreviations mark for which methods the entry applies:

- 'Any': any method
- '3D-GPD': 3D Gripping Point Detection
- 'AD': anomaly detection
- 'CL': classification
- 'MLC': multi-label classification
- 'OCR-D': Deep OCR detection component
- 'OCR-R': Deep OCR recognition component
- 'GC-AD': Global Context Anomaly Detection
- 'OD': object detection

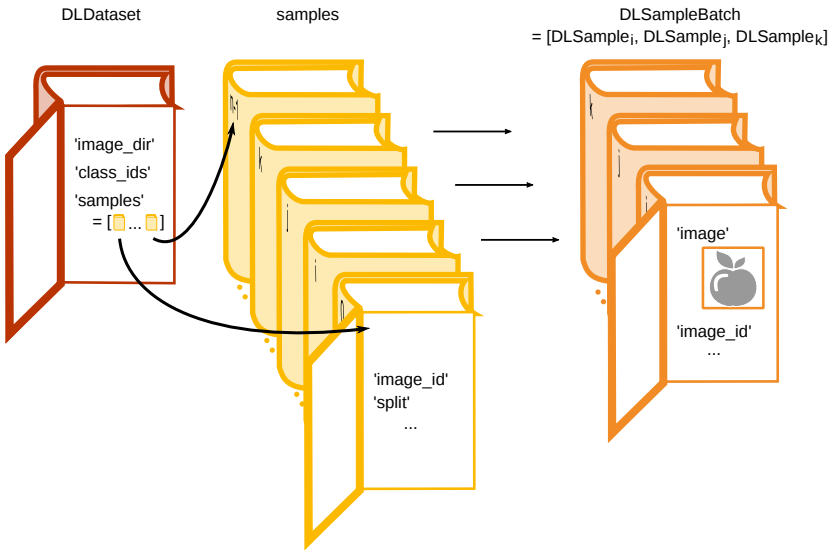
  In case the entry is only applicable for a certain *'instance_type'*, the specification 'r1': *'rectangle1'*, 'r2': *'rectangle2'* is added.

  For entries only applicable for instance segmentation the specification 'is' is added.
- 'SE': semantic segmentation

The entries only applicable for certain methods are described more extensively in the corresponding chapter.

Training and evaluation input data The dataset consists of images and corresponding information. They have to be provided in a way the model can process them. Concerning the image requirements, find more information in the section "Images" below.

The information about the images and the dataset is represented in a dictionary `DLDataset`, which serves as a database. More precisely, it stores the general information about the dataset and the dictionaries of the individual samples collected under the key `samples`. When the actual image data is needed, a dictionary `DLSample` is created (or read if it already exists) for each image required. The relation of these dictionaries is illustrated in the figure below.



Schematic illustration of the different dataset dictionaries used for training and evaluation. For visibility purpose only few entries are registered and `BatchSize` is set to three. In this example we have n samples. Thereof three samples are chosen randomly: i,j, and k. The corresponding dictionaries `DLSample` are created and joined in the tuple `DLSampleBatch`.

In the following we look at these dictionaries.

DLDataset The dictionary `DLDataset` serves as a database. It stores general information about the dataset and collects the dictionaries of the individual samples. Thereby iconic data is not included in `DLDataset` but the paths to the respective images. The dictionary `DLDataset` is used by the training and evaluation procedures. It is not necessary for the model, but we highly recommend to create it. Its necessary entries are described below. This dictionary is either created directly when labeling your data using the MVTec Deep Learning Tool or it is created by one of the following method-specific procedures:

- `read_dl_dataset_3d_gripping_point_detection` (3D Gripping Point Detection)
- `read_dl_dataset_anomaly` (anomaly detection, Global Context Anomaly Detection)
- `read_dl_dataset_classification` (classification)
- `read_dl_dataset_ocr_detection` (Deep OCR - detection component)
- `read_dl_dataset_ocr_recognition` (Deep OCR - recognition component)
- `read_dl_dataset_from_coco` (object detection with *'instance_type' = 'rectangle1'*)
- `read_dl_dataset_segmentation` (semantic segmentation).

Please see the respective procedure documentation for the requirements on the data in order to use these procedures. In case you create `DLDataset` in an other way, it has to contain at least the entries not marked with a number in the description below. During the preprocessing of your dataset the respective procedures include the further entries of the dictionary `DLDataset`.

Depending on the model type, this dictionary can have the following entries:

image_dir: Any
Common base path to all images.
format: string

dlsample_dir: Any [1]
Common base path of all sample files (if present).
format: string

class_names: Any except OCR-R
Names of all classes that are to be distinguished.
format: tuple of strings

class_ids: Any except OCR-R
IDs of all classes that are to be distinguished (range: 0-65534).
format: tuple of integers

preprocess_param: Any [1]
All parameter values used during preprocessing.
format: dictionary

samples: Any
Collection of sample descriptions.
format: tuple of dictionaries

normals_dir: 3D-GPD
Optional. Common base path of all normals images.
format: string

xyz_dir: 3D-GPD
Common base path of all XYZ-images.
format: string

anomaly_dir: AD, GC-AD
Common base path of all anomaly regions (regions indicating anomalies in the image).
format: string

class_weights: CL, SE [1]
Weights of the different classes.
format: tuple of reals

segmentation_dir: SE, 3D-GPD
Common base path of all segmentation images.
format: string

This dictionary is directly created when labeling your data using the MVTec Deep Learning Tool. It is also created by the procedures mentioned above for reading in your data. The entries marked with [1] are added by the preprocessing procedures.

samples The `DLDataset` key `samples` gets a tuple of dictionaries as value, one for each sample in the dataset. These dictionaries contain the information concerning an individual sample of the dataset. Depending on the model type, this dictionary can have the following entries:

image_file_name: Any
File name of the image and its path relative to `image_dir`.
format: string

image_id: Any
Unique image ID (encoding format: UINT8).
format: integer

split: Any [2]
Specifies the assigned split subset (*'train'*,*'validation'*,*'test'*).
format: string

dlsample_file_name: Any [3]
File name of the corresponding dictionary `DLSample` and its path relative to `dlsample_dir`.
format: string

normals_file_name: 3D-GPD
Optional. File name of the normals image and its path relative to `normals_dir`.
format: string

segmentation_file_name: 3D-GPD, SE
File name of the ground truth segmentation image and its path relative to `segmentation_dir`.
format: string

xyz_file_name: 3D-GPD
File name of the XYZ-image and its path relative to `xyz_dir`.
format: string

anomaly_file_name: AD, GC-AD
Optional. Path to region files with ground truth annotations (relative to `anomaly_dir`).
format: string

anomaly_label: AD, GC-AD
Ground truth anomaly label on image level (in the form of `class_names`).
format: string

image_label_id: CL
Ground truth label for the image (in the form of `class_ids`).
format: tuple of integers

image_label_ids: MLC
Ground truth labels for the image (in the form of `class_ids`).
format: tuple of integers

image_id_origin: OCR-R
ID of the original image the sample was extracted from.
format: integer

word: OCR-D, OCR-R
Ground truth word.
format: string

bbox_label_id: OD, OCR-D
Ground truth labels for the bounding boxes (in the form of `class_ids`).
format: tuple of integers

bbox_row1: OD:r1 [4]
Ground truth bounding boxes: upper left corner, row coordinate.
format: tuple of reals

bbox_col1: OD:r1 [4]
Ground truth bounding boxes: upper left corner, column coordinate.
format: tuple of reals

bbox_row2: OD:r1 [4]
Ground truth bounding boxes: lower right corner, row coordinate.
format: tuple of reals

bbox_col2: OD:r1 [4]
Ground truth bounding boxes: lower right corner, column coordinate.
format: tuple of reals

coco_raw_annotations: OD:r1
Optional. It contains for every bbox_label_id within this image a dictionary with all raw COCO annotation information.
format: tuple of dictionaries

bbox_row: OCR-D, OCR-R, OD:r2 [4]
Ground truth bounding boxes: center point, row coordinate.
format: tuple of reals

bbox_col: OCR-D, OCR-R, OD:r2 [4]
Ground truth bounding boxes: center point, column coordinate.
format: tuple of reals

bbox_phi: OCR-D, OCR-R, OD:r2 [4]
Ground truth bounding boxes: angle phi.
format: tuple of reals

bbox_length1: OCR-D, OCR-R, OD:r2 [4]
Ground truth bounding boxes: half length of edge 1.
format: tuple of reals

bbox_length2: OCR-D, OCR-R, OD:r2 [4]
Ground truth bounding boxes: half length of edge 2.
format: tuple of reals

mask: OD:is
Ground truth mask marking the instance regions.
format: tuple of regions

These dictionaries are part of DLDataset and thus they are created concurrently. An exception are the entries with a mark in the table, [2]: the procedure split_dl_dataset adds split, [3]: the procedure preprocess_dl_samples adds dlsample_file_name. [4]: Used coordinates: Pixel centered, subpixel accurate coordinates.

DLSample The dictionary DLSample serves as input for the model. For a batch, they are handed over as the entries of the tuple DLSampleBatch for apply_dl_model or train_dl_model_batch. They are created out of DLDataset for every sample by the procedure gen_dl_samples followed by preprocess_dl_samples. Note, preprocess_dl_samples will update the corresponding DLSample dictionary. If preprocessing is done using the standard procedure preprocess_dl_dataset, the preprocessed samples are stored on the file system. Afterwards they need to be retrieved with the procedure read_dl_samples.

DLSample contains the preprocessed image and, in case of training and evaluation, all ground truth annotations. Depending on the model type, it can have the following entries:

anomaly_ground_truth: AD, GC-AD
Anomaly image or region, read from anomaly_file_name.
format: image or region

anomaly_label: AD, GC-AD
Ground truth anomaly label on image level (in the form of class_names).
format: string

anomaly_label_id: AD, GC-AD
Ground truth anomaly label ID on image level (in the form of class_ids).
format: integer

bbox_label_id: OD
Ground truth labels for the image part within the bounding box (in the form of class_ids).
format: tuple of integers

bbox_row1: OD:r1 [4]
Ground truth bounding boxes: upper left corner, row coordinate.
format: tuple of reals

bbox_col1: OD:r1 [4]
Ground truth bounding boxes: upper left corner, column coordinate.
format: tuple of reals

bbox_row2: OD:r1 [4]
Ground truth bounding boxes: lower right corner, row coordinate.
format: tuple of reals

bbox_col2: OD:r1 [4]
Ground truth bounding boxes: lower right corner, column coordinate.
format: tuple of reals

bbox_row: OCR-D, OD:r2 [4]
Ground truth bounding boxes: center point, row coordinate.
format: tuple of reals

bbox_col: OCR-D, OD:r2 [4]
Ground truth bounding boxes: center point, column coordinate.
format: tuple of reals

bbox_phi: OCR-D, OD:r2 [4]
Ground truth bounding boxes: angle phi.
format: tuple of reals

bbox_length1: OCR-D, OD:r2 [4]
Ground truth bounding boxes: half length of edge 1.
format: tuple of reals

bbox_length2: OCR-D, OD:r2 [4]
Ground truth bounding boxes: half length of edge 2.
format: tuple of reals

image: Any
Input image.
format: image

image_label_id: CL
Ground truth label for the image (in the form of `class_ids`).
format: integer

image_label_ids: MLC
Ground truth labels for the image (in the form of `class_ids`).
format: tuple of integers

mask: OD:is
Ground truth mask marking the instance regions.
format: tuple of regions

normals: 3D-GPD
2D mappings (3-channel image)
format: image

segmentation_image: SE, 3D-GPD
Image with the ground truth segmentations, read from `segmentation_file_name`.
format: image

weight_image: SE [5]
Image with the pixel weights.
format: image

target_orientation: OCR-D
Orientation target image for the word orientation.
format: image

target_text: OCR-D
Text target image for the character detection.
format: image

target_link: OCR-D
Link target image for the connection of detected character centers to a connected word.
format: image

target_weight_orientation: OCR-D
Weight with respect to `target_orientation`.
format: image

target_weight_link: OCR-D
Weight with respect to `target_link`.
format: image

target_weight_text: OCR-D
Weight with respect to target_text.
format: image

word: OCR-D, OCR-R
Ground truth word.
format: string

x: 3D-GPD
X-image (values need to increase from left to right).
format: image

y: 3D-GPD
Y-image (values need to increase from top to bottom).
format: image

z: 3D-GPD
Z-image (values need to increase from points close to the sensor to far points; this is for example the case if the data is given in the camera coordinate system).
format: image

These dictionaries are created by the procedure gen_dl_samples followed by preprocess_dl_samples. An exception is the entry marked in the table above, [5]: created by the procedure gen_dl_segmentation_weights. [4]: Used coordinates: Pixel centered, subpixel accurate coordinates.

Inference input data The inference input data consists of a single DLSample dictionary or a tuple of such. In contrast to training and evaluation, only the following keys are used:

image: Any
Input image
format: image

normals: 3D-GPD
2D mappings (3-channel image).
format: image

x: 3D-GPD
X-image (values need to increase from left to right).
format: image

y: 3D-GPD
Y-image (values need to increase from top to bottom).
format: image

z: 3D-GPD
Z-image (values need to increase from points close to the sensor to far points; this is for example the case if the data is given in the camera coordinate system).
format: image

Concerning the image requirements, find more information in the subsection "Images" below.

For the inference, such a dictionary containing only the image data can be created using the procedure gen_dl_samples_from_images or gen_dl_samples_3d_gripping_point_detection (only for 3D Gripping Point Detection). These dictionaries can be passed one at a time or within a tuple DLSampleBatch.

Training output data The training output data is given in the dictionary DLTrainResult. Its entries depend on the model and thus on the operator used (for further information see the documentation of the corresponding operator):

3D-GPD, CL, MLC, OCR-D, OCR-R, GC-AD, OD, SE:
The operator train_dl_model_batch returns

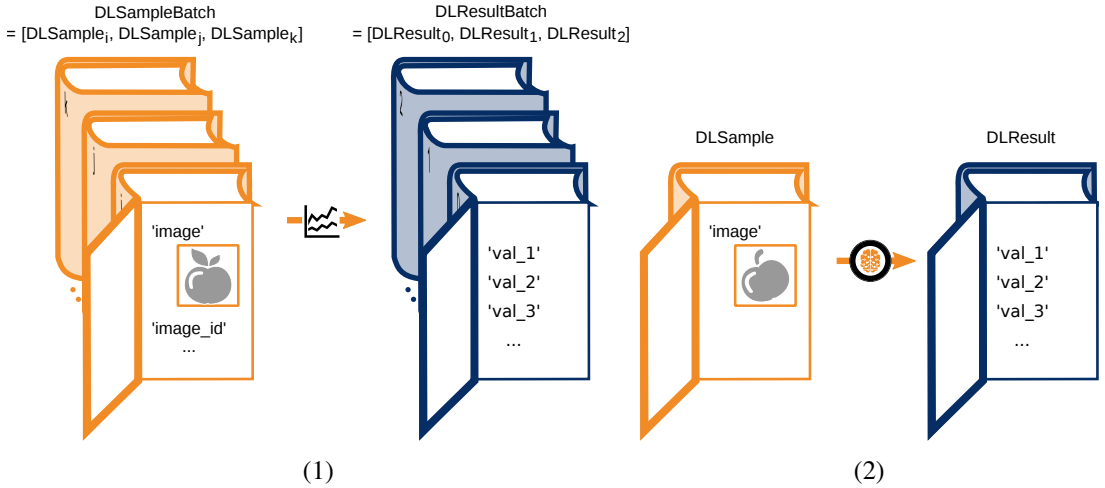
- total_loss
- possible further losses included in your model

AD:

The operator `train_dl_model_anomaly_dataset` returns

- `final_error`
- `final_epoch`

Inference and evaluation output data As output from the operator `apply_dl_model`, the model will return a dictionary `DLResult` for each sample. An illustration is given in the figure below. The evaluation is based on these results and the annotations. Evaluation results are stored in the dictionary `EvaluationResult`.



Schematic illustration of the dictionaries serving as model input: (1) Evaluation: `DLSample` includes the image as well as information about the image and its content. This data serves as basis for the evaluation. For visibility purpose `BatchSize` is set to three (containing the randomly chosen samples i,j,and k, see above) and only few entries are registered. (2) Inference: `DLSample` contains only the image. These dictionaries can be passed one at a time or within a tuple.

Depending on the model type, the dictionary `DLResult` can have the following entries:

`gripping_confidence`: 3D-GPD
Image, containing raw, uncalibrated confidence values for every point in the scene.
format: image

`gripping_map`: 3D-GPD
Binary image, indicating for each pixel of the scene whether the model predicted a gripping point (pixel value = *1.0*) or not (*0.0*).
format: image

`anomaly_image`: AD, GC-AD
Single channel image whose gray values are scores, indicating how likely the corresponding pixel in the input image belongs to an anomaly.
format: image

`anomaly_image_combined`: GC-AD
Single channel image whose gray values are scores, indicating how likely the corresponding pixel in the input image belongs to an anomaly. Calculated by combining the *'local'* and *'global'* subnetworks of the model.
Format: Bild

`anomaly_image_global`: GC-AD
Single channel image whose gray values are scores, indicating how likely the corresponding pixel in the input image belongs to an anomaly. Calculated by the *'global'* subnetwork of the model.
format: image

`anomaly_image_local`: GC-AD
Single channel image whose gray values are scores, indicating how likely the corresponding pixel in the input image belongs to an anomaly. Calculated by the *'local'* subnetwork of the model.
format: image

anomaly_score: AD, GC-AD
Anomaly score on image level calculated from anomaly_image.
format: real

anomaly_score_local: GC-AD
Anomaly score on image level calculated from anomaly_image_local.
format: real

anomaly_score_global: GC-AD
Anomaly score on image level calculated from anomaly_image_global.
format: real

classification_class_ids: CL
Inferred class ids for the image sorted by confidence values.
format: tuple of integers

classification_class_names: CL
Inferred class names for the image sorted by confidence values.
format: tuple of strings

classification_confidences: CL
Confidence values of the image inference for each class.
format: tuple of reals

class_ids: MLC
Inferred class ids for the image sorted by confidence values.
format: tuple of integers

class_names: MLC
Inferred class names for the image sorted by confidence values.
format: tuple of strings

confidences: MLC
Confidence values of the image inference for each class.
format: tuple of reals

selected_class_ids: MLC
Class ids for the image selected by the confidence threshold(min_confidence).
format: tuple of integers

selected_class_names: MLC
Class names for the image selected by the confidence threshold (min_confidence).
format: tuple of strings

selected_confidences: MLC
Confidence values of the image selected by the confidence threshold for each class.
format: tuple of reals

char_candidates: OCR-R
Candidates for each character of the word and their confidences.
format: tuple of dictionaries

word: OCR-R
Recognized word.
format: string

score_maps: OCR-D Scores given as image with four channels:

- Character score: Score for the character detection.
- Link score: Score for the connection of detected character centers to a connected word.
- Orientation 1: Sine component of the predicted word orientation.
- Orientation 2: Cosine component of the predicted word orientation.

format: image

words: OCR-D Dictionary containing the following entries. Thereby, the entries are tuples with a value for every found word.

- row: Localized word: Center point, row coordinate.

- `col`: Localized word: Center point, column coordinate.
- `phi`: Localized word: Angle phi.
- `length1`: Localized word: Half length of edge 1.
- `length2`: Localized word: Half length of edge 2.
- `line_index`: Line index of localized word if *'detection_sort_by_line'* set to *'true'*.

format: dictionary with tuples of reals and strings

word_boxes_on_image: OCR-D Dictionary with the word localization on the coordinate system of the preprocessed images placed in `image`. The entries are tuples with a value for every found word.

- `row`: Localized word: Center point, row coordinate.
- `col`: Localized word: Center point, column coordinate.
- `phi`: Localized word: Angle phi.
- `length1`: Localized word: Half length of edge 1.
- `length2`: Localized word: Half length of edge 2.

format: dictionary with tuples of reals

word_boxes_on_score_maps: OCR-D Dictionary with the word localization on the coordinate system of the score images placed in `score_maps`. The entries are the same as for `word_boxes_on_image` above. *format:* dictionary with tuples of reals

bbox_class_id: OD
Inferred class for the bounding box (in the form of `class_ids`).
format: tuple of integers

bbox_class_name: OD
Name of the inferred class for the bounding box.
format: tuple of strings

bbox_confidence: OD
Confidence value of the inference for the bounding box.
format: tuple of reals

bbox_row1: OD:r1 [6]
Inferred bounding boxes: upper left corner, row coordinate.
format: tuple of reals

bbox_col1: OD:r1 [6]
Inferred bounding boxes: upper left corner, column coordinate.
format: tuple of reals

bbox_row2: OD:r1 [6]
Inferred bounding boxes: lower right corner, row coordinate.
format: tuple of reals

bbox_col2: OD:r1 [6]
Inferred bounding boxes: lower right corner, row coordinate.
format: tuple of reals

bbox_row: OD:r2 [6]
Inferred bounding boxes: center point, row coordinate.
format: tuple of reals

bbox_col: OD:r2 [6]
Inferred bounding boxes: center point, column coordinate.
format: tuple of reals

bbox_phi: OD:r2 [6]
Inferred bounding boxes: angle phi.
format: tuple of reals

bbox_length1: OD:r2 [6]
Inferred bounding boxes: half length of edge 1.
format: tuple of reals

bbox_length2: OD:r2 [6]
Inferred bounding boxes: half length of edge 2.
format: tuple of reals

mask: OD:is
Inferred mask marking the instance regions.
format: tuple of regions

mask_probs: OD:is
Image with the confidence values of the inferred mask.
format: image

segmentation_image: SE
Image with the segmentation result.
format: image

segmentation_confidence: SE
Image with the confidence values of the segmentation result.
format: image

[6]: Used coordinates: Pixel centered, subpixel accurate coordinates.

For a further explanation to the output values we refer to the chapters of the respective method, e.g., Deep Learning / Semantic Segmentation and Edge Extraction.

Images Regardless of the application, the network poses requirements on the images. The specific values depend on the network itself and can be queried using `get_dl_model_param`. In order to fulfill these requirements, you may have to preprocess your images. Standard preprocessing of the entire dataset and therewith also the images is implemented in `preprocess_dl_samples`. In case of custom preprocessing this procedure offers guidance on the implementation.

add_dl_pruning_batch (: : DLModelHandleToPrune, DLPruningHandle, DLSampleBatch :)

Calculate scores to prune a deep learning model.

`add_dl_pruning_batch` calculates pruning scores for the deep learning model `DLModelHandleToPrune`. More precisely, the scores are calculated on the images given in `DLSampleBatch` and internally accumulated by each call of `add_dl_pruning_batch` in the pruning data handle `DLPruningHandle`.

The parameter `DLModelHandleToPrune` specifies the deep learning model to use. Note that `add_dl_pruning_batch` supports only deep learning models of type *'classification'*.

The parameter `DLPruningHandle` specifies the pruning data handle, which is used to pass information as e.g., the accumulated scores or the pruning mode. See `create_dl_pruning` for further information about implemented pruning modes.

The parameter `DLSampleBatch` specifies the batch with input images based on which the scores are calculated. Note that the number of images in the tuple `DLSampleBatch` needs to be equal to the value set for the model parameter *'batch_size'*.

For an explanation of the concept of deep learning see the introduction of chapter Deep Learning.

Parameters

▷ **DLModelHandleToPrune** (input_control) . . . . . dl_model ⇝ *handle*
Handle of a deep learning model to prune.

▷ **DLPruningHandle** (input_control) . . . . . dl_pruning ⇝ *handle*
Pruning data handle.

▷ **DLSampleBatch** (input_control) . . . . . dict-array ⇝ *handle*
Tuple of dictionaries with input images.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_dl_model, create_dl_pruning, set_dl_pruning_param

Possible Successors

get_dl_pruning_param, gen_dl_pruned_model

Module

Deep Learning Professional

apply_dl_model (: : DLModelHandle, DLSampleBatch, Outputs : DLResultBatch)

Apply a deep-learning-based network on a set of images for inference.

apply_dl_model applies the deep-learning-based network given by DLModelHandle on the batch of input images handed over through the tuple of dictionaries DLSampleBatch. The operator returns DLResultBatch, a tuple with a result dictionary DLResult for every input image.

Please see the chapter Deep Learning / Model for more information on the concept and the dictionaries of the deep learning model in HALCON.

In order to apply the network on images, you have to hand them over through a tuple of dictionaries DLSampleBatch, where a dictionary refers to a single image. You can create such a dictionary conveniently using the procedure gen_dl_samples_from_images. The tuple DLSampleBatch can contain an arbitrary number of dictionaries. The operator apply_dl_model always processes a batch with up to *'batch_size'* images simultaneously. In case the tuple contains more images, apply_dl_model iterates over the necessary number of batches internally. For a DLSampleBatch with less than *'batch_size'* images, the tuple is padded to a full batch which means that the time required to process a DLSampleBatch is independent of whether the batch is filled up or just consists of a single image. This also means that if fewer images than *'batch_size'* are processed in one operator call, the network still requires the same amount of memory as for a full batch. The current value of *'batch_size'* can be retrieved using get_dl_model_param.

Note that the images might have to be preprocessed before feeding them into the operator apply_dl_model in order to fulfill the network requirements. You can retrieve the current requirements of your network, such as e.g., the image dimensions, using get_dl_model_param. The procedure preprocess_dl_dataset provides guidance on how to implement such a preprocessing stage.

The results are returned in DLResultBatch, a tuple with a dictionary DLResult for every input image. Please see the chapter Deep Learning / Model for more information to the output dictionaries in DLResultBatch and their keys. In Outputs you can specify, which output data is returned in DLResult. Outputs can be a single string, a tuple of strings, or an empty tuple with which you retrieve all possible outputs. If apply_dl_model is used with an AI^2-interface, it might be required to set *'is_inference_output'* = *'true'* for all requested layers in Outputs before the model is optimized for the AI^2-interface, see optimize_dl_model_for_inference and set_dl_model_layer_param for further details. The values for Outputs depend on the model type of your network:

Models of *'type'*=*'3d_gripping_point_detection'*

- Outputs=*'[]'*: DLResult containing:
  - 'gripping_map': Binary image, indicating for each pixel of the scene whether the model predicted a gripping point (pixel value = *1.0*) or not (*0.0*).
  - 'gripping_confidence': Image, containing raw, uncalibrated confidence values for every point in the scene.

Models of *'type'*=*'anomaly_detection'*

- Outputs=*'[]'*: DLResult contains an image where each pixel has the score of the according input image pixel. Additionally it contains a score for the entire image.

Models of *'type'='counting'*

This model type cannot be run with the operator `apply_dl_model`.

Models of *'type'='gc_anomaly_detection'*

For each value of `Outputs`, `DLResult` contains an image where each pixel has the score of the according input image pixel. Additionally it contains a score for the entire image.

- `Outputs`=*'[]'*: The scores of each input image pixel are calculated as a combination of all available networks.
- `Outputs`=*'anomaly_image_local'*: The scores of each input image pixel are calculated from the *'local'* network only. If the *'local'* network is not available, an error is raised.
- `Outputs`=*'anomaly_image_global'*: The scores of each input image pixel are calculated from the *'global'* network only. If the *'global'* network is not available, an error is raised.
- `Outputs`=*'anomaly_image_combined'*: The scores of each input image pixel are calculated by combining the *'global'* and the *'local'* networks. If one or both of the networks are not available, an error is raised.

Models of *'type'='classification'*

- `Outputs`=*'[]'*: `DLResult` contains a tuple with confidence values in descending order and tuples with the class names and class IDs sorted accordingly. If Out-of-Distribution Detection is available, `DLResult` additionally contains the out-of-distribution score, the out-of-distribution result and the threshold that was used for the prediction.

Models of *'type'='multi_label_classification'*

- `Outputs`=*'[]'*: `DLResult` contains a tuple with the selected class names, class IDs and the corresponding confidence values according to the model parameter *'min_confidence'*. Additionally, it contains tuples with all class names, class IDs and corresponding confidence values.

Models of *'type'='detection'*

- `Outputs`=*'[]'*: `DLResult` contains the bounding box coordinates as well as the inferred classes and their confidence values resulting from all levels.
- `Outputs`= *'[bboxhead + level + _prediction, classhead + level + _prediction]'*, where *'level'* stands for the selected level which lies between *'min_level'* and *'max_level'*: `DLResult` contains the bounding box coordinates as well as the inferred classes and their confidence values resulting from specific levels.

Models of *'type'='ocr_recognition'*

- `Outputs`=*'[]'*: `DLResult` contains the recognized word. Additionally it contains candidates for each character of the word and their confidences.

Models of *'type'='ocr_detection'*

- `Outputs`=*'[]'*: `DLResult` contains the bounding boxes coordinates of localized words.

Models of *'type'='segmentation'*

- `Outputs`=*'segmentation_image'*: `DLResult` contains an image where each pixel has a value corresponding to the class its corresponding pixel has been assigned to.
- `Outputs`=*'segmentation_confidence'*: `DLResult` contains an image where each pixel has the confidence value out of the classification of the according pixel.
- `Outputs`=*'[]'*: `DLResult` contains all output values.

Attention

System requirements: To run this operator on GPU by setting *'device'* to *'gpu'* (see `get_dl_model_param`), cuDNN and cuBLAS are required. For further details, please refer to the "`Installation Guide`", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods".

Parameters

▷ **DLModelHandle** (input_control) . . . . . dl_model ⇝ *handle*
Handle of the deep learning model.

▷ **DLSampleBatch** (input_control) . . . . . dict-array ⇝ *handle*
Input data.

▷ **Outputs** (input_control) . . . . . string-array ⇝ *string*
Requested outputs.
Default: []
List of values: `Outputs` ∈ {[], 'segmentation_image', 'segmentation_confidence', 'bboxhead2_prediction', 'classhead2_prediction'}

▷ **DLResultBatch** (output_control) . . . . . dict-array ⇝ *handle*
Result data.

Result

If the parameters are valid, the operator `apply_dl_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`read_dl_model`, `train_dl_model_batch`, `train_dl_model_anomaly_dataset`, `set_dl_model_param`

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Deep Learning Enhanced, Deep Learning Professional

clear_dl_model (: : DLModelHandle :)

Clear a deep learning model.

`clear_dl_model` clears the handle of the deep learning model given by `DLModelHandle` and frees all memory required for the model. After calling `clear_dl_model`, the model can no longer be used and the handle `DLModelHandle` becomes invalid.

For further explanations to deep learning models in HALCON, see the chapter Deep Learning / Model.

Parameters

▷ **DLModelHandle** (input_control) . . . . . dl_model(-array) ⇝ *handle*
Handle of the deep learning model.

Result

If the parameters are valid, the operator `clear_dl_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DLModelHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

read_dl_model, apply_dl_model, train_dl_model_batch, train_dl_model_anomaly_dataset

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Matching, Deep Learning Enhanced, Deep Learning Professional

create_dl_pruning (: : DLModelHandle, Mode, GenParam : DLPruningHandle)

Create a pruning data handle.

The operator create_dl_pruning creates a handle DLPruningHandle. This handle is used to pass information when pruning a deep learning model.

Mode specifies the pruning method. The mode implies what will be removed and also which scores are needed. Currently only the following *'mode'* is implemented:

'oracle': Kernels of convolution layers are removed. In order to do so, a score is calculated for every (potentially removable) kernel, indicating its importance within the given network for the images used. See the given reference for details.

GenParam is a dictionary for setting generic parameters. Currently no generic parameters are supported.

For an explanation of the concept of deep learning see the introduction of chapter Deep Learning.

Parameters

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Handle of a deep learning model.

▷ **Mode** (input_control) ..... string ⇝ *string*
Pruning method.
Default: 'oracle'
List of values: Mode ∈ {'oracle'}

▷ **GenParam** (input_control) ..... dict ⇝ *handle*
Dictionary with generic parameters.
Default: []

▷ **DLPruningHandle** (output_control) ..... dl_pruning ⇝ *handle*
Pruning data handle.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_dl_model

Possible Successors

get_dl_pruning_param, set_dl_model_param

References

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, Jan Kautz "Pruning Convolutional Neural Networks

for Resource Efficient Inference", ICLR 2017, 5th International Conference on Learning Representations, Toulon, France.

Module

Deep Learning Professional

deserialize_dl_model (: : SerializedItemHandle : DLModelHandle)

Deserialize a deep learning model.

deserialize_dl_model deserializes the deep learning model defined by the handle SerializedItemHandle and previously serialized by serialize_dl_model.

The operator acts the same as read_dl_model except that the input is a serialized item instead of a file. For a detailed description please refer to the documentation of read_dl_model.

For further explanations to deep learning models in HALCON, see the chapter Deep Learning / Model.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **DLModelHandle** (output_control) ..... dl_model ⇝ *handle*
Handle of the deep learning model.

Result

If the parameters are valid, the operator deserialize_dl_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_dl_model

Possible Successors

apply_dl_model, train_dl_model_batch, train_dl_model_anomaly_dataset, set_dl_model_param, get_dl_model_param

See also

serialize_dl_model

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Matching, Deep Learning Enhanced, Deep Learning Professional

gen_dl_model_heatmap (: : DLModelHandle, DLSample, HeatmapMethod, TargetClasses, GenParam : DLResult)

Infer the sample and generate a heatmap.

The operator gen_dl_model_heatmap infers the sample given by DLSample and generates a heatmap for the class specified by TargetClasses. DLSample can be a single dictionary or a tuple of dictionaries for different samples. To do so, the operator uses the deep learning model given by DLModelHandle. The parameter HeatmapMethod determines, which method is used to calculate the heatmap. The operator returns DLResult with a result dictionary for every given sample. Note, DLResult will be newly created and already existing parameters with this name will be overwritten.

A heatmap can be useful to analyze which parts of an image have a strong influence for the inference into a certain class. Shape and area of these parts and therewith the heatmaps may vary widely for differing networks especially if their architectures differ. There are different methods how a heatmap is calculated. The following ones can be selected using HeatmapMethod:

'grad_cam': Provides region-based heatmap information based on Gradient-weighted Class Activation Mapping (Grad-CAM). For a more detailed description refer to the referenced paper below.

'guided_grad_cam': Provides pixel-based heatmap information based on Guided gradient-weighted Class Activation Mapping (Guided Grad-CAM). For a more detailed description refer to the referenced paper below.

The input parameter TargetClasses determines the target class, the class for which the heatmap is generated. As value the class ID has to be set. Alternatively, an empty tuple can be handed over. In this case, the heatmap is calculated for the class with the highest confidence value, thus the top inferred class. Currently, TargetClasses only supports a single value or an empty tuple.

The following entries can be set in the dictionary GenParam:

target_conv_layer: Specifies the convolution layer whose activation and gradient are used for the heatmap. The convolution layer can be specified using its NAME, as returned using *'summary'* in get_dl_model_param, e.g., *'conv10'*. As default, the last (deepest) convolution layer of the network with a non-trivial activation (width and height not equal to 1) is used. As a general rule, the activation of the deepest convolution layer is most suitable for calculating the heatmap, therefore it is recommended to keep the default layer.

use_conv_only: In case the convolution layer (selected with target_conv_layer) is fused with a directly following ReLU activation layer, this parameter can be used to determine, from which of these layers the activation and gradient will be used for the heatmap. The following values are supported:

'true': convolution layer

'false': ReLU layer

Restriction: Only applicable if HeatmapMethod is set to *'grad_cam'*.

Default: use_conv_only = *'true'*.

scaling: Sets the scaling method for the heatmap. The following values are supported:

'scale_after_relu': Negative values of the heatmap are set to 0 and then all values are scaled within the range [0,1]. As a consequence, areas within the heatmap can attain very high values close to 1 although their contribution to the classification result might be small.

'scale_before_relu': All values of the heatmap are divided by the maximum absolute value and then negative values are set to 0. This leads to values for the heatmap in [0,1]. However, areas of less activation (and hence with a small contribution to the classification result) may not tend to attain values near 1.

'none': The heatmap values are not scaled.

Restriction: Only applicable if HeatmapMethod is set to *'grad_cam'*.

Default: scaling = *'scale_after_relu'*.

Every output dictionary in DLResult contains the inference results as obtained using apply_dl_model. Additionally it includes a nested dictionary under the key heatmap_method, where method is the name of the specified method as given in HeatmapMethod. In this nested dictionary the heatmap is saved under the key heatmap_image_class_classID, in case of *'guided_grad_cam'* the key is guided_grad_cam_image_class_classID, where classID is the ID of the target class.

Attention

System requirements: This operator performs a backward pass. As a consequence the same system requirements apply as for the training of deep-learning-based models. For further details, please refer to the "Installation Guide", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods".

The heatmap should be used as a tool for visualizing and better understanding classification results. It is not intended as a segmentation tool. Moreover, gen_dl_model_heatmap currently only supports models with *'type'='classification'*.

Parameters

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Handle of a Deep learning model.

▷ **DLSample** (input_control) ..... dict(-array) ⇝ *handle*
Dictionaries with the sample input data.

▷ **HeatmapMethod** (input_control) ..... string ⇝ *string*
Method to be used for the heatmap calculation.
Default: 'grad_cam'
List of values: `HeatmapMethod` ∈ {'grad_cam', 'guided_grad_cam'}

▷ **TargetClasses** (input_control) ..... tuple ⇝ *integer*
ID of the target class.
Default: []

▷ **GenParam** (input_control) ..... dict ⇝ *handle*
Dictionary for generic parameters.
Default: []

▷ **DLResult** (output_control) ..... dict(-array) ⇝ *handle*
Dictionaries with the result data.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`apply_dl_model`

References

R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, D. Batra: "Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization," 2017 IEEE International Conference on Computer Vision (ICCV), Venice, 2017, pp. 618-626. doi: 10.1109/ICCV.2017.74

Module

Deep Learning Enhanced

gen_dl_pruned_model (: : DLModelHandleToPrune, DLPruningHandle : DLModelHandlePruned)

Prune a deep learning model.

The operator `gen_dl_pruned_model` copies the input model `DLModelHandleToPrune` and returns the pruned copy in `DLModelHandlePruned`. Note that `gen_dl_pruned_model` supports only deep learning models of *'type'='classification'*.

The parameter `DLPruningHandle` provides the necessary information as e.g., the selected pruning mode, the percentage or the kernel scores. See `create_dl_pruning` for further information about implemented pruning modes.

Behavior in special cases:

- In case the set amount of pruning *'percentage'* is too small to result in any removals, the returned model `DLModelHandlePruned` is simply a copy of the input model `DLModelHandleToPrune`.
- In case the model could not be pruned successfully, an empty handle is returned.

Parameters

▷ **DLModelHandleToPrune** (input_control) . . . . . dl_model ⇝ *handle*
Input model.

▷ **DLPruningHandle** (input_control) . . . . . dl_pruning ⇝ *handle*
Pruning data handle.

▷ **DLModelHandlePruned** (output_control) . . . . . dl_model ⇝ *handle*
Pruned model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_dl_pruning_batch, set_dl_pruning_param

Module

Deep Learning Professional

get_dl_model_param (: : DLModelHandle,
GenParamName : GenParamValue)

Return the parameters of a deep learning model.

get_dl_model_param returns the parameter values GenParamValue of GenParamName for the deep learning model DLModelHandle.

For a deep learning model, parameters GenParamName can be set using set_dl_model_param or create_dl_model_detection, depending on the parameter and the model type. With this operator, get_dl_model_param, you can retrieve the parameter values GenParamValue. Below we give an overview of the different parameters and an explanation, except of those you can only set. For latter ones, please see the documentation of corresponding operator. The parameters are listed corresponding to the deep learning model methods:

To mark which operators are available for the methods, we use the following notations:

- set: The parameter can be set using set_dl_model_param.
- get: The parameter can be retrieved using get_dl_model_param.
- create: The parameter can be set using create_dl_model_detection.
- Certain parameters are set as non-optional parameters, the corresponding notation is given in brackets.

3D Gripping Point Detection

GenParamName	set	get
'adam_beta1'	x	x
'adam_beta2'	x	x
'adam_epsilon'	x	x
'batch_size'	x	x
'batch_size_multiplier'	x	x
'batchnorm_momentum'	x	
'class_ids'	x	x
'class_names'		x
'device'	x	x
'enable_resizing'	x	
'fuse_bn_relu'	x	
'fuse_conv_relu'	x	
'gpu'	x	x
'image_dimensions'	x	x
'image_height', *'image_width'*	x	x
'image_num_channels'		x
'image_range_max', *'image_range_min'*	x	x
'image_size'	x	x
'input_dimensions'		x
'learning_rate'	x	x
'meta_data'	x	x
'min_version'		x
'momentum'	x	x
'num_trainable_params'		x
'optimize_for_inference'	x	x
'precision'		x
'precision_is_converted'		x
'runtime'	x	x
'runtime_init'	x	
'solver_type'	x	x
'type'		x
'weight_prior'	x	x

Anomaly Detection

GenParamName	set	get
'batch_size'	x	x
'batchnorm_momentum'	x	
'complexity'	x	x
'device'	x	x
'enable_resizing'	x	
'fuse_bn_relu'	x	
'fuse_conv_relu'	x	
'gpu'	x	x
'image_dimensions'	x	x
'image_height', *'image_width'*	x	x
'image_num_channels'		x
'image_range_max', *'image_range_min'*		x
'image_size'	x	x
'input_dimensions'	x	x
'meta_data'	x	x
'min_version'		x
'num_trainable_params'		x
'precision'		x
'precision_is_converted'		x
'runtime'	x	x
'runtime_init'	x	
'standard_deviation_factor'	x	x
'type'		x

Classification

GenParamName	set	get
'adam_beta1'	x	x
'adam_beta2'	x	x
'adam_epsilon'	x	x
'batch_size'	x	x
'batch_size_multiplier'	x	x
'batchnorm_momentum'	x	
'class_ids'		x
'class_names'	x	x
'class_weights'	x	x
'clear_ood'	x	
'device'	x	x
'enable_resizing'	x	
'extract_feature_maps'	x	x
'fuse_bn_relu'	x	
'fuse_conv_relu'	x	
'gpu'	x	x
'image_dimensions'	x	x
'image_height', *'image_width'*	x	x
'image_num_channels'	x	x
'image_range_max', *'image_range_min'*		x
'image_size'	x	x
'input_dimensions'	x	x
'layer_names'		x
'learning_rate'	x	x
'meta_data'	x	x
'min_version'		x
'momentum'	x	x
'num_trainable_params'		x
'ood_available'		x
'ood_threshold'	x	x
'optimize_for_inference'	x	x
'precision'		x
'precision_is_converted'		x
'runtime'	x	x
'runtime_init'	x	
'solver_type'	x	x
'summary'		x
'type'		x
'weight_prior'	x	x

Multi-Label Classification

GenParamName	set	get
'batch_size'	x	x
'batch_size_multiplier'	x	x
'class_ids'	x	x
'class_names'	x	x
'device'	x	x
'gpu'	x	x
'image_dimensions'	x	x
'image_height', *'image_width'*	x	x
'image_num_channels'	x	x
'image_range_max', *'image_range_min'*		x
'image_size'	x	x
'input_dimensions'	x	x
'layer_names'		x
'learning_rate'	x	x
'meta_data'	x	x
'min_version'		x
'min_confidence'	x	x
'momentum'	x	x
'num_trainable_params'		x
'optimize_for_inference'	x	x
'precision'		x
'precision_is_converted'		x
'runtime'	x	x
'runtime_init'	x	
'solver_type'	x	x
'summary'		x
'type'		x
'weight_prior'	x	x

Deep Counting component

GenParamName	set	get
'batch_size'	x	x
'batch_size_multiplier'	x	x
'device'	x	x
'gpu'	x	x
'input_dimensions'		x
'layer_names'		x
'learning_rate'	x	x
'meta_data'	x	x
'min_version'		x
'momentum'	x	x
'num_trainable_params'		x
'optimize_for_inference'		x
'precision'		x
'precision_is_converted'		x
'runtime'	x	x
'solver_type'		x
'summary'		x
'type'		x
'weight_prior'	x	x

Deep OCR detection component

GenParamName	set	get
'adam_beta1'	x	x
'adam_beta2'	x	x
'adam_epsilon'	x	x
'batch_size'	x	x
'batchnorm_momentum'	x	
'device'	x	x
'enable_resizing'	x	
'fuse_bn_relu'	x	
'fuse_conv_relu'	x	
'gpu'	x	x
'image_dimensions'	x	x
'image_height', *'image_width'*	x	x
'image_num_channels'	x	x
'image_range_max', *'image_range_min'*	x	x
'image_size'	x	x
'input_dimensions'	x	x
'layer_names'		x
'learning_rate'	x	x
'meta_data'	x	x
'min_character_score'	x	x
'min_link_score'	x	x
'min_version'		x
'min_word_area'	x	x
'min_word_score'	x	x
'momentum'	x	x
'num_trainable_params'		x
'orientation'	x	x
'optimize_for_inference'	x	x
'precision'		x
'precision_is_converted'		x
'runtime'	x	x
'runtime_init'	x	
'solver_type'	x	x
'sort_by_line'	x	x
'summary'		x
'tiling'	x	x
'tiling_overlap'	x	x
'type'		x
'weight_prior'	x	x

Deep OCR recognition component

GenParamName	set	get
'adam_beta1'	x	x
'adam_beta2'	x	x
'adam_epsilon'	x	x
'alphabet'	x	x
'alphabet_internal'	x	x
'alphabet_mapping'	x	x
'batch_size'	x	x
'batchnorm_momentum'	x	
'device'	x	x
'enable_resizing'	x	
'fuse_bn_relu'	x	
'fuse_conv_relu'	x	
'gpu'	x	x
'image_dimensions'	x	x
'image_height', *'image_width'*	x	x
'image_num_channels'	x	x
'image_range_max', *'image_range_min'*	x	x
'image_size'	x	x
'input_dimensions'	x	x
'layer_names'		x
'learning_rate'	x	x
'meta_data'	x	x
'min_version'		x
'momentum'	x	x
'num_trainable_params'		x
'optimize_for_inference'	x	x
'precision'		x
'precision_is_converted'		x
'runtime'	x	x
'runtime_init'	x	
'solver_type'	x	x
'summary'		x
'type'		x
'weight_prior'	x	x

Global Context Anomaly Detection

GenParamName	set	get
'adam_beta1'	x	x
'adam_beta2'	x	x
'adam_epsilon'	x	x
'anomaly_score_tolerance'	x	x
'batch_size'	x	x
'batch_size_multiplier'	x	x
'batchnorm_momentum'	x	
'device'	x	x
'enable_resizing'	x	
'fuse_bn_relu'	x	
'fuse_conv_relu'	x	
'gc_anomaly_networks'	x	x
'gpu'	x	x
'image_dimensions'	x	x
'image_height', *'image_width'*	x	x
'image_num_channels'	x	x
'image_range_max', *'image_range_min'*		x
'image_size'	x	x
'input_dimensions'		x
'layer_names'		x
'learning_rate'	x	x
'meta_data'	x	x
'min_version'		x
'momentum'	x	x
'num_trainable_params'		x
'optimize_for_inference'	x	x
'patch_size'	x	x
'precision'		x
'precision_is_converted'		x
'runtime'	x	x
'runtime_init'	x	
'solver_type'	x	x
'summary'		x
'type'		x
'weight_prior'	x	x

Object Detection, Instance Segmentation

GenParamName	set	get	create
'adam_beta1'	x	x	
'adam_beta2'	x	x	
'adam_epsilon'	x	x	
'anchor_angles'		x	x
'anchor_aspect_ratios'		x	x
'anchor_num_subscales'		x	x
'backbone' (Backbone)		x	x
'backbone_docking_layers'	x	x	x
'batch_size'	x	x	
'batch_size_multiplier'	x	x	
'batchnorm_momentum'	x		
'bbox_heads_weight', *'class_heads_weight'*	x	x	x
'capacity'		x	x
'class_ids'	x	x	x
'class_ids_no_orientation'		x	x
'class_names'	x	x	x
'class_weights'		x	
'device'	x	x	
'enable_resizing'	x		
'freeze_backbone_level'	x	x	x
'fuse_bn_relu'	x		
'fuse_conv_relu'	x		
'gpu'	x	x	
'ignore_direction'		x	x
'image_dimensions'		x	x
'image_height', *'image_width'*		x	x
'image_num_channels'		x	x
'image_range_max', *'image_range_min'*		x	
'image_size'		x	x
'input_dimensions'		x	
'instance_segmentation'		x	x
'instance_type'		x	x
'layer_names'		x	
'learning_rate'	x	x	
'mask_head_weight'	x	x	x
'max_level', *'min_level'*		x	x
'max_num_detections'	x	x	x
'max_overlap'	x	x	x
'max_overlap_class_agnostic'	x	x	x
'meta_data'	x	x	
'min_confidence'	x	x	x

'min_version'		x	
'momentum'	x	x	
'num_classes' (NumClasses)		x	x
'num_trainable_params'		x	
'optimize_for_inference'	x	x	x
'precision'		x	
'precision_is_converted'		x	
'runtime'	x	x	
'runtime_init'	x		
'solver_type'	x	x	
'summary'		x	
'type'		x	
'weight_prior'	x	x	

Semantic Segmentation, Edge Extraction

GenParamName	set	get
'adam_beta1'	x	x
'adam_beta2'	x	x
'adam_epsilon'	x	x
'batch_size'	x	x
'batch_size_multiplier'	x	x
'batchnorm_momentum'	x	
'class_ids'	x	x
'class_names'	x	x
'device'	x	x
'enable_resizing'	x	
'fuse_bn_relu'	x	
'fuse_conv_relu'	x	
'gpu'	x	x
'ignore_class_ids'	x	x
'image_dimensions'	x	x
'image_height', *'image_width'*	x	x
'image_num_channels'	x	x
'image_range_max', *'image_range_min'*	x	x
'image_size'	x	x
'input_dimensions'	x	x
'layer_names'		x
'learning_rate'	x	x
'meta_data'	x	x
'min_version'		x
'momentum'	x	x
'num_classes'		x
'num_trainable_params'		x
'optimize_for_inference'	x	x
'precision'		x
'precision_is_converted'		x
'runtime'	x	x
'runtime_init'	x	
'solver_type'	x	x
'summary'		x
'type'		x
'weight_prior'	x	x

In the following we list and explain the parameters `GenParamName` for which you can retrieve their value using this operator, `get_dl_model_param`. Note, that only parameters that do not change the model architecture can be set after the model has been optimized with `optimize_dl_model_for_inference`. A list of these parameters can be found at `optimize_dl_model_for_inference`. Thereby, the following symbols denote the model type for which the parameter can be get:

- 'Any': any method
- '3D-GPD': *'type'='3d_gripping_point_detection'*
- 'AD': *'type'='anomaly_detection'*

- 'CL': *'type'='classification'*
- 'MLC': *'type'='multi_label_classification'*
- 'DC': *'type'='counting'*
- 'OCR-D': *'type'='ocr_detection'*
- 'OCR-R': *'type'='ocr_recognition'*
- 'GC-AD': *'type'='gc_anomaly_detection'*
- 'OD': *'type'='detection'*
- 'SE': *'type'='segmentation'*
- 'Gen': *'type'='generic'*

'adam_beta1': **3D-GPD CL OCR-D OCR-R GC-AD OD SE Gen**

This value defines the moment for the linear term in Adam solver. For more information we refer to the documentation of `train_dl_model_batch`.

Restriction: Only applicable for *'solver_type'* = *'adam'*.

Default: *'adam_beta1'* = *0.9*

'adam_beta2': **3D-GPD CL OCR-D OCR-R GC-AD OD SE Gen**

This value defines the moment for the quadratic term in Adam solver. For more information we refer to the documentation of `train_dl_model_batch`.

Restriction: Only applicable for *'solver_type'* = *'adam'*.

Default: *'adam_beta2'* = *0.999*

'adam_epsilon': **3D-GPD CL OCR-D OCR-R GC-AD OD SE Gen**

This value defines the epsilon in the Adam solver formula and is purposed to guarantee the numeric stability. For more information we refer to the documentation of `train_dl_model_batch`.

Restriction: Only applicable for *'solver_type'* = *'adam'*.

Default: *'adam_epsilon'* = *1e-08*

'alphabet': **OCR-R**

The character set that can be recognized by the Deep OCR model.

It contains all characters that are not mapped to the Blank character of the internal alphabet (see parameters *'alphabet_mapping'* and *'alphabet_internal'*).

The alphabet can be changed or extended if needed. Changing the alphabet with this parameter will edit the internal alphabet and mapping in such a way that it tries to keep the length of the internal alphabet unchanged. After changing the alphabet, it is recommended to retrain the model on application specific data (see the HDevelop example `deep_ocr_recognition_training_workflow.hdev`). Previously unknown characters will need more training data.

Note, that if the length of the internal alphabet changes, the last model layers have to be randomly initialized and thus the output of the model will be random strings (see *'alphabet_internal'*). In that case it is required to retrain the model.

'alphabet_internal': **OCR-R**

The full character set which the Deep OCR recognition component has been trained on.

The first character of the internal alphabet is a special character. In the pretrained model this character is specified as Blank (U+2800) and is not to be confused with a space. The Blank is never returned in a word output but can occur in the reported character candidates. It is required and cannot be omitted. If the internal alphabet is changed, the first character has to be the Blank. Furthermore, if *'alphabet'* is used to change the alphabet, the Blank symbol is added automatically to the character set.

The length of this tuple corresponds to the depth of the last convolution layer in the model. If the length changes, the last convolution layer and all layers after it have to be resized and potentially reinitialized randomly. After such a change, it is required to retrain the model (see HDevelop example `deep_ocr_recognition_training_workflow.hdev`).

It is recommended to use the parameter *'alphabet'* to change the alphabet, as it will automatically try to preserve the length of the internal alphabet.

'alphabet_mapping': **OCR-R**

Tuple of integer indices.

It is a mapping that is applied by the model during the decoding step of each word. The mapping overwrites a character of *'alphabet_internal'* with the character at the specified index in *'alphabet_internal'*.

In the decoding step each prediction is mapped according to the index specified in this tuple. The tuple has to be of same length as the tuple *'alphabet_internal'*. Each integer index of the mapping has to be within *0* and |*'alphabet_internal'*|-1.

In some applications it can be helpful to map certain characters onto other characters. E.g. if only numeric words occur in an application it might be helpful to map the character "O" to the "0" character without the need to retrain the model.

If an entry contains a *0*, the corresponding character in *'alphabet_internal'* will not be decoded in the word.

'anchor_angles': **OD**

The parameter *'anchor_angles'* determines the orientation angle of the anchors for a model of *'instance_type'* = *'rectangle2'*.

Thereby, the orientation is given in arc measure and indicates the angle between the horizontal axis and `Length1` (mathematically positive). See the chapter Deep Learning / Object Detection and Instance Segmentation for more explanations to anchors.

You can set a tuple of values. A higher number of angles increases the number of anchors which might lead to a better localization but also increases the runtime and memory-consumption.

Value range:

'anchor_angles' $\in (-\pi, \pi]$ for *'ignore_direction'* = *'false'*, *'anchor_angles'* $\in (-\pi/2, \pi/2]$ for *'ignore_direction'* = *'true'*

Default: *'anchor_angles'* = *[0.0]*

'anchor_aspect_ratios' **(legacy:** *'aspect_ratios'***): OD**

The parameter *'anchor_aspect_ratios'* determines the aspect ratio of the anchors. Thereby, the definition of the ratio depends on the *'instance_type'*:

- *'rectangle1'*: height-to-width ratio
- *'rectangle2'*: ratio length1 to length2

E.g., for instance type *'rectangle1'* the ratio 2 gives a narrow and 0.5 a broad anchor. The size of the anchor is affected by the parameter *'anchor_num_subscales'* and with its explanation we give the formula for the sizes and lengths of the generated anchors. See the chapter Deep Learning / Object Detection and Instance Segmentation for more explanations to anchors.

You can set a tuple of values. A higher number of aspect ratios increases the number of anchors which might lead to a better localization but also increases the runtime and memory-consumption.

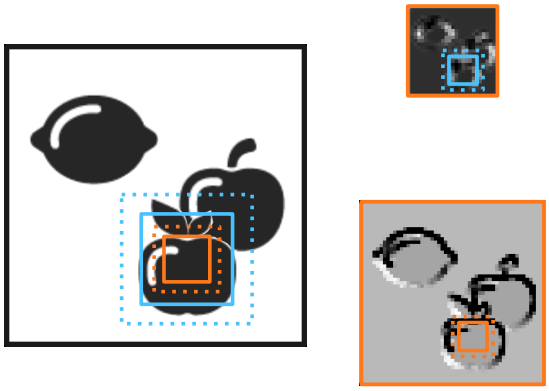
For reasons of backward compatibility, the parameter name *'aspect_ratios'* can be used instead of *'anchor_aspect_ratios'*.

Default: *'anchor_aspect_ratios'* = *[1.0, 2.0, 0.5]*

'anchor_num_subscales' **(legacy:** *'num_subscales'***): OD**

This parameter determines the number of different sizes with which the anchors are generated at the different levels used.

In HALCON for every anchor point, thus every pixel of every feature map of the feature pyramid, a set of anchors is proposed. See the chapter Deep Learning / Object Detection and Instance Segmentation for more explanations to anchors. Thereby the parameter *'anchor_num_subscales'* affects the size of the anchors. An example is shown in the figure below.



With *'anchor_num_subscales'*=2 we generate for every aspect ratio 2 anchors of different size on each level: One with the base length (solid line) and an additional, larger one (dotted line). Thereby, in the image these additional anchors of the lower level (orange) converge to the anchor of the next higher level (blue).

An anchor of level l has by default a edge lengths of $b \cdot 2^l$ in the input image, whereby the parameter has the value $b = 4$. With the parameter *'anchor_num_subscales'* additional anchors can be generated, which converge in size to the smallest anchor of the level $l + 1$. More precisely, these anchors of level l have in the input image the edge lengths

$$scale_{li} = b \cdot 2^{l+i/'anchor\_num\_subscales'}$$

where $i \in [0, 'anchor\_num\_subscales' - 1]$. For subscale i, this results on level l in an anchor of height h and width w equal

$$h_{lij} = scale_{li} \cdot \sqrt{r_j}, \qquad w_{lij} = scale_{li} \cdot \frac{1}{\sqrt{r_j}}$$

where r_j is the ratio of this anchor (see *'anchor_aspect_ratios'*).

A larger number of subscales increases the number of anchors and will therefore increase the runtime and memory-consumption.

For reasons of backward compatibility, the parameter name *'num_subscales'* can be used instead of *'anchor_num_subscales'*.

Default: *'anchor_num_subscales' = 3*

'anomaly_score_tolerance': **GC-AD**

The image-level anomaly score is calculated internally such that a certain fraction of the pixel-level anomaly scores in `anomaly_image` is greater or equal to the image-level anomaly score. The value of this fraction can be set by *'anomaly_score_tolerance'* and has to be in the interval *[0.0, 1.0]*. For example, for *'anomaly_score_tolerance'=0.01*, 1 percent of the pixel anomaly scores are larger or equal to the image anomaly score. This can be used to suppress outliers that might appear in the pixel anomaly scores.

Default: *'anomaly_score_tolerance' = 0.0*

'backbone': **OD**

The parameter *'backbone'* is the name (together with the path) of the backbone network which is used to create the model. A list of the delivered backbone networks can be found under `create_dl_model_detection`.

'backbone_docking_layers': **CL**

The parameter *'backbone_docking_layers'* specifies which layers of the backbone are to be used as docking layers by the feature pyramid. Thereby the layers are referenced by their names.

The docking layers can be specified for every classifier, also without using them as backbone. The specification is only considered for object detection backbones. When selecting the docking layers, consider that the feature map lengths have to be halved from one docking layer to the other. Rule of thumb: Use the deepest layers for every (lateral) resolution in the backbone (corresponding to one of the required levels for your object detection task).

Information about the names and sizes of the layers in a model can be enquired using *'summary'*.

Default: For the pretrained backbones delivered by HALCON the defaults depend on the classifier. Other classifiers do not have any docking layers set by default and therefore need to have this parameter set before they can be used as backbone.

'batch_size': **Any**

Number of input images (and corresponding labels) in a batch that is transferred to device memory.

For a training using `train_dl_model_batch`, the batch of images which are processed simultaneously in a single training iteration contains a number of images which is equal to *'batch_size'* times *'batch_size_multiplier'*. Please refer to `train_dl_model_batch` for further details.

For inference, the *'batch_size'* can be generally set independently from the number of input images. See `apply_dl_model` for details on how to set this parameter for greater efficiency.

Models of *'type'*='*classification*':

The parameter *'batch_size'* is stored in the pretrained classifier. Per default, the *'batch_size'* is set such that a training of the pretrained classifier with up to `100` classes can be easily performed on a device with `8` gigabyte of memory.

For the pretrained classifiers, the default values are hence given as follows:

pretrained classifier	default value of *'batch_size'*
'pretrained_dl_classifier_alexnet.hdl'	*230*
'pretrained_dl_classifier_compact.hdl'	*160*
'pretrained_dl_classifier_enhanced.hdl'	*96*
'pretrained_dl_classifier_mobilenet_v2.hdl'	*40*
'pretrained_dl_classifier_resnet18.hdl'	*24*
'pretrained_dl_classifier_resnet50.hdl'	*23*

Models of *'type'='counting'***:** The parameter *'batch_size'* has no effect.

*'batch_size_multiplier'***: Any**

Multiplier for *'batch_size'* to enable training with larger numbers of images in one step which would otherwise not be possible due to GPU memory limitations. This model parameter does only affect `train_dl_model_batch` and thus has no impact during evaluation and inference. For detailed information see `train_dl_model_batch`.

Models of *'type'='anomaly_detection'***:** The parameter *'batch_size_multiplier'* has no effect.

Models of *'type'='counting'***:** The parameter *'batch_size_multiplier'* has no effect.

Models of *'type'='ocr_recognition'***:** The parameter *'batch_size_multiplier'* has no effect.

Default: *'batch_size_multiplier' = 1*

'bbox_heads_weight', *'class_heads_weight'***: OD**

The parameters *'bbox_heads_weight'* and *'class_heads_weight'* are weighting factors for the calculation of the total loss. This means, when the losses of the individual networks are summed up, the contributions from the bounding box regression heads are weighted by a factor *'bbox_heads_weight'* and the contributions from the classification heads are weighted by a factor *'class_heads_weight'*.

Default: *'bbox_heads_weight' = 1.0*, *'class_heads_weight' = 1.0*

*'capacity'***: OD**

This parameter roughly determines the number of parameters (or filter weights) in the deeper sections of the object detection network (after the backbone). Its possible values are *'high'*, *'medium'*, and *'low'*.

It can be used to trade-off between detection performance and speed. For simpler object detection tasks, the *'low'* or *'medium'* settings may be sufficient to achieve the same detection performance as with *'high'*.

Default: *'capacity' = 'high'*

*'class_ids'***: 3D-GPD CL MLC OD SE**

Unique IDs of the classes the model shall distinguish. The tuple is of length *'num_classes'*.

We stress out the slightly different meanings and restrictions depending on the model type:

Models of *'type'='3d_gripping_point_detection'***:**

Two classes, in fixed order [*'gripping_map'*, *'background'*], are supported for this model. Therefore, for such a model the tuple has a fixed length of 2. Thereby, you can set any integer within the interval $[0, 65534]$ as class ID value.

Models of *'type'='classification'***:**

The IDs are unique identifiers, which are automatically assigned to each class. The ID of a class corresponds to the index within the tuple *'class_names'*.

Models of *'type'='multi_label_classification'***:**

The IDs are unique identifiers of the classes to be detected. Thereby, you can set any integer as class ID value.

Value range: $[0, 65534]$.

Models of *'type'='detection'***:**

Only the classes of the objects to be detected are included and therewith no background class. Thereby, you can set any integer within the interval $[0, 65534]$ as class ID value.

Note that the values of *'class_ids_no_orientation'* depend on *'class_ids'*. Thus if *'class_ids'* is changed after the creation of the model, *'class_ids_no_orientation'* is reset to an empty tuple.

Default: *'class_ids' = '[0,...,num_classes-1]'*

Models of *'type'='segmentation'***:**

Every class used for training has to be included and therewith also the class ID of the 'background' class. Therefore, for such a model the tuple has a minimal length of 2. Thereby, you can set any integer within the interval $[0, 65534]$ as class ID value.

'class_ids_no_orientation': **OD**

With this parameter you can declare classes, for which the orientation will not be considered, e.g., round or other point symmetrical objects. For each class, whose class ID is present in *'class_ids_no_orientation'*, the network returns axis-aligned bounding boxes.

Note, this parameter only affects networks of *'instance_type'* = *'rectangle2'*.

Note that the values of *'class_ids_no_orientation'* depend on *'class_ids'*. Thus if *'class_ids'* is changed after the creation of the model, *'class_ids_no_orientation'* is reset to an empty tuple.

Default: *'class_ids_no_orientation'* = *[]*

'class_names': **3D-GPD CL MLC OD SE**

Unique names of the classes the model shall distinguish. The order of the class names remains unchanged after the setting. The tuple is of length *'num_classes'*.

Restriction: For a *'3d_gripping_point_detection'* model, the *'class_names'* tuple has a fixed length of 2, with constant value [*'gripping_map'*, *'background'*].

'class_weights': **CL OD**

The parameter *'class_weights'* is a tuple of class specific weighting factors for the loss. Giving the unique classes a different weight, it is possible to force the network to learn the classes with different importance. This is useful in cases where a class dominates the dataset. The weighting factors have to be within the interval $(0, 1)$. Thereby a class gets a stronger impact during the training the larger its weight is. The weights in the tuple *'class_weights'* are sorted the same way as the classes in the tuple *'class_ids'*. We stress out the slightly different meanings and restrictions depending on the model type:

Models of *'type'*=*'classification'*:
Default: *'class_weights'* = *1.0* for each class.

Models of *'type'*=*'detection'*:
Default: *'class_weights'* = *0.25* for each class.

'clear_ood': **CL**

This parameter deactivates the out-of-distribution detection of a deep learning model. All related information will be deleted. `fit_dl_out_of_distribution` can be called to fit the out-of-distribution detection again.

Default: *'true'*

'complexity': **AD**

This parameter controls the capacity of the model to deal with more complex applications. A higher value allows for the model to represent images showing more complexity. Increasing the parameter leads to higher runtimes during training and inference. Please note that this parameter can only be set before the model is trained. Setting *'complexity'* on an already trained model would render this model useless. When trying to do so, an error is returned but the model itself is unchanged.

Default: *'complexity'* = *15*

'device': **Any**

Handle of the device on which the deep learning operators will be executed.

If the model was already optimized for a device, setting *'device'* might not be necessary anymore, see `optimize_dl_model_for_inference` for details.

To get a tuple of handles of all available potentially deep-learning capable hardware devices use `query_available_dl_devices`.

Default: Handle of the default device, thus the GPU with index *0*. If not available, this is an empty tuple.

'extract_feature_maps': **CL**

With this parameter value you can extract feature maps of the specified model layer for an inferred image. The selected layer must be part of the existing network. An overview of all existing layers of the model can be returned by the operator `get_dl_model_param` with the corresponding parameter *'summary'*.

Note, using this option modifies the network architecture: The network is truncated after a selected layer. This modification can not be reversed. If the original network architecture should be used again it must be read in again with the operator `read_dl_model`.

'freeze_backbone_level': **OD**

This parameter determines the backbone levels whose weights are kept (meaning not updated and thus frozen) during training. Thereby the given number signifies the highest level whose layers are frozen in the backbone. Setting *'freeze_backbone_level'* to *0*, for no level the weights are frozen and as a consequence the weights of

all layers are updated. It is recommended to set this in case the weights have been randomly initialized (e.g., after certain changes of the number of image channels) or the in case the backbone is not pretrained.

Default: *'freeze_backbone_level' = 2*

'gc_anomaly_networks': **GC-AD**

This parameter is used to select the subnetworks of the model. The following values can be set:

- *'local'*: The local subnetwork will be extracted.
- *'global'*: The global subnetwork will be extracted.

We refer to Deep Learning / Anomaly Detection and Global Context Anomaly Detection for more general information on the subnetworks and their properties.

Note, the original model contains both a *'local'* and a *'global'* subnetwork. Once the model architecture is reduced to a single subnetwork, the original network is truncated. This cannot be undone. If the original model architecture is to be used again, it must be read in again with `read_dl_model`.

Default: *'gc_anomaly_networks' = ['local', 'global']*

'gpu': **Any**

Identifier of the GPU where the training and inference operators (`train_dl_model_batch` and `apply_dl_model`) are executed. Per default, the first available GPU is used. `get_system` with *'cuda_devices'* can be used to retrieve a list of available GPUs. Pass the index in this list to *'gpu'*.

Note that the parameter *'gpu'* is only taken into account for *'runtime' = 'gpu'*. Therefore, it is preferable to set the GPU device, on which operators are run, using the parameter *'device'*. executed.

Default: *'gpu' = 0*

'ignore_class_ids': **SE**

With this parameter you can declare one or multiple classes as 'ignore' classes, see the chapter Deep Learning / Semantic Segmentation and Edge Extraction for further information. These classes are declared over their ID (integers).

Note, you can not set a class ID in *'ignore_class_ids'* and *'class_ids'* simultaneously.

'ignore_direction': **OD**

This parameter determines whether for the oriented bounding box also the direction of the object within the bounding box is considered or not. In case the direction within the bounding box is not to be considered you can set *'ignore_direction'* to *'true'*. In order to determine the bounding box unambiguously, in this case (but only in this case) the following conventions apply:

- $-\pi/2 <$ *'phi'* $\leq \pi/2$
- *'bbox_length1' > 'bbox_length2'*

This is consistent to `smallest_rectangle2`.

Note, this parameter only affects networks of *'instance_type' = 'rectangle2'*.

List of values: *'true'*, *'false'*

Default: *'ignore_direction' = 'false'*

'image_dimensions': **3D-GPD AD CL MLC OCR-D OCR-R GC-AD OD SE**

Tuple containing the input image dimensions *'image_width'*, *'image_height'*, and number of channels *'image_num_channels'*.

The respective default values and possible value ranges depend on the model and model type. Please see the individual dimension parameter description for more details.

'image_height', *'image_width'*: **3D-GPD AD CL MLC OCR-D OCR-R GC-AD OD SE**

Height and width of the input images, respectively, that the network will process.

This parameter can attain different values depending on the model type:

Models of *'type'='3d_gripping_point_detection'*:

The network architectures allow changes of the image height and width.
The default values are given by the network, see `read_dl_model`.

Models of *'type'='anomaly_detection'*:

The default values depend on the specific pretrained network, see `read_dl_model`. The network architectures allow changes of the image dimensions, which can be done using `set_dl_model_param`. Please also refer to `read_dl_model` for restrictions each of the delivered networks has on the input image size. Note that these parameters have to be set before training the model. Setting them on an already trained model would render this model useless. When trying to do so, an error is returned but the model itself is unchanged.

Models of *'type'='classification'*:
The default values depend on the specific pretrained classifier, see `read_dl_model`. The network architectures allow changes of the image dimensions, which can be done using `set_dl_model_param`. But for networks with at least one fully connected layer such a change makes a retraining necessary. Networks without fully connected layers are directly applicable to different image sizes. However, images with a size differing from the size with which the classifier has been trained are likely to show a reduced classification accuracy.

Models of *'type'='detection'*:
The network architectures allow changes of the image dimensions. But the image lengths are halved for every level, that is why the dimensions *'image_width'* and *'image_height'* need to be an integer multiple of 2^{level}. $level$ depends on the *'backbone'* and the parameter *'max_level'*, see `create_dl_model_detection` for further information.
Default: *'image_height'* = *640*, *'image_width'* = *640*

Models of *'type'='gc_anomaly_detection'*:
The network architecture allows changes of the image dimensions. Note that these parameters have to be set before training the model. Setting them on an already trained model would render this model useless.
Restriction: The *'image_width'* and the *'image_height'* must be greater than or equal to the value of the *'patch_size'* parameter.
Default: *'image_height'* = *256*, *'image_width'* = *256*

Models of *'type'='ocr_recognition'*:
The network architectures allow changes of the image width.
The default and minimal values are given by the network, see `read_dl_model`.

Models of *'type'='segmentation'*:
The network architectures allow changes of the image dimensions.
The default and minimal values are given by the network, see `read_dl_model`.

'image_num_channels': **3D-GPD AD CL MLC OCR-D OCR-R GC-AD OD SE**
Number of channels of the input images the network will process. The default value is given by the network, see `read_dl_model` and `create_dl_model_detection`.

For models of *'type'='anomaly_detection'* or *'type'='gc_anomaly_detection'*, only the values *1* and *3* are supported. In addition, this parameter should be set before the model is trained. For models of *'type'='anomaly_detection'*, setting *'image_num_channels'* on an already trained model would render this model useless. Therefore, an error is returned when trying to do so, but the model itself is unchanged.

Restriction: For models of *'type'='3d_gripping_point_detection'* the parameter *'image_num_channels'* cannot be set.

For other models, any number of input image channels is possible.

If number of channels is changed to a value >1, the weights of the first layers after the input image layer will be initialized with random values. Note, in this case more data for the retraining is needed. If the number of channels is changed to 1, the weights of the concerned layers are fused.

Models of *'type'='anomaly_detection'*:
Default: *'image_num_channels'* = *3*

Models of *'type'='detection'*:
Default: *'image_num_channels'* = *3*

'image_range_max', *'image_range_min'*: **3D-GPD AD CL MLC OCR-D OCR-R GC-AD OD SE**
Maximum and minimum gray value of the input images, respectively, the network will process.

The default values are given by the network, see `read_dl_model` and `create_dl_model_detection`.

'image_size': **3D-GPD AD CL MLC OCR-D OCR-R GC-AD OD SE**
Tuple containing the input image size *'image_width'*, *'image_height'*.

The respective default values and possible value ranges depend on the model and model type. Please see the individual dimension parameter description for more details.

'input_dimensions': **Any**
This parameter returns a dictionary containing all input dimensions of the network. Examples for such inputs: input image, `weight_image` (for models of *'type'='segmentation'*).

These dimensions are given in the dictionary as a tuple [`width`, `height`, `depth`]. In case this parameter is used to set the dimension, for every dimension a value of -1 may be set to keep the current value.

'instance_segmentation': **OD**

This parameter determines if the model is created for instance segmentation. If the parameter is set to *'true'* in `create_dl_model_detection`, the detection deep learning network is extended by additional layers for instance segmentation.

List of values: *'true'*, *'false'*

Default: *'instance_segmentation'* = *'false'*

'instance_type': **OD**

The parameter *'instance_type'* determines, which instance type is used for the object model. The current implementations differ regarding the allowed orientations of the bounding boxes. See the chapter Deep Learning / Object Detection and Instance Segmentation for more explanations to the different types and their bounding boxes.

List of values: *'rectangle1'*, *'rectangle2'*

Default: *'instance_type'* = *'rectangle1'*

'layer_names': **Any except 3D-GPD**

This parameter returns a tuple containing the name for every layer of the model. This name is the same human-readable identifier as is returned by `get_dl_model_param` with *'summary'*.

Note, for some networks distributed with HALCON, the network architecture is confidential. In this case `get_dl_model_param` returns an empty tuple with *'layer_names'*.

'learning_rate': **Any**

Value of the factor determining the gradient influence during training using `train_dl_model_batch`. Please refer to `train_dl_model_batch` for further details.

The default values depend on the model. Note that changing *'solver_type'* sets *'learning_rate'* back to its default value.

Models of *'type'*=*'anomaly_detection'*:

The parameter *'learning_rate'* has no effect.

Models of *'type'*=*'counting'*:

The parameter *'learning_rate'* has no effect.

'mask_head_weight': **OD**

The parameter *'mask_head_weight'* is a weighting factor for the calculation of the total loss. This means, when the losses of the individual network heads are summed up, the contribution from the mask prediction head is weighted by a factor *'mask_head_weight'*.

Restriction: Only applicable to models with *'instance_segmentation'*=*'true'*

Default: *'mask_head_weight'* = *1.0*

'max_level', *'min_level'*: **OD**

These parameters determine on which levels the additional networks are attached on the feature pyramid. We refer to the chapter Deep Learning / Object Detection and Instance Segmentation for further explanations to the feature pyramid and the attached networks.

From these (*'max_level'* - *'min_level'* + 1) networks all predictions with a minimum confidence value are kept as long they do not strongly overlap (see *'min_confidence'* and *'max_overlap'*).

The level declares how often the size of the feature map already has been scaled down. Thus, level 0 corresponds to the feature maps with size of the input image, level 1 to feature maps subscaled once, and so on. As a consequence, smaller objects are detected in the lower levels, whereas larger objects are detected in higher levels.

The value for *'min_level'* needs to be at least 2.

If *'max_level'* is larger than the number of levels the backbone can provide, the backbone is extended with additional (randomly initialized) convolutional layers in order to generate deeper levels. Further, *'max_level'* may have an influence on the minimal input image size.

Note, for small input image dimensions, high levels might not be meaningful, as the feature maps could already be too small to contain meaningful information.

A higher number of used levels might increase the runtime and memory-consumption, whereby especially lower levels carry weight.

Default: *'max_level'* = *6*, *'min_level'* = *2*

'max_num_detections': **OD**

This parameter determines the maximum number of detections (bounding boxes) per image proposed from the network.

Default: *'max_num_detections'* = *100*

'max_overlap': **OD**

The maximum allowed intersection over union (IoU) for two predicted bounding boxes of the same class. Or, vice-versa, when two bounding boxes are classified into the same class and have an IoU higher than *'max_overlap'*, the one with lower confidence value gets suppressed. We refer to the chapter Deep Learning / Object Detection and Instance Segmentation for further explanations to the IoU.

Default: *'max_overlap' = 0.5*

'max_overlap_class_agnostic': **OD**

The maximum allowed intersection over union (IoU) for two predicted bounding boxes independently of their predicted classes. Or, vice-versa, when two bounding boxes have an IoU higher than *'max_overlap_class_agnostic'*, the one with lower confidence value gets suppressed. As default, *'max_overlap_class_agnostic'* is set to 1.0, hence class agnostic bounding box suppression has no influence.

Default: *'max_overlap_class_agnostic' = 1.0*

'meta_data': **Any**

Dictionary with user defined meta data, whose entries can be set freely. The meta data may be used to store information such as the model author or a model version along with the model.

Restriction: Dictionary values are limited to strings.

'min_character_score': **OCR-D**

The parameter *'min_character_score'* specifies the lower threshold used for the character score map to estimate the dimensions of the characters. By adjusting the parameter, suggested instances can be split up or neighboring instances can be merged.

Value range: $\in [0, 1]$.

Default: *0.5*

'min_confidence': **MLC OD**

Models of *'type'='detection'*: This parameter determines the minimum confidence, when the image part within the bounding box is classified in order to keep the proposed bounding box. This means, when `apply_dl_model` is called, all output bounding boxes with a confidence value smaller than *'min_confidence'* are suppressed.

Models of *'type'='multi_label_classification'*: This parameter determines the minimum confidence level required for a class to be considered as detected. This means, classes with a confidence score below *'min_confidence'* are regarded as not detected, while classes with a confidence value equal to or exceeding *'min_confidence'* are deemed as detected.

Default: *'min_confidence' = 0.5*

'min_link_score': **OCR-D**

The parameter *'min_link_score'* defines the minimum link score required between two localized characters to recognize these characters as coherent word.

Value range: $\in [0, 1]$.

Default: *0.3*

'min_version': **Any**

The parameter *'min_version'* defines the minimum required HALCON version to use this model.

'min_word_area': **OCR-D**

The parameter *'min_word_area'* defines the minimum size that a localized word must have in order to be suggested. This parameter can be used to filter suggestions that are too small.

Value range: ≥ 0.

Default: *10*.

'min_word_score': **OCR-D**

The parameter *'min_word_score'* defines the minimum score a localized instance must contain to be suggested as valid word. With this parameter uncertain words can be filtered out.

Value range: $\in [0, 1]$.

Default: *0.7*

'momentum': **Any**

When updating the weights of the network, the hyperparameter *'momentum'* specifies to which extent previous updating vectors will be added to the current updating vector. Only applicable for *'solver_type' = 'sgd'*.

Please refer to train_dl_model_batch for further details.

The default value is given by the model.

Models of *'type'='anomaly_detection'*: The parameter *'momentum'* has no effect.

Models of *'type'='counting'*: The parameter *'momentum'* has no effect.

'num_classes': **OD SE**

Number of distinct classes that the model is able to distinguish for its predictions.

This parameter differs between the model types:

Models of *'type'='detection'*:

This parameter is set as NumClasses over create_dl_model_detection. *'class_ids'* and *'class_names'* always need to have *'num_classes'* entries.

Models of *'type'='segmentation'*:

A model of *'type'='segmentation'* does predict background and therefore in this case the 'background' class is included in *'num_classes'*. For these models, *'num_classes'* is determined implicitly by the length of *'class_ids'*.

'num_trainable_params': **Any**

Number of trainable parameters (weights and biases) of the model. This value is an indicator for the size of the model when it is serialized.

'ood_available': **CL**

This parameter specifies whether a classification model has been extended for Out-of-Distribution Detection. If the operator fit_dl_out_of_distribution has been called for the model, the value of *'ood_available'* is *'true'*. Otherwise it is *'false'*.

Default: *'false'*

'ood_threshold': **CL**

A threshold that determines when a sample is classified as Out-of-Distribution. After calling fit_dl_out_of_distribution for a classification model, apply_dl_model additionally returns an *'ood_score'* and an *'ood_result'* in the output dictionaries. If the value of *'ood_score'* is greater than or equal to *'ood_threshold'*, the value of *'ood_result'* is *'true'*. Otherwise it is *'false'*. *'ood_threshold'* is only available after fit_dl_out_of_distribution has been called. That operator calculates a value for *'ood_threshold'* based on the provided dataset.

Value range: ≥ 0.

'orientation': **OCR-D**

This parameter allows to set a predefined orientation angle for the word detection. To revert to default behavior using the internal orientation estimation, *'orientation'* is set to *'auto'*.

Value range: $(-\pi, \pi]$.

Default: *'auto'*

'optimize_for_inference': **3D-GPD CL MLC OCR-D OCR-R GC-AD OD SE Gen**

Defines whether the model is optimized and only applicable for inference.

The model remains executable on HALCON standard devices even after optimization (unlike optimize_dl_model_for_inference).

Setting this parameter to *'true'* frees model memory not needed for inference (e.g., memory of gradients). This can significantly reduce the amount of memory needed by the model. As a consequence, models with this characteristic have no gradients accessible (needed e.g., for training or calculations of heatmaps). Operators using values from freed memory (e.g., train_dl_model_batch) will automatically reset this parameter value to *'false'*.

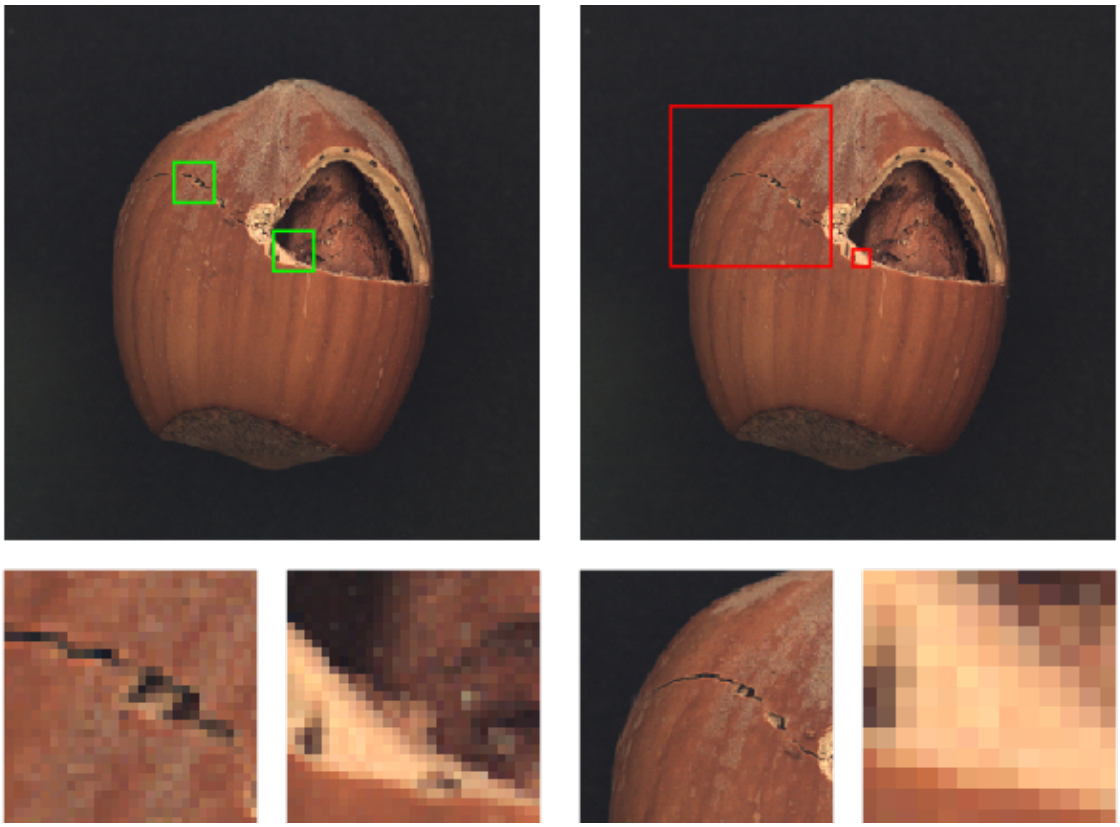
In case the value is reset to *'false'* (both, manually or automatically), memory needed by the model for training is reallocated. This implies, a following training behaves as if *'momentum'* is temporarily set to *0* (as possible updating vectors have to be accumulated again).

Default: *'false'*

Restriction: For models of *'type'='counting'* the parameter *'optimize_for_inference'* cannot be set to *'true'*.

'patch_size': **GC-AD**

This parameter determines the size of the patches analyzed by the *'local'* subnetwork. The patch size should be chosen such that patches containing defects will differ clearly from patches without defects. This should cover all kinds of anomalies that might occur during inference. The patch size does not need to cover the defects as a whole. Note that the image is not divided into separate patches, but *'patch_size'* solely determines the scale on which the image is gradually analyzed by the *'local'* subnetwork. If the image size is changed *'patch_size'* should be adjusted accordingly.



(1) (2)

(1) Examples of patches with a suitable size (green). (2) The example patches (red) are either too large or too small to work for all possible defect classes that could occur in the images.

Restriction: The parameter *'patch_size'* must be smaller than or equal to *'image_width'* and *'image_height'*.

Default: *'patch_size'* = *33*

'precision': **Any**

Defines the data type that is internally used for the calculation of a forward pass of a deep learning model.

Default: *'float32'*

'precision_is_converted': **Any**

Indicates whether the model was subjected to a conversion process after training done by `optimize_dl_model_for_inference`.

Default: *'false'*

'runtime': **Any**

Defines the device on which the deep learning operators will be executed.

Note that the parameter *'device'* should be preferred to set the devices on which the deep learning operators will be executed.

Default: *'runtime'* = *'gpu'*

'cpu':

The training and inference operator will be executed on CPU. Note, training is only supported for specific platform types, please see the HALCON "`Installation Guide`".
In case the GPU has been used before, CPU memory is initialized, and if necessary values stored on the GPU memory are moved to the CPU memory.
For parallelization: The runtime is highly dependent on the number of threads set. The use of all available threads does not necessarily create a faster performance. How many threads are currently set can be queried with the operator `get_system`.
The implemented CPU parallelization is dependent on the architecture:

- Intel or AMD architecture: OpenMP. By default all available threads of the OpenMP runtime environment are used. The number of threads used can be specified with the parameter *'tsp_thread_num'* of the operator `set_system`.
- Arm architectures: Global Thread Pool. The number of threads can be set with the global parameter *'thread_num'* of the operator `set_system`.

For both architectures mentioned above, it is not possible to specify a thread-specific number of threads (via the parameter *'tsp_thread_num'* of the operator `set_system`).

'gpu':
The GPU memory is initialized. The operators `apply_dl_model`, `train_dl_model_batch`, and `train_dl_model_anomaly_dataset` will be executed on the GPU. For the specific requirements please refer to the HALCON "`Installation Guide`".

'solver_type': **Any**
This value defines the optimization algorithm with the goal to minimize the value of the total loss function. For more information we refer to the documentation of `train_dl_model_batch`. The following values can be set:

- *'adam'*: Adaptive moment estimation
- *'sgd'*: Stochastic gradient descent

Note that changing *'solver_type'* sets *'learning_rate'* back to its default value.

Models of *'type'*=*'anomaly_detection'*: The parameter *'solver_type'* has no effect.

Models of *'type'*=*'counting'*: The parameter *'solver_type'* has no effect.

'sort_by_line': **OCR-D**
The words are sorted line-wise based on the orientation of the localized word instances. If the parameter *'sort_by_line'* is set to *'false'*, the results will not be sorted.

Default: *'true'*

'standard_deviation_factor': **AD**
The anomaly score is calculated internally as the mean of certain internal scores s plus lambda times their standard deviation.

$$S_{Image} = \mu(s) + \lambda\sigma(s)$$

Where s denotes a pixel value of the internal `anomaly_image`, $\mu(s)$ the mean value of s and $\sigma(s)$ the standard deviation of s. The parameter *'standard_deviation_factor'* sets the value λ and thus controls how important the standard deviation is in comparison to the mean.

Default: *'standard_deviation_factor' = 3.0*

'summary': **Any**
This parameter returns information on the layers of the model. More precisely, it returns a tuple with a string for every layer. The string is as follows: `ID; NAME; TYPE; OUTPUT_SHAPE; CONNECTED_NODES`

- `ID`: Index of the layer in the CNN graph.
- `NAME`: Human-readable identifier (optional).
- `TYPE`: Human-readable identifier representing the type of the layer (e.g., `input` or `convolution`).
- `OUTPUT_SHAPE`: Size of the output, given in the form (`Width`, `Height`, `Depth`, *'batch_size'*). This means, the layer has feature maps of size `Width` times `Height` and therefrom `Depth` many. Together they form an iconic object with a channel for every feature map. The parameter *'batch_size'* determines, how many objects are returned together.
- `CONNECTED_NODES`: Comma separated list with IDs of the layers using the output of the current layer as input

E.g., `'3; conv1; convolution; (112, 112, 64, 160); 4'`.

Note, for some networks distributed with HALCON, the network architecture is confidential. In this case `get_dl_model_param` returns an empty tuple with *'summary'*.

'tiling': **OCR-D**
The input image is automatically split into overlapping tile images of size *'image_size'*, which are processed separately by the model. This allows processing images that are much larger than the actual *'image_size'* without having to zoom the input image. **Default:** *'false'*

'tiling_overlap': **OCR-D**
This parameter defines how much neighboring tiles overlap when input images are split (see *'tiling'*). The overlap is given in pixels.

Value range: ≥ 0.

Default: *64*

'type': **Any**

This parameter returns the HALCON-specific model type. The following types are distinguished:

- *'3d_gripping_point_detection'*
- *'anomaly_detection'*
- *'classification'*
- *'counting'*
- *'detection'*
- *'gc_anomaly_detection'*
- *'segmentation'*
- *'ocr_recognition'*
- *'ocr_detection'*
- *'generic'* - for certain read in models or models created with the DL framework, see `set_dl_model_param`.

'weight_prior': **Any**

Regularization parameter $\alpha \geq 0.0$ used for the regularization of the loss function. For a detailed description of the regularization term we refer to `train_dl_model_batch`. Simply put: Regularization favors simpler models that are less likely to learn noise in the data and generalize better. In case the classifier overfits the data, it is strongly recommended to try different values for the parameter *'weight_prior'* to improve the generalization properties of the neural network. Choosing its value is a trade-off between the models ability to generalize, overfitting, and underfitting. If α is too small, the model might overfit, if its too large the model might loose its ability to fit the data, because all weights are effectively zero. For finding an ideal value for α, we recommend a cross-validation, i.e. to perform the training for a range of values and choose the value that results in the best validation error. For typical applications, we recommend testing the values for *'weight_prior'* on a logarithmic scale between $1.0, ..., 0.0001$. If the training takes a very long time, one might consider performing the hyperparameter optimization on a reduced amount of data.

Models of *'type'*=*'anomaly_detection'*: The parameter *'weight_prior'* has no effect.

Models of *'type'*=*'counting'*: The parameter *'weight_prior'* has no effect.

Default: *'weight_prior'* = *0.0*, with exception of the pretrained classifiers

- `pretrained_dl_classifier_resnet18`: *'weight_prior'* = *0.0001*
- `pretrained_dl_classifier_resnet50`: *'weight_prior'* = *0.0001*
- `pretrained_dl_classifier_alexnet`: *'weight_prior'* = *0.0005*

Parameters

▷ **`DLModelHandle`** (input_control) . . . . . dl_model ⇝ *handle*
Handle of the deep learning model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'batch_size'
List of values: `GenParamName` ∈ {'adam_beta1', 'adam_beta2', 'adam_epsilon', 'alphabet', 'alphabet_internal', 'alphabet_mapping', 'anchor_aspect_ratios', 'anchor_num_subscales', 'anchor_angles', 'anomaly_score_tolerance', 'backbone', 'backbone_docking_layers', 'batch_size', 'batch_size_multiplier', 'bbox_heads_weight', 'capacity', 'class_heads_weight', 'class_ids', 'class_ids_no_orientation', 'class_names', 'class_weights', 'complexity', 'device', 'extract_feature_maps', 'freeze_backbone_level', 'gc_anomaly_networks', 'gpu', 'ignore_class_ids', 'ignore_direction', 'image_dimensions', 'image_height', 'image_num_channels', 'image_range_max', 'image_range_min', 'image_size', 'image_width', 'input_dimensions', 'instance_segmentation', 'instance_type', 'layer_names', 'learning_rate', 'mask_head_weight', 'max_level', 'max_num_detections', 'max_overlap', 'max_overlap_class_agnostic', 'meta_data', 'min_character_score', 'min_confidence', 'min_level', 'min_link_score', 'min_word_area', 'min_word_score', 'momentum', 'num_classes', 'num_trainable_params', 'ood_available', 'ood_threshold', 'optimize_for_inference', 'orientation', 'sort_by_line', 'patch_size', 'precision', 'precision_is_converted', 'runtime', 'solver_type', 'standard_deviation_factor', 'summary', 'tiling', 'tiling_overlap', 'type', 'weight_prior', 'model_uid', 'release_version', 'min_version'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.name(-array) ⇝ *integer* / string / real
Value of the generic parameter.

Result

If the parameters are valid, the operator `get_dl_model_param` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_dl_model`, `set_dl_model_param`

Possible Successors

`set_dl_model_param`, `apply_dl_model`, `train_dl_model_batch`, `train_dl_model_anomaly_dataset`

See also

`set_dl_model_param`

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Matching, Deep Learning Enhanced, Deep Learning Professional

get_dl_pruning_param (: : DLPruningHandle, GenParamName : GenParamValue)

Get information from a pruning data handle.

`get_dl_pruning_param` returns the parameter and pruning values `GenParamValue` of `GenParamName` from the pruning data handle `DLPruningHandle`.

The values of following parameters `GenParamName` can be retrieved:

'mode': Specifies the pruning method See `create_dl_pruning` for further information.
Default: *'oracle'*

'percentage': Determines how much will be removed in percents. E.g., in case of *'mode'='oracle'* it specifies the relative amount of possibly removable kernels to be removed, whereby only removable kernels are considered.
Default: *10*

Additionally, for *'mode'='oracle'*, the following pruning values can be retrieved using `GenParamName`:

'prunable_conv_num': Number of prunable convolutional layers in the network. Whether a layer is prunable depends on the specifications of the layer itself as well as the network architecture.

'prunable_conv_layer_name': Names of the prunable convolutional layers in the network.

'prunable_kernel_num': Number of prunable convolution kernels in the network. It considers that a pruned convolution has to retain at least 1 convolution kernel.

'pruned_kernel_num': Number of pruned convolution kernels in the network after a call of `gen_dl_pruned_model`.

'pruned_percentage': Actual percentage of pruned convolution kernels in the network after a call of `gen_dl_pruned_model`: the ratio *'pruned_kernel_num'* versus *'prunable_kernel_num'*.

The following table gives an overview, which parameters and values can be set using `set_dl_pruning_param` and which ones can be retrieved using `get_dl_pruning_param`.

Parameters	set	get
'mode'	x	x
'percentage'	x	x
'prunable_conv_num'		x
'prunable_conv_layer_name'		x
'prunable_kernel_num'		x
'pruned_kernel_num'		x
'pruned_percentage'		x

Parameters

▷ **DLPruningHandle** (input_control) . . . . . dl_pruning ⇝ *handle*
Pruning data handle.

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'percentage'
List of values: GenParamName ∈ {'mode', 'percentage', 'prunable_conv_num', 'prunable_conv_layer_name', 'prunable_kernel_num', 'pruned_kernel_num', 'pruned_percentage'}

▷ **GenParamValue** (output_control) . . . . . attribute.name(-array) ⇝ *integer* / string / real
Value of the generic parameter.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_dl_pruning, set_dl_pruning_param

See also

set_dl_pruning_param

Module

Deep Learning Professional

read_dl_model (: : FileName : DLModelHandle)

Read a deep learning model from a file.

The operator read_dl_model reads a deep learning model. Such models have to be in the HALCON format or in the ONNX format (see the reference below). Restrictions apply to the latter. As a result, the handle DLModelHandle is returned.

The model is loaded from the file FileName. This file is thereby searched in the directory $HALCONROOT/dl/ as well as in the currently used directory. The default HALCON file extension for deep learning networks is '.hdl'.

Please note that the values of runtime specific parameters are not written to file, see write_dl_model. As a consequence, when reading a model, these parameters are initialized with their default value, see get_dl_model_param.

Models that require a higher HALCON version (*'min_version'*) than the one currently in use cannot be read. Before writing a model the *'min_version'* can be checked with the operator get_dl_model_param.

For further explanations on deep learning models in HALCON, see the chapter Deep Learning / Model.

Reading in a Model Provided by HALCON

HALCON provides pretrained neural networks for classification and semantic segmentation. These neural networks are good starting points when training a custom network. They have been pretrained on a large image dataset. For anomaly detection, HALCON provides initial models.

Models for 3D Gripping Point Detection

The following network is provided for 3D Gripping Point Detection:

'pretrained_dl_3d_gripping_point.hdl'

The network expects up to 5 images of type `real`:

'image': intensity (gray value) image

'x': X-image (values need to increase from left to right)

'y': Y-image (values need to increase from top to bottom)

'z': Z-image (values need to increase from points close to the sensor to far points; this is for example the case if the data is given in the camera coordinate system)

'normals': 2D mappings

Additionally, the network requires certain image properties (for all input images mentioned above). The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values:

'image_width': 640

'image_height': 480

The network architecture allows changes concerning the image dimensions.

Models for Anomaly Detection

The following networks are provided for anomaly detection:

'initial_dl_anomaly_medium.hdl'

This neural network is designed to be memory and runtime efficient.

The network expects the images to be of the type `real`. Additionally, the network requires certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values:

'image_width': 480

'image_height': 480

'image_num_channels': 3

'image_range_min': -2

'image_range_max': 2

The network architecture allows changes concerning the image dimensions, but the sizes *'image_width'* and *'image_height'* have to be multiples of 32 pixels, resulting in a minimum of 32 pixels.

'initial_dl_anomaly_large.hdl'

This neural network is assumed to be better suited for more complex anomaly detection tasks. This comes at the cost of being more time and memory demanding.

The network expects the images to be of the type `real`. Additionally, the network requires certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values:

'image_width': 480

'image_height': 480

'image_num_channels': 3

'image_range_min': -2

'image_range_max': 2

The network architecture allows changes concerning the image dimensions, but the sizes *'image_width'* and *'image_height'* have to be multiples of 32 pixels, resulting in a minimum of 32 pixels.

Models for Global Context Anomaly Detection

The following networks are provided for Global Context Anomaly Detection:

'pretrained_dl_anomaly_global_context.hdl'

The network expects the images to be of the type `real`. Additionally, the network requires certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values:

'image_width': 256

'image_height': 256

'image_num_channels': 3

'image_range_min': -127.0

'image_range_max': 128.0

Models for Classification

The following pretrained neural networks are provided for classification and usable as backbones for detection:

'pretrained_dl_classifier_alexnet.hdl':

This neural network is designed for simple classification tasks. It is characterized by its convolution kernels in the first convolution layers, which are larger than those in other networks with comparable classification performance (e.g., *'pretrained_dl_classifier_compact.hdl'*). This may be beneficial for feature extraction.

This classifier expects the images to be of the type `real`. Additionally, the network is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224

'image_height': 224

'image_num_channels': 3

'image_range_min': -127.0

'image_range_max': 128.0

The network architecture allows changes concerning the image dimensions. *'image_width'* and *'image_height'* should not be less than 29 pixels. There is no maximum image size, but large image sizes will increase the memory demand and the runtime significantly. Changing the image size will reinitialize the weights of the fully connected layers and therefore makes a retraining necessary.

Note that one can improve the runtime for this network by fusing the convolution and ReLU layers, see `set_dl_model_param` and the parameter *'fuse_conv_relu'*.

'pretrained_dl_classifier_compact.hdl':

This neural network is designed to be more memory and runtime efficient.

The classifier expects the images to be of the type `real`. Additionally, the network requires certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224

'image_height': 224

'image_num_channels': 3

'image_range_min': -127.0

'image_range_max': 128.0

This network does not contain any fully connected layer. The network architecture allows changes concerning the image dimensions. *'image_width'* and *'image_height'* should not be less than 15 pixels.

'pretrained_dl_classifier_enhanced.hdl':

This neural network has more hidden layers than *'pretrained_dl_classifier_compact.hdl'* and is therefore assumed to be better suited for more complex classification tasks. This comes at the cost of being more time and memory demanding.

The classifier expects the images to be of the type `real`. Additionally, the network requires certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224

'image_height': 224

'image_num_channels': 3

'image_range_min': -127.0

'image_range_max': 128.0

The network architecture allows changes concerning the image dimensions. *'image_width'* and *'image_height'* should not be less than 47 pixels. There is no maximum image size, but large image sizes will increase the memory demand and the runtime significantly. Changing the image size will reinitialize the weights of the fully connected layers and therefore makes a retraining necessary.

'pretrained_dl_classifier_mobilenet_v2.hdl':

This classifier is a small and low-power model, for what reason it is more suitable for mobile and embedded vision applications.

The classifier expects the images to be of the type `real`. Additionally, the network requires certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224

'image_height': 224

'image_num_channels': 3

'image_range_min': -127.0

'image_range_max': 128.0

The network architecture allows changes concerning the image dimensions. *'image_width'* and *'image_height'* should not be less than 32 pixels. There is no maximum image size, but large image sizes will increase the memory demand and the runtime significantly.
On the GPU, the network architecture can benefit greatly from special optimizations, without which the network can be significantly slower.

'pretrained_dl_classifier_resnet18.hdl':

As the neural network *'pretrained_dl_classifier_enhanced.hdl'*, this classifier is suited for more complex tasks. However, due to its special structure, it provides the advantage of making the training more stable and internally more robust. Compared to the neural network *'pretrained_dl_classifier_resnet50.hdl'* it is less complex and has faster inference times.
The classifier expects the images to be of the type `real`. Additionally, the network requires certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224

'image_height': 224

'image_num_channels': 3

'image_range_min': -127.0

'image_range_max': 128.0

The network architecture allows changes concerning the image dimensions. *'image_width'* and *'image_height'* should not be less than 32 pixels. There is no maximum image size, but large image sizes will increase the memory demand and the runtime significantly. Despite the fully connected layer a change of the image size does not lead to a reinitialization of the weights.

'pretrained_dl_classifier_resnet50.hdl':

As the neural network *'pretrained_dl_classifier_enhanced.hdl'*, this classifier is suited for more complex tasks. However, due to its special structure, it provides the advantage of making the training more stable and internally more robust.
The classifier expects the images to be of the type `real`. Additionally, the network requires certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224

'image_height': 224

'image_num_channels': 3

'image_range_min': -127.0

'image_range_max': 128.0

The network architecture allows changes concerning the image dimensions. *'image_width'* and *'image_height'* should not be less than 32 pixels. There is no maximum image size, but large image sizes will increase the memory demand and the runtime significantly. Despite the fully connected layer a change of the image size does not lead to a reinitialization of the weights.

Models for Semantic Segmentation

The following pretrained neural networks are provided for semantic segmentation:

'pretrained_dl_edge_extractor.hdl':

This neural network is designed and pretrained for edge extraction. As a consequence this model is meant for two class problems with one class for edges and one for background.
This network expects the images to be of the type `real`. Additionally, the network is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the model has been trained:

'image_width': 512

'image_height': 512
'image_num_channels': 1
'image_range_min': -127.0
'image_range_max': 128.0
'num_classes': 2

The network architecture allows changes concerning the image dimensions, but the sizes *'image_width'* and *'image_height'* have to be multiples of 16 pixels, resulting in a minimum of 16 pixels.

'pretrained_dl_segmentation_compact.hdl':
This neural network is designed to handle segmentation tasks with detailed structures and uses only few memory and is runtime efficient.
The network architecture allows changes concerning the image dimensions, but requires a minimum *'image_width'* and *'image_height'* of 21 pixels.

'pretrained_dl_segmentation_enhanced.hdl':
This neural network has more hidden layers than *'pretrained_dl_segmentation_compact.hdl'* and is therefore better suited for segmentation tasks including more complex scenes.
The network architecture allows changes concerning the image dimensions, but requires a minimum *'image_width'* and *'image_height'* of 47 pixels.

Models for Deep OCR

The following pretrained neural networks are provided for Deep OCR:

'pretrained_deep_ocr_detection.hdl':
This neural network is the default pretrained detection component of a Deep OCR model, but can be retrained, too. It is designed to detect words in images.
This network expects the images to be of the type `real`. Additionally, the network is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the model has been trained:

'image_width': 1024
'image_height': 1024
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The network architecture allows changes concerning the image dimensions *'image_width'* and *'image_height'*.

'pretrained_deep_ocr_detection_compact.hdl':
This neural network is a more efficient pretrained network that can be used as detection component of a Deep OCR model. It is designed to detect words in images, and it can be retrained as well. This neural network is designed to be more memory and runtime efficient.
Regarding the input images and image dimensions, this network has the same requirements as the default model *'pretrained_deep_ocr_detection_compact.hdl'*.

'pretrained_deep_ocr_recognition.hdl':
This neural network is the default pretrained recognition component of a Deep OCR model, but can be retrained, too. It is designed to recognize words in images that are cropped to a single word.
This network expects the images to be of the type `real`. Additionally, the network is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the model has been trained:

'image_width': 120
'image_height': 32
'image_num_channels': 1
'image_range_min': -1.0
'image_range_max': 1.0

The network architecture allows changes concerning the image width *'image_width'*. The image height *'image_height'* cannot be changed. The parameter *'image_width'* is very important: its value can be decreased or increased to adapt to the expected lengths of words, e.g., due to the average width per character. A bigger *'image_width'* will consume more time and memory resources. The image width *'image_width'* may be changed after training.

Reading in a Model in the ONNX Format

You can read in an ONNX model, but there are some points to consider.

Restrictions

Reading in ONNX models with `read_dl_model`, some restrictions apply:

- Version 1.8.1 of the ONNX specification is supported. This means only operators until ONNX operator set version (OpSetVersion) 13 are supported. For operators with a higher OpSetVersion there is no guarantee that it can be supported. Further limitations are listed above.
- Only 32 bit floating point tensors are supported.
- Only models ending with a SoftMax layer are automatically recognized as classifiers. All other models are considered as generic model, thus models of *'type'* = *'generic'*. `set_dl_model_param` can be used to change the model type.
- The input graph nodes (images) must be of shape dimension 4: Number of images (=*'batch_size'*), *'num_channels'*, *'image_height'*, and *'image_width'*.

Automatic transformations

After reading an ONNX model with `read_dl_model`, some network transformations are executed automatically:

- Every non-global pooling layer with a resulting feature map of size 1x1 is converted to a global pooling layer. Doing so enables resizable input images. For more information about pooling layer and possible modes of operation, see the "`Solution Guide on Classification`".
- Layer pairs consisting of a convolution layer without activation and a directly connected activation layer with ReLU activation are fused. In order to so do, the output of the convolution layer is only used as input for the activation layer. As a result a convolution layer with activation mode ReLU is obtained. For more information about layers and possible modes of operation, see the "`Solution Guide on Classification`".

Supported operations

ONNX models with the following operations can be read by `read_dl_model`:

`'Add'`: No restrictions.

`'ArgMax'`: The following restrictions apply:

- attribute *'axis'*: The value must be *1*.
- attribute *'keepdims'*: The value must be *1*.
- attribute *'select_last_index'*: The value must be *0*.

`'AveragePool'`: The following restrictions apply:

- attribute *'count_include_pad'*: The value must be *0*.

`'BatchNormalization'`: No restrictions.

`'Clip'`: The following restrictions apply:

- attribute *'min'*: The value must be *0*.
- attribute *'max'*: The value must be greater than *0* and less than maximum float number.

`'Concat'`: No restrictions.

`'Constant'`: The following restrictions apply:

- attribute *'sparse_value'*: The attribute is not supported.
- attribute *'value'*: All entries in the tensor have to be identical.
- attribute *'value_floats'*: The attribute is not supported.
- attribute *'value_ints'*: The attribute is not supported.
- attribute *'value_string'*: The attribute is not supported.
- attribute *'value_strings'*: The attribute is not supported.

`'Conv'`: The following restrictions apply:

- attribute *'pads'*: Padding values greater than or equal to kernel size are not supported.

`'ConvTranspose'`: The following restrictions apply:

- attribute *'dilations'*: Only the value *'(1, 1)'* (no dilations) is supported.
- attribute *'group'*: Only the value *1* is supported (no grouped transposed convolution).
- attribute *'kernel_shape'*: Only symmetric kernel shapes are supported.

- attribute *'output_padding'*: See restrictions mentioned in `create_dl_layer_transposed_convolution`.
- attribute *'output_shape'*: The attribute is not supported.
- attribute *'pads'*: Padding values greater than or equal to kernel size are not supported.
- attribute *'strides'*: Only symmetric strides are supported.

'DepthToSpace': The following restrictions apply:

- attribute *'mode'*: The value must be *'CRD'*.

'Dropout': No restrictions.

'Gemm': The following restrictions apply:

- attribute *'alpha'*: The value must be *1*.
- attribute *'beta'*: The value must be *1*.
- attribute *'transA'*: The value must be *0*.

'GlobalAveragePool': No restrictions.

'GlobalMaxPool': The following restrictions apply:

- attribute *'dilations'*: The value must be *1*.

'LeakyRelu': No restrictions.

'LogSoftmax': The following restrictions apply:

- attribute *'axis'*: The value must be *1*.

'LRN': No restrictions. Hint: Attribute *'size'* has no effect.

'MaxPool': No restrictions.

'Mean': No restrictions.

'Mul': No restrictions.

'ReduceL2':

- attribute *'noop_with_empty_axes'*: The attribute is optional. The value must be *0*.
- attribute *'keepdims'*: The attribute is optional. The value must be *1*.
- attribute *'axes'*: The attribute is optional. If empty reduce all dimensions. In the new opset versions the attribute *'axes'* was moved to the inputs.

'ReduceMax': The following restrictions apply:

- attribute *'axes'*: The value must be *1*.
- attribute *'keepdims'*: The value must be *1*.

'ReduceSum':

- attribute *'noop_with_empty_axes'*: The attribute is optional. The value must be *0*.
- attribute *'keepdims'*: The attribute is optional. The value must be *1*.
- attribute *'axes'*: The attribute is optional. If empty reduce all dimensions. In the new opset versions the attribute *'axes'* was moved to the inputs.

'Relu': No restrictions.

'Resize': The following restrictions apply:

- attribute *'mode'*: Only the values *'linear'* or *'bilinear'* are supported.
- attribute *'coordinate_transformation_mode'*: Only the values *'pytorch_half_pixel'* and *'align_corners'* are supported.
- input tensor *'roi'*: If values are set they have no effect on the inference.
- The attributes *'cubic_coeff_a'*, *'exclude_outside'*, *'extrapolation_value'*, or *'nearest_mode'* have no effect.

'Reshape': The following restrictions apply:

- attribute *'allowzero'*: If the attribute is used its value must be *0*.

'Sigmoid': No restrictions.

'Softmax': The following restrictions apply:

- attribute *'axis'*: If the attribute is used its value must be *1*.

'Sub': No restrictions.

'Sum': No restrictions.

'Transpose': No restrictions.

Moreover the ONNX `'metadata_props'` field is supported. It is written to the model parameter *'meta_data'*.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Filename
Default: 'pretrained_dl_classifier_compact.hdl'
List of values: FileName ∈ {'initial_dl_anomaly_large.hdl', 'initial_dl_anomaly_medium.hdl', 'pretrained_deep_ocr_detection.hdl', 'pretrained_deep_ocr_detection_compact.hdl', 'pretrained_deep_ocr_recognition.hdl', 'pretrained_dl_3d_gripping_point.hdl', 'pretrained_dl_classifier_alexnet.hdl', 'pretrained_dl_classifier_compact.hdl', 'pretrained_dl_classifier_enhanced.hdl', 'pretrained_dl_classifier_resnet18.hdl', 'pretrained_dl_classifier_resnet50.hdl', 'pretrained_dl_classifier_mobilenet_v2.hdl', 'pretrained_dl_anomaly_global_context.hdl', 'pretrained_dl_segmentation_compact.hdl', 'pretrained_dl_segmentation_enhanced.hdl', 'pretrained_dl_edge_extractor.hdl'}
File extension: .hdl, .onnx

▷ **DLModelHandle** (output_control) . . . . . dl_model ⇝ *handle*
Handle of the deep learning model.

Result

If the parameters are valid, the operator read_dl_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_dl_model_param, get_dl_model_param, apply_dl_model, train_dl_model_batch, train_dl_model_anomaly_dataset

Alternatives

create_dl_model_detection

References

Open Neural Network Exchange (ONNX), https://onnx.ai/

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Matching, Deep Learning Enhanced, Deep Learning Professional

serialize_dl_model (: : DLModelHandle : SerializedItemHandle)

Serialize a deep learning model.

serialize_dl_model serializes the deep learning model defined by the handle DLModelHandle. The serialized model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_dl_model. See fwrite_serialized_item for an introduction of the basic principle of serialization.

The operator acts the same as write_dl_model except that the output is a serialized item instead of a file. For a detailed description please refer to the documentation of write_dl_model.

For further explanations to deep learning models in HALCON, see the chapter Deep Learning / Model.

Parameters

▷ **DLModelHandle** (input_control) . . . . . dl_model ⇝ *handle*
Handle of the deep learning model.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_dl_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`train_dl_model_batch`, `train_dl_model_anomaly_dataset`

Possible Successors

`train_dl_model_batch`, `train_dl_model_anomaly_dataset`, `fwrite_serialized_item`, `send_serialized_item`

See also

`deserialize_dl_model`, `apply_dl_model`, `train_dl_model_batch`, `train_dl_model_anomaly_dataset`

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Matching, Deep Learning Enhanced, Deep Learning Professional

set_dl_model_param (: : DLModelHandle, GenParamName, GenParamValue :)

Set the parameters of a deep learning model.

`set_dl_model_param` sets the parameters and hyperparameters `GenParamName` of the deep learning model `DLModelHandle` to the values `GenParamValue`.

The values `GenParamName` can attain, depend on the model type: There are parameters which can be set for any deep learning model while others can only be set for specific model types. A description of the parameters whose value you can only set but not retrieve is given below. For all other parameters the specific description is given in `get_dl_model_param`. In `get_dl_model_param` we also give an overview, for which type of model and using which operator a parameter can be set.

In the following we list the parameters `GenParamName` you can set using this operator, `set_dl_model_param`. Thereby, the following symbols denote the model type for which the parameter can be set and has a possible influence:

- 'Any': any method
- '3D-GPD': *'type'='3d_gripping_point_detection'*
- 'AD': *'type'='anomaly_detection'*
- 'CL': *'type'='classification'*
- 'MLC': *'type'='multi_label_classification'*
- 'DC': *'type'='counting'*
- 'OCR-D': *'type'='ocr_detection'*
- 'OCR-R': *'type'='ocr_recognition'*
- 'GC-AD': *'type'='gc_anomaly_detection'*

- 'OD': *'type'='detection'*
- 'SE': *'type'='segmentation'*
- 'Gen': *'type'='generic'*

'adam_beta1': **3D-GPD CL OCR-D OCR-R GC-AD OD SE Gen**
Moment for linear term in Adam solver. Only applicable if *'solver_type'* = *'adam'*.

'adam_beta2': **3D-GPD CL OCR-D OCR-R GC-AD OD SE Gen**
Moment for quadratic term in Adam solver. Only applicable if *'solver_type'* = *'adam'*.

'adam_epsilon': **3D-GPD CL OCR-D OCR-R GC-AD OD SE Gen**
Epsilon for numeric stability in Adam solver. Only applicable if *'solver_type'* = *'adam'*.

'alphabet': **OCR-R**

'alphabet_internal': **OCR-R**

'alphabet_mapping': **OCR-R**

'anomaly_score_tolerance': **GC-AD**

'backbone_docking_layers': **CL**
The docking layers can be specified for every classifier, also without using them as backbone. The specification is only considered for object detection backbones.

'batch_size': **Any**

'batch_size_multiplier': **3D-GPD CL MLC OCR-D OCR-R GC-AD OD SE Gen**

'batchnorm_momentum': **Any except DC**
Calling this option sets the momentum of all batch normalization layers within the network. The settable values are the same as described in `create_dl_layer_batch_normalization` for the parameter `Momentum`.
Note, if the network has no such layer, nothing is done and therewith the operation is regarded as successful.

'bbox_heads_weight', *'class_heads_weight'*: **OD**

'class_ids': **3D-GPD MLC OD SE**

'class_names': **CL MLC OD SE**

'class_weights': **CL**

'complexity': **AD**

'device': **Any**

'enable_resizing': **Any except DC**
Calling this option converts certain pooling layers within the network. More precisely, every non-global pooling layer with a resulting feature map of size 1x1 is converted into a global pooling layer. This means, a pooling layer performing e.g., average pooling is converted into one performing global average pooling. For more information about pooling layer and possible modes of operation, see the "`Solution Guide on Classification`".
Note, if this operation is performed, it can not be undone. Accordingly, you can only call it with the value *'true'*. Also, if the network has no such layer, nothing is done and therewith the operation is regarded as successful.

'extract_feature_maps': **CL**

'freeze_backbone_level': **OD**

'fuse_bn_relu': **Any except DC**
Calling this option, fuses layer pairs consisting of a batch normalization layer without activation and a directly connected activation layer with ReLU activation. In order to so do, the output of the batch normalization layer is only used as input for the activation layer. As a result a batch normalization layer with activation mode ReLU is obtained. For more information about layers and possible modes of operation, see the "`Solution Guide on Classification`".
Note, if this operation is performed, it can not be undone. Accordingly, you can only call it with the value *'true'*. Also, if the network has no such fusible layers, nothing is done and therewith the operation is regarded as successful.
Restriction: Leaky ReLU layers cannot be fused with batch normalization layers.

'fuse_conv_relu': **Any except DC**

Calling this option, fuses layer pairs consisting of a convolution layer without activation and a directly connected activation layer with ReLU activation. In order to so do, the output of the convolution layer is only used as input for the activation layer. As a result a convolution layer with activation mode ReLU is obtained. For more information about layers and possible modes of operation, see the "`Solution Guide on Classification`".

Note, if this operation is performed, it can not be undone. Accordingly, you can only call it with the value *'true'*. Also, if the network has no such fusible layers, nothing is done and therewith the operation is regarded as successful.

Restriction: Leaky ReLU layers cannot be fused with convolution layers.

'gc_anomaly_networks': **GC-AD**

'gpu': **Any**

'ignore_class_ids': **SE**

'image_dimensions': **3D-GPD AD CL MLC OCR-D OCR-R GC-AD SE**

'image_height', *'image_width'*: **3D-GPD AD CL MLC OCR-D OCR-R GC-AD SE**

'image_num_channels': **AD CL MLC OCR-D OCR-R GC-AD SE**

'image_range_max', *'image_range_min'*: **3D-GPD OCR-D OCR-R SE**

'image_size': **3D-GPD AD CL MLC OCR-D OCR-R GC-AD SE**

'input_dimensions': **AD CL MLC OCR-D OCR-R SE Gen**

'learning_rate': **3D-GPD CL MLC OCR-D OCR-R GC-AD OD SE Gen**

'mask_head_weight': **OD**

This parameter is only available for models with *'instance_segmentation'='true'*.

'max_num_detections': **OD**

'max_overlap': **OD**

'max_overlap_class_agnostic': **OD**

'meta_data': **Any**

'min_character_score': **OCR-D**

'min_confidence': **MLC OD**

'min_link_score': **OCR-D**

'min_word_area': **OCR-D**

'min_word_score': **OCR-D**

'momentum': **3D-GPD CL MLC OCR-D OCR-R GC-AD OD SE Gen**

Momentum for SGD solver.

Restriction: Only applicable for *'solver_type'* = *'sgd'*.

'ood_threshold': **CL**

'optimize_for_inference': **3D-GPD CL MLC OCR-D OCR-R GC-AD OD SE Gen**

'orientation': **OCR-D**

'patch_size': **GC-AD**

'runtime': **Any**

'runtime_init': **Any except DC**

If called with *'immediately'*, the GPU memory is initialized and the corresponding handle created. Otherwise this is done on demand, which may result in significantly larger execution times for the first call of `apply_dl_model` or `train_dl_model_batch`.

If the network architecture is changed subsequently, the GPU memory is reinitialized. This can happen e.g., for changes of *'batch_size'*, *'image_dimensions'* or *'input_dimensions'* with subsequent calls of `set_dl_model_param`.

Note, this parameter has no effect if:

- Running on CPUs, thus if *'runtime'* is set to *'cpu'*.
- Running with an AI^2-interface.

- The device has been set before using *'device'*.

'solver_type': **3D-GPD CL MLC OCR-D OCR-R GC-AD OD SE Gen**

'sort_by_line': **OCR-D**

'standard_deviation_factor': **AD**

'tiling': **OCR-D**

'tiling_overlap': **OCR-D**

'type': **Gen**

This parameter returns the HALCON-specific model type.

Models of *'generic'* fulfill all model functions. But several deep learning procedures rely a specific *'type'*. The value of *'type'* can be set from *'generic'* to the following values:

- *'classification'*
- *'segmentation'*

Setting a value for *'type'* to model it is checked if it has all layers necessary for inference and training. In case such layers are missing, they are added.

'weight_prior': **3D-GPD CL OCR-D OCR-R GC-AD OD SE Gen**

Attention

System requirements: To successfully set *'gpu'* parameters, cuDNN and cuBLAS are required, i.e., to set the parameter `GenParamName` *'runtime'* to *'gpu'*. For further details, please refer to the "`Installation Guide`", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods".

Parameters

▷ **`DLModelHandle`** (input_control) . . . . . dl_model ⇝ *handle*
Handle of the deep learning model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'batch_size'
List of values: `GenParamName` ∈ {'adam_beta1', 'adam_beta2', 'adam_epsilon', 'alphabet', 'alphabet_internal', 'alphabet_mapping', 'anchor_aspect_ratios', 'anchor_num_subscales', 'anchor_angles', 'anomaly_score_tolerance', 'backbone', 'backbone_docking_layers', 'batch_size', 'batch_size_multiplier', 'batchnorm_momentum', 'bbox_heads_weight', 'capacity', 'class_heads_weight', 'class_ids', 'class_names', 'class_weights', 'complexity', 'device', 'enable_resizing', 'extract_feature_maps', 'freeze_backbone_level', 'fuse_bn_relu', 'fuse_conv_relu', 'gc_anomaly_networks', 'gpu', 'ignore_class_ids', 'ignore_direction', 'image_dimensions', 'image_height', 'image_num_channels', 'image_range_max', 'image_range_min', 'image_size', 'image_width', 'input_dimensions', 'instance_type', 'learning_rate', 'mask_head_weight', 'max_level', 'max_num_detections', 'max_overlap', 'max_overlap_class_agnostic', 'meta_data', 'min_character_score', 'min_confidence', 'min_level', 'min_link_score', 'min_word_area', 'min_word_score', 'momentum', 'num_classes', 'orientation', 'ood_threshold', 'optimize_for_inference', 'patch_size', 'runtime', 'runtime_init', 'solver_type', 'standard_deviation_factor', 'sort_by_line', 'summary', 'tiling', 'tiling_overlap', 'type', 'weight_prior'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value(-array) ⇝ *integer* / string / real
Value of the generic parameter.
Default: 1
Suggested values: `GenParamValue` ∈ {1, 2, 3, 50, [80,60], 80, 60, 0.001, -127, 128, 'adam', 'cpu', 'gpu', 'immediately', 'sgd'}

Result

If the parameters are valid, the operator `set_dl_model_param` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_dl_model, get_dl_model_param

Possible Successors

get_dl_model_param, apply_dl_model, train_dl_model_batch, train_dl_model_anomaly_dataset

See also

get_dl_model_param

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Matching, Deep Learning Enhanced, Deep Learning Professional

set_dl_pruning_param (: : DLPruningHandle, GenParamName, GenParamValue :)

Set parameter in a pruning data handle.

set_dl_pruning_param sets the parameter GenParamName in the pruning handle DLPruningHandle to the value GenParamValue.

An overview as well as a description of the settable parameters is given in get_dl_model_param.

Parameters

▷ **DLPruningHandle** (input_control) . . . . . dl_pruning ⇝ *handle*
Pruning data handle.

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'percentage'
List of values: GenParamName ∈ {'mode', 'percentage'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / string / integer
Value of the generic parameter.
Default: 10
Suggested values: GenParamValue ∈ {'oracle', 10, 20, 30, 40, 50}

Result

If the parameters are valid, the operator set_dl_pruning_param returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_dl_pruning, get_dl_pruning_param

See also

get_dl_pruning_param

Module

Deep Learning Professional

train_dl_model_batch (: : DLModelHandle, DLSampleBatch : DLTrainResult)

Train a deep learning model.

The operator train_dl_model_batch performs a training step of the deep learning model contained in DLModelHandle. The current loss values are returned in the dictionary DLTrainResult.

For DLModelHandle all model types but *'anomaly_detection'* and *'counting'* are valid. See train_dl_model_anomaly_dataset for the training of anomaly detection models.

A training step means here to perform a single update of the weights, based on the batch images given in DLSampleBatch. The optimization algorithms which can be used are explained further in the subsection "Further Information on the Algorithms" below. For more information on how to train a network, please see the subchapter "The Network and the Training Process" in Deep Learning.

To successfully train the model, its applicable hyperparameters need to be set and the training data handed over according to the model requirements. For information to the hyperparameters, see the chapter of the corresponding model and the general chapter Deep Learning.

The training data consists of images and corresponding information. This operator expects one batch of training data, handed over in the tuple of dictionaries DLSampleBatch. Such a DLSample dictionary is created out of DLDataset for every image sample, e.g., by the procedure gen_dl_samples. See the chapter Deep Learning / Model for further information to the used dictionaries and their keys.

The number of images in a DLSampleBatch tuple needs to be a multiple of the *'batch_size'*. In particular on GPU the parameter *'batch_size'* is limited by the amount of available memory. In order to process more images in one training step, the model parameter *'batch_size_multiplier'* can be set to a value greater than 1. The number of DLSample dictionaries being passed to the training operator needs to be equal to *'batch_size'* times *'batch_size_multiplier'*. Note that a training step calculated for a batch and a *'batch_size_multiplier'* greater 1 is an approximation of a training step calculated for the same batch but with a *'batch_size_multiplier'* equal to 1 and an accordingly greater *'batch_size'*. As an example, the loss calculated with a *'batch_size'* of 4 and a *'batch_size_multiplier'* of 2 is usually not equal to the loss calculated with a *'batch_size'* of 8 and a *'batch_size_multiplier'* of 1, although the same number of DLSample dictionaries is used for training in both cases. However, the approximation generally delivers comparably good results, so it can be utilized if you wish to train with a larger number of images than your GPU allows. In some rare cases the approximation with a *'batch_size'* of 1 and an accordingly large *'batch_size_multiplier'* does not show the expected performance. Set the *'batch_size'* to a value greater than 1 can help to solve this issue.

In the output dictionary DLTrainResult you get the current value of the total loss as the value for the key total_loss as well as the values for all other losses included in your model.

For models of *'type'* = *'detection'* such losses are e.g., the losses for the heads of every selected level, namely the 'Huber Loss' for the bounding box regression heads and the 'Focal Loss' for the classification heads (see also Deep Learning / Object Detection and Instance Segmentation as well as *'max_level'* and *'min_level'* in get_dl_model_param).

Further Information on the Algorithms

During training, an optimization algorithm is applied with the goal to minimize the value of the total loss function. The latter one is determined based on the prediction of the neural network for the current batch of images.

In HALCON we have two optimization algorithms available so far, the SGD (stochastic gradient descent) and Adam (adaptive moment estimation).

SGD: The SGD updates the layers' weights of the previous iteration $t-1$, w_{t-1}, to the new values w_t at iteration t as follows:

$$\begin{aligned} v_t &= \mu v_{t-1} - \lambda \nabla_w L \\ w_t &= w_{t-1} + v_t. \end{aligned}$$

Here, λ is the learning rate, μ the momentum, L the total loss, and $\nabla_w L$ the gradient of the total loss with respect to the weights. The variable v_t is used to include the influence of the momentum μ.

Adam: Like the SGD, Adam updates the layer's weights of the previous iteration but comes with an adaptive moment estimation to automatically estimate a scaling for the learning rate. In this way it is determined how fast the solver moves towards a minimum. The estimated moments are the first two moments of the weights' gradients which are the mean and the uncentered variance. To estimate the moments Adam uses exponentially moving averages m_t and v_t, computed on the gradient evaluated on a mini-batch. This results in the following formula:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2. \end{aligned}$$

g is the weight's gradient on the current mini-batch, β_1 is the moment for the linear term, and β_2 is the moment for the quadratic term of the Adam solver. Furthermore, Adam has so-called bias correctors $\hat{m}_t$ and $\hat{v}_t$. These values are computed as follows:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}. \end{aligned}$$

As a last step the moving averages are used to scale the learning rates individually for each parameter. To perform the weights update this results with w_t as the model weights, and λ as the learning rate in the following formula.

$$w_t = w_{t-1} - \lambda \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

Here ϵ is the parameter to ensure numeric stability. For a more detailed description we refer to the referenced paper.

The different models may have several losses implemented, which are summed up. To this sum the regularization term $E_\alpha(w)$ is added, which generally penalizes large weights, and together they form the total loss. The different types of losses are:

`Huber Loss` (model of *'type'='detection'*): The 'Huber Loss' is also known as 'Smooth L1 Loss'. The total 'Huber Loss' is the sum of the contributions from all bounding box variables of all found instances in the batch. For a single bounding box variable this contribution defined as follows:

$$L_{Huber}(x) = \begin{cases} 0.5x^2/\beta & \text{if } |x| < \beta \\ |x| - 0.5\beta & \text{else} \end{cases}.$$

Thereby, x denotes a bounding box variable and β a parameter fixed to a value of 0.11.

We refer to `create_dl_layer_loss_huber` for more information.

`Focal Loss` (model of *'type'='detection'*): The total 'Focal Loss' is the sum of the contributions from all found instance in the batch. For a single sample, this contribution is defined as follows:

$$L_{focal}(p) = -\sum_{c=0}^{C-1} \alpha_t^c \left(1 - p_t^c\right)^\gamma \log\left(p_t^c\right),$$

where γ is a parameter fixed to a value of 2. α^c stands for the class specific weight ('class_weights') of the c-th class and p_t, α_t are defined as

$$p_t := \begin{cases} p & \text{if } y = 1 \\ 1-p & \text{else} \end{cases}, \quad \text{and} \quad \alpha_t := \begin{cases} \alpha & \text{if } y = 1 \\ 1-\alpha & \text{else} \end{cases}.$$

Here, $p = (p^0, \ldots, p^{C-1})$ is a tuple of the model's estimated probabilities for each of the C-many classes, and y_c is a one-hot encoded target vector that encodes the class of the annotation.

We refer to `create_dl_layer_loss_focal` for more information.

`Multinomial Logistic Loss` (model of *'type'= 'classification', 'segmentation'*): The 'Multinomial Logistic Loss' is also known as 'Cross Entropy Loss'. It is defined as follows:

$$L_{mn}\big(f(x,w)\big) \quad = \quad -\frac{1}{N}\sum_{n=0}^{N-1}\alpha(y_n)\langle y_n, \log\big(f(x_n,w)\big)\rangle$$

Here, $f(x,w)$ is the predicted result which depends on the network weights w and the input batch x. y_n is a one-hot encoded target vector that encodes the label of the n-th image x_n of the batch x containing N-many images, and $\log(f(x_n,w))$ shall be understood to be a vector such that $\log$ is applied on each component of $f(x_n,w)$. The value $\alpha(y_n)$ is a class specific weight for the class given by y_n. This weight corresponds to the value set by 'class_weights' and is normalized by the sum over the weights for all classes in addition.

We refer to `create_dl_layer_loss_cross_entropy` for more information.

The regularization term $E_\alpha(w)$ is a weighted l^2-norm involving all K weights except for biases. Its influence can be controlled through α. Latter one is the hyperparameter *'weight_prior'*, which can be set with `set_dl_model_param`.

$$E_\alpha(w) = \frac{\alpha}{2}\sum_{k=0}^{K-1}|w_k|^2$$

Here the index k runs over all weights of the network, except for the biases which are not regularized. The regularization term $E_\alpha(w)$ generally penalizes large weights, thus pushing the weights towards zero, which effectively reduces the complexity of the model.

Attention

The operator `train_dl_model_batch` internally calls functions that might not be deterministic. Therefore, results from multiple calls of `train_dl_model_batch` can slightly differ, although the same input values have been used. Setting *'cudnn_deterministic'* of `set_system` may influence this behavior.

System requirements: Implementation on CPU is limited to specific platform types. To run this operator on GPU by setting *'runtime'* to *'gpu'* (see `get_dl_model_param`), cuDNN and cuBLAS are required. Please refer to the "`Installation Guide`", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods", for the specific system requirements.

Parameters

- ▷ **`DLModelHandle`** (input_control) . . . . . dl_model ⇝ *handle*
  Deep learning model handle.
- ▷ **`DLSampleBatch`** (input_control) . . . . . dict-array ⇝ *handle*
  Tuple of Dictionaries with input images and corresponding information.
- ▷ **`DLTrainResult`** (output_control) . . . . . dict ⇝ *handle*
  Dictionary with the train result data.

Result

If the parameters are valid, the operator `train_dl_model_batch` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`read_dl_model`, `set_dl_model_param`, `get_dl_model_param`

Possible Successors

`apply_dl_model`

See also

`apply_dl_model`

References

D. P. Kingma, J. Ba: "Adam: A method for Stochastic Optimization", 2014, https://arxiv.org/pdf/1412.6980.pdf

Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Deep Learning Professional

write_dl_model (: : DLModelHandle, FileName :)

Write a deep learning model in a file.

write_dl_model writes the deep learning model DLModelHandle to the file given by FileName. Please note that the values of the runtime specific parameters *'gpu'*, *'runtime'*, and *'runtime_init'* are not written.

The default HALCON file extension for deep learning models is '.hdl'.

The model can be read with read_dl_model.

For further explanations to deep learning models in HALCON, see the chapter Deep Learning / Model.

Parameters

▷ **DLModelHandle** (input_control) ..... dl_model ⇝ *handle*
Handle of the deep learning model.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Filename
File extension: .hdl

Result

If the parameters are valid, the operator write_dl_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_dl_model, train_dl_model_batch, train_dl_model_anomaly_dataset, set_dl_model_param

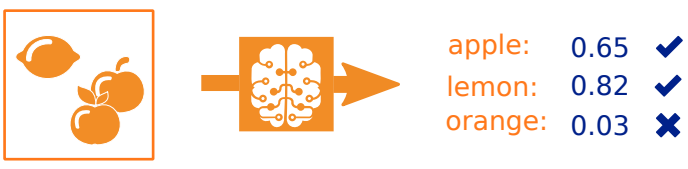
Module

Foundation. This operator uses dynamic licensing (see the 'Installation Guide'). Which of the following modules is required depends on the specific usage of the operator:
3D Metrology, OCR/OCV, Matching, Deep Learning Enhanced, Deep Learning Professional

9.5 Multi-Label Classification

This chapter explains how to use multi-label classification based on deep learning, both for the training and inference phases.

Multi-label classification based on deep learning is a method, in which an image gets assigned multiple confidence values. These confidence values indicate how likely the image belongs to each of the different classes. Thus, multi-label classification means to assign multiple specific classes to an image. This is illustrated with the following schema.



A possible multi-label classification example, in which the network has three classes. The input image gets confidence values assigned for each of the three classes: 'apple' 0.65, 'lemon' 0.82, and 'orange' 0.03. The top predictions tell us, the image has the classes 'apple' and 'lemon'.

In order to do your specific task, thus to classify your data into the different classes, the classifier has to be trained accordingly. In HALCON, we use a technique called transfer learning (see also the chapter Deep Learning). Hence, we provide pretrained networks, representing classifiers which have been trained on huge amounts of labeled image data. These classifiers have been trained and tested to perform well on industrial image classification tasks. One of these classifiers, already trained for general multi-label classifications, is now retrained for your specific task. To do this, the classifier must know, which classes have to be distinguished and what examples of these classes look like. This is represented by your dataset, i.e., your images with the corresponding ground truth labels. More information on the data requirements can be found in the section "Data".

In HALCON, multi-label classification with deep learning is implemented within the more general deep learning model. For more information to the latter one, see the chapter Deep Learning / Model. For the specific system requirements in order to apply deep learning, please refer to the HALCON "Installation Guide".

The following sections are introductions to the general workflow needed for multi-label classification, information related to the involved data and parameters, and explanations to the evaluation measures.

General Workflow

In this paragraph, we describe the general workflow for a multi-label classification task based on deep learning. It is subdivided into the five parts creation of the model, preprocessing of the data, training of the model, evaluation of the trained model, and inference on new images. Thereby we assume, your dataset is already labeled, see also the section "Data" below. Have a look at the HDevelop example `dl_multi_label_classification_workflow.hdev` for an application.

Create a Multi-Label Classification Model This part is about how to create a multi-label classification model. The model is created based on a backbone model. The creation can be done with the procedure

- `create_dl_model_multi_label_classification`.

Thereby, a model `DLModelHandle` is created, where the backbone serves as feature extractor, to which the necessary layers for the training are added. Generally it is possible to use any deep learning model as a backbone, but, depending on the complexity, we recommend one of the classification models. For further information about the backbone and which one to use, see the procedure documentation of `create_dl_model_multi_label_classification`.

`create_dl_model_multi_label_classification` uses a docking layer, to attach the necessary layers for multi-label classification to the backbone classifier. The pretrained classifiers provided by HALCON have already specified docking layers. But when you use a self-provided classifier as backbone, you have to specify it yourself.

After the model has been created it can be saved using `write_dl_model` and subsequently read using `read_dl_model`.

Preprocess the data This part is about how to preprocess your data. The single steps are also shown in the HDevelop example `dl_multi_label_classification_workflow.hdev`.

1. The information what is to be found on an image of your training dataset needs to be transferred. This is done by the procedure
   - `read_dl_dataset_multi_label_classification`.

   Thereby a dictionary `DLDataset` is created, which serves as a database and stores all necessary information about your data. For more information about the data and the way it is transferred, see the section "Data" below and the chapter Deep Learning / Model.
2. Split the dataset represented by the dictionary `DLDataset`. This can be done using the procedure
   - `split_dl_dataset`.

   The resulting split will be saved under the key `split` in each sample entry of `DLDataset`.
3. Now you can preprocess your dataset. For this, you can use the procedure

- preprocess_dl_dataset.

In case of custom preprocessing, this procedure offers guidance on the implementation.

To use this procedure, specify the preprocessing parameters as e.g., the image size. Store all the parameters with their values in a dictionary DLPreprocessParam, wherefore you can use the procedure

- create_dl_preprocess_param.

We recommend to save this dictionary DLPreprocessParam in order to have access to the preprocessing parameter values later during the inference phase.

Training of the model This part is about how to train a multi-label classifier. The single steps are also shown in the HDevelop example dl_multi_label_classification_workflow.hdev.

1. Set the training parameters and store them in the dictionary TrainParam. These parameters include:
   - the hyperparameters, for an overview see the chapter Deep Learning.
   - parameters for possible data augmentation (optional).
   - parameters for the evaluation during training.
   - parameters for the visualization of training results.
   - parameters for serialization.

   This can be done using the procedure
   - create_dl_train_param.
2. Train the model. This can be done using the procedure
   - train_dl_model.

   The procedure expects:
   - the model handle DLModelHandle
   - the dictionary with the data information DLDataset
   - the dictionary with the training parameter TrainParam
   - the information, over how many epochs the training shall run.

   In case the procedure train_dl_model is used, the total loss as well as optional evaluation measures are visualized.

Evaluation of the trained model In this part we evaluate the trained classifier. The single steps are also shown in the HDevelop example dl_multi_label_classification_workflow.hdev.

1. The evaluation can conveniently be done using the procedure
   - evaluate_dl_model.
2. The dictionary EvaluationResult holds the asked evaluation measures. You can visualize your evaluation results using the procedure
   - dev_display_multi_label_classification_evaluation.

Inference on new images This part covers the application of a deep-learning-based multi-label classification model. The single steps are also shown in the HDevelop example dl_multi_label_classification_workflow.hdev.

1. Set the parameters as e.g., *'batch_size'* using the operator
   - set_dl_model_param.
2. Generate a data dictionary DLSample for each image. This can be done using the procedure
   - gen_dl_samples_from_images.
3. Preprocess the images as done for the training. We recommend to do this using the procedure
   - preprocess_dl_samples.

   When you saved the dictionary DLPreprocessParam during the preprocessing step, you can directly use it as input to specify all parameter values.
4. Apply the model using the operator

- apply_dl_model.

5. Retrieve the results from the dictionary *'DLResultBatch'*.

Data

We distinguish between data used for training and data for inference. Latter one consists of bare images. But for the former one you already know to which classes the images belong and provide this information over the corresponding labels.

As a basic concept, the model handles data over dictionaries, meaning it receives the input data over a dictionary DLSample and returns a dictionary DLResult and DLTrainResult, respectively. More information on the data handling can be found in the chapter Deep Learning / Model.

Data for training and evaluation The dataset consists of images and corresponding information. They have to be provided in a way the model can process them. Concerning the image requirements, find more information in the section "Images" below.

The training data is used to train and evaluate a network for your specific task. With the aid of this data the network can learn which classes are to be distinguished, how such examples look like, and how to find them. For each image the class of every object in the image is provided as label. There are various ways to store and retrieve this information. How the data has to be formatted in HALCON for a DL model is explained in the chapter Deep Learning / Model. In short, a dictionary DLDataset serves as a database for the information needed by the training and evaluation procedures.

If you have already a labeled multi-label classification dataset, you can use the procedure read_dl_dataset_multi_label_classification. It formats the data and creates a dictionary DLDataset. This procedure can also be used to create a DLDataset for multi-label classification from object detection, segmentation or classification data.

For training a multi-label classifier, we use a technique called transfer learning (see the chapter Deep Learning). For this, you need less resources, but still a suitable set of data. While in general the network should be more reliable when trained on a larger dataset, the amount of data needed for training also depends on the complexity of the task. You also want enough training data to split it into three subsets, used for training, validation, and testing the network. These subsets are preferably independent and identically distributed, see the section "Data" in the chapter Deep Learning.

Images Regardless of the application, the network poses requirements on the images regarding e.g., the image dimensions. The specific values depend on the backbone-network itself and can be queried with get_dl_model_param. In order to fulfill these requirements, you may have to preprocess your images. Standard preprocessing is implemented in preprocess_dl_dataset and in preprocess_dl_samples for a single sample, respectively. In case of custom preprocessing these procedures offer guidance on the implementation.

Network output The network output depends on the task:

training As output, the operator train_dl_model_batch will return a dictionary DLTrainResult with the current value of the total loss as well as values for all other losses included in your model.

inference and evaluation As output, the operator apply_dl_model will return a dictionary DLResult for every image. For multi-label classification, this dictionary will include a tuple for every image, with the confidence values for every class to be distinguished in descending order and a second tuple with the corresponding class IDs. Further information on the output dictionary can be found in the chapter Deep Learning / Model.

Interpreting the Multi-Label Classification Results

When we classify an image, we obtain a set of confidence values, telling us the affinity of the image to every class. It is also possible to compute the following values.

Precision, Recall, and F-Score In multi-label classification whole images are classified. To check how well the trained network is performing, precision and recall are computed.

The precision is the proportion of all correct predicted positives to all predicted positives (true and false ones). Thus, it is a measure of how many positive predictions really belong to the selected class.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

The recall, also called the "true positive rate", is the proportion of all correct predicted positives to all real positives. Thus, it is a measure of how many samples belonging to the selected class were predicted correctly as positives.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

A classifier with high recall but low precision finds most members of positives (thus members of the class), but at the cost of also classifying many negatives as member of the class. A classifier with high precision but low recall is just the opposite, classifying only few samples as positives, but most of these predictions are correct. An ideal classifier with high precision and high recall will classify many samples as positive with a high accuracy.

To represent this with a single number, we compute the F1-score, the harmonic mean of precision and recall. Thus, it is a measure of the classifier's accuracy.

$$\text{F1-score} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

Mean Average Precision Mean average precision (mAP), and average precision (AP) of a class for a threshold.

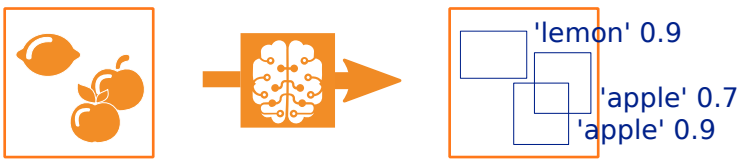
The AP value is an average of maximum precision at different recall values. In simple words it tells us, if the classes predicted for the images are generally correct predictions or not. Thereby we pay more attention to the predictions with high confidence values. The higher the value, the better.

You can obtain the specific AP values, the averages over the classes, the averages over the thresholds, and the average over both, the classes and the thresholds. The latter one is the mAP, a measure to tell us how well images are classified.

9.6 Object Detection and Instance Segmentation

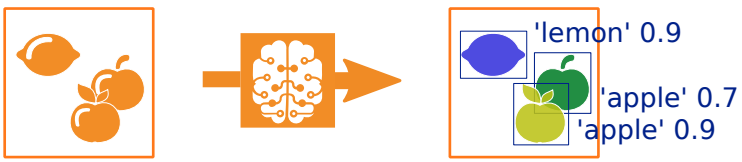
This chapter explains how to use object detection based on deep learning.

With **object detection** we want to find the different instances in an image and assign them to a class. The instances can partially overlap and still be distinguished as distinct. This is illustrated in the following schema.



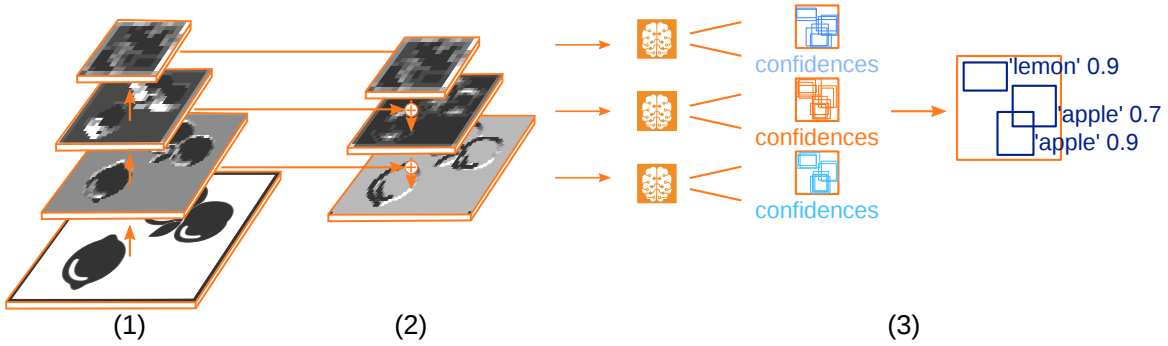
A possible example for object detection: Within the input image three instances are found and assigned to a class.

Instance segmentation is a special case of object detection, where the model also predicts an instance mask marking the specific region of the instance within the image. This is illustrated in the following schema. In general the explanations to object detection also apply to instance segmentation. Possible differences are brought up in the specific sections.



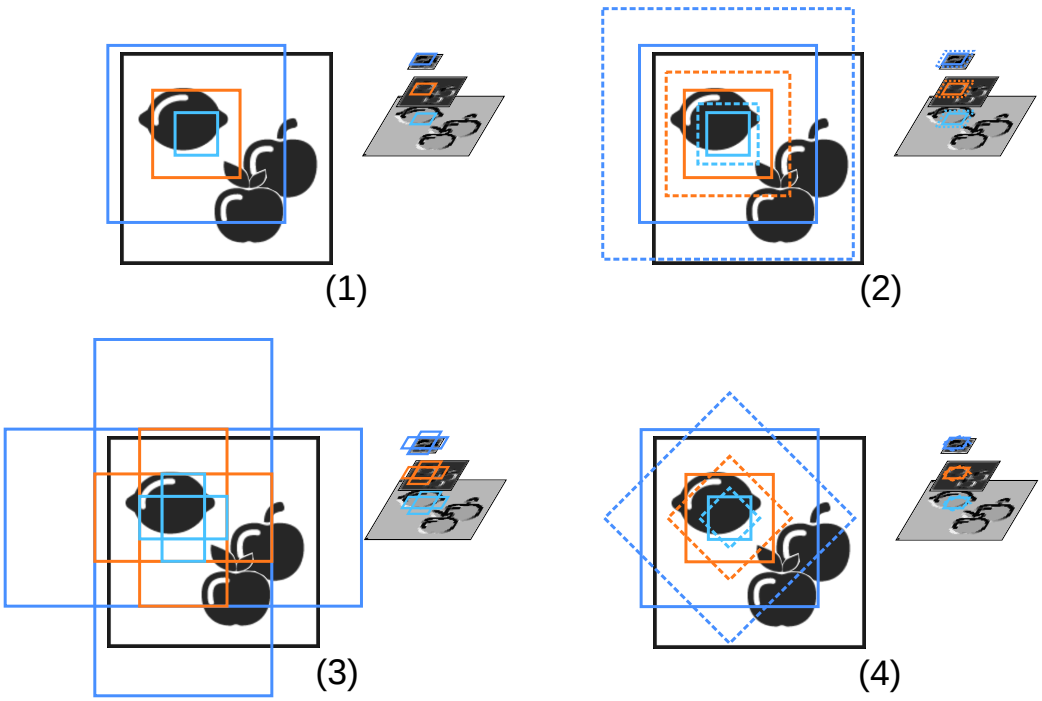
A possible example for instance segmentation: Within the input image three instances are found. Each instances is assigned to a class and obtains a mask marking its particular region.

Object detection leads to two different tasks: Finding the instances and classifying them. In order to do so, we use a combined network consisting of three main parts. The first part, called backbone, consists of a pretrained classification network. Its task is to generate various feature maps, so the classifying layer is removed. These feature maps encode different kinds of information at different scales, depending how deep they are in the network. See also the chapter Deep Learning. Thereby, feature maps with the same width and height are said to belong to the same level. In the second part, backbone layers of different levels are combined. More precisely, backbone levels of different levels are specified as docking layers. Their feature maps are combined. As a result we obtain feature maps containing information of lower and higher levels. These are the feature maps we will use in the third part. This second part is also called feature pyramid and together with the first part it constitutes the feature pyramid network. The third part consists of additional networks, called heads, for every selected level. They get the corresponding feature maps as input and learn how to localize and classify, respectively, potential objects. Additionally this third part includes the reduction of overlapping predicted bounding boxes. An overview of the three parts is shown in the following figure.



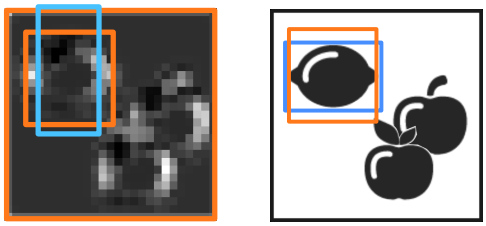
A schematic overview of the mentioned three parts: (1) The backbone. (2) Backbone feature maps are combined and new feature maps generated. (3) Additional networks, called heads, which learn how to localize and classify, respectively, potential objects. Overlapping bounding boxes are suppressed.

Let us have a look what happens in this third part. In object detection, the location in the image of an instance is given by a rectangular bounding box. Hence, the first task is to find a suiting bounding box for every single instance. To do so, the network generates reference bounding boxes and learns, how to modify them to fit the instances best possible. These reference bounding boxes are called anchors. The better these anchors represent the shape of the different ground truth bounding boxes, the easier the network can learn them. For this purpose the network generates a set of anchors on every anchor point, thus on every pixel of the used feature maps of the feature pyramid. Such a set consists of anchors of all combinations of shapes, sizes, and for instance type *'rectangle2'* (see below) also orientations. The shape of those boxes is affected by the parameter *'anchor_aspect_ratios'* the size by the parameter *'anchor_num_subscales'*, and the orientation by the parameter *'anchor_angles'*, see the illustration below and `get_dl_model_param`. If the parameters generate multiple identical anchors, the network internally ignores those duplicates.



Schema for anchors in the feature map (right) and in the input image (left). (1) Anchors are created on the feature maps of different levels, e.g., the ones drawn (in light blue, orange, and dark blue). (2) Anchors of different sizes are created setting *'anchor_num_subscales'*. (3) Anchors of different shapes are created setting *'anchor_aspect_ratios'*. (4) Anchors of different orientations are created setting *'anchor_angles'* (only for instance type *'rectangle2'*).

The network predicts offsets how to modify the anchors in order to obtain bounding boxes fitting the potential instances better. The network learns this with its bounding box heads, which compare the anchors generated for their level with the corresponding ground truth bounding boxes, thus the information where in the image the single instances are. An illustration is shown in the figure below.



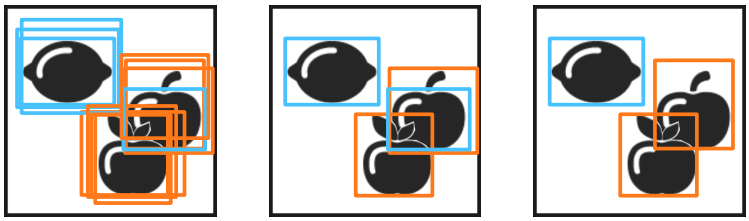
Bounding box comparisons, illustrated for instance type *'rectangle1'*. (1) The network modifies the anchor (light blue) in order to predict a better fitting bounding box (orange). (2) During training, the predicted bounding box (orange) gets compared with the most overlapping ground truth bounding box (blue) so the network can learn the necessary modifications.

As mentioned before, feature maps of different levels are used. Depending on the size your instances have in comparison to the total image it is beneficial to include early feature maps (where the feature map is not very compressed and therefore small features are still visible) and deeper feature maps (where the feature map is very compressed and only large features are visible) or not. This can be controlled by the parameters *'min_level'* and *'max_level'*, which determine the levels of the feature pyramid.

With these bounding boxes we have the localization of a potential instance, but the instance is not classified yet. Hence, the second task consists of classifying the content of the image part within the bounding boxes. This is done by the class heads. For more information about classification in general, see the chapter Deep Learning / Classification and the "`Solution Guide on Classification`".

Most probably the network will find several promising bounding boxes for a single object. The reduction of overlapping predicted bounding boxes is done by non-maximum suppression, set over the parameters *'max_overlap'*

and *'max_overlap_class_agnostic'* when creating the model or using `set_dl_model_param` afterwards. An illustration is given in the figure below.

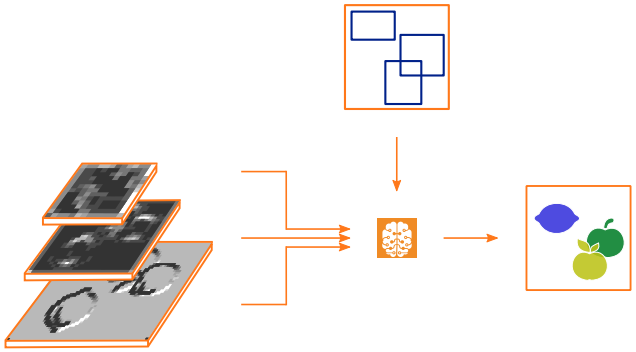


(1) (2) (3)

Suppression of significant overlapping bounding boxes, illustrated for instance type *'rectangle1'*. (1) The network finds several promising instances for the class apple (orange) and lemon (blue). (2) The suppression of overlapping instances assigned to the same class is set by *'max_overlap'*. Overlapping instances of different classes are not suppressed. (3) Using the parameter *'max_overlap_class_agnostic'*, also strongly overlapping instances of different classes get suppressed.

As output you get bounding boxes proposing possible localizations of objects and confidence values, expressing the affinity of this image part to one of the classes.

In case of instance segmentation a further part follows. An additional head obtains as input those parts of the feature maps that correspond to the predicted bounding boxes. Thereout it predicts a (class-agnostic) binary mask image. This mask marks the region within the image belonging to the predicted instance. A schematic overview is given below.



A schematic overview of the mask prediction. The illustration uses the parts shown in the overview of the three parts of an object detection network shown above.

In HALCON, object detection with deep learning is implemented within the more general deep learning model. For more information to the latter one, see the chapter Deep Learning / Model. Two different instance types of object detection models are implemented, differing in the orientation of the bounding boxes:

instance type *'rectangle1'* The rectangular bounding boxes are axis-aligned.

instance type *'rectangle2'* The rectangular bounding boxes have an arbitrary orientation.

For the specific system requirements in order to apply deep learning, please refer to the HALCON "`Installation Guide`".

The following sections are introductions to the general workflow needed for object detection, information related to the involved data and parameters, and explanations to the evaluation measures.

General Workflow

In this paragraph, we describe the general workflow for an object detection task based on deep learning. It is subdivided into the four parts creating the model and preprocessing of the data, training of the model, evaluation of the trained model, and inference on new images. Thereby we assume, your dataset is already labeled, see also the section "Data" below. Have a look at the HDevelop example series `detect_pills_deep_learning` for an application. The example `dl_instance_segmentation_workflow.hdev` shows the complete workflow for an instance segmentation application.

Creation of the model and dataset preprocessing This part covers the creation of a DL object detection model and the adaptation the data for this model. The single steps are also shown in the HDevelop example `detect_pills_deep_learning_1_prepare.hdev`.

1. Create a model using the operator
   - `create_dl_model_detection`.

   Thereby you will have to specify at least the backbone and the number of classes to be distinguished. For instance segmentation the parameter *'instance_segmentation'* has to be set to create an according model.

   Further parameters can be set over the dictionary `DLModelDetectionParam`. Their values should be well chosen for the specific task, not least to possibly reduce memory consumption and runtime. See the operator documentation for more information.

   Anchor parameters suiting your dataset can be estimated using the procedure
   - `determine_dl_model_detection_param`.

   In case the dataset is not representing all orientations the network will face during training (e.g., due to augmentation), the suggested values need to be adapted accordingly.

   Note, after creation of the model, its underlying network architecture is fixed to the specified input values. As a result the operator returns a handle *'DLModelHandle'*.

   Alternatively you can also use
   - `read_dl_model`.

   to read in a model you have already saved with `write_dl_model`.
2. The information what is to be found on which image of your training dataset needs to be read in. This can be done reading the data out of
   - a `DLDataset` dictionary using `read_dict`, or
   - a file in the COCO data format using the procedure `read_dl_dataset_from_coco`, whereby a dictionary `DLDataset` is created.

   The dictionary `DLDataset` serves as a database storing all necessary information about your data. For more information about the data and the way it is transferred, see the section "Data" below and the chapter Deep Learning / Model.
3. Split the dataset represented by the dictionary `DLDataset`. This can be done using the procedure
   - `split_dl_dataset`.

   The resulting split will be saved over the key `split` in each sample entry of `DLDataset`.
4. The network imposes requirements on the images, as e.g., the image width and height. You can retrieve every single value using the operator
   - `get_dl_model_param`.

   or you can retrieve all necessary parameter using the procedure
   - `create_dl_preprocess_param_from_model`.

   Note, for classes declared by *'class_ids_no_orientation'* the bounding boxes need to be treated specially during preprocessing. As a consequence, these classes should be set at this point latest.

   Now you can preprocess your dataset. For this, you can use the procedure
   - `preprocess_dl_dataset`.

   This procedure also offers guidance on how to implement a customized preprocessing procedure. We recommend to preprocess and store all images used for the training before starting the training, since this speeds up the training significantly.

   To visualize the preprocessed data, the procedure
   - `dev_display_dl_data`

   is available.

Training of the model This part covers the training of a DL object detection model. The single steps are also shown in the HDevelop example `detect_pills_deep_learning_2_train.hdev`.

1. Set the training parameters and store them in the dictionary `TrainParam`. These parameters include:
   - the hyperparameters, for an overview see the section "Model Parameters and Hyperparameters" below and the chapter Deep Learning.
   - parameters for possible data augmentation
2. Train the model. This can be done using the procedure
   - `train_dl_model`.

   The procedure expects:
   - the model handle `DLModelHandle`
   - the dictionary with the data information `DLDataset`
   - the dictionary with the training parameters `TrainParam`
   - the information, over how many epochs the training shall run.

   During the training you should see how the total loss minimizes.

Evaluation of the trained model In this part we evaluate the object detection model. The single steps are also shown in the HDevelop example `detect_pills_deep_learning_3_evaluate.hdev`.

1. Set the model parameters which may influence the evaluation.
2. The evaluation can conveniently be done using the procedure
   - `evaluate_dl_model`.

   This procedure expects a dictionary `GenParamEval` with the evaluation parameters. Set the parameter `detailed_evaluation` to *'true'* to get the data necessary for the visualization.
3. You can visualize your evaluation results using the procedure
   - `dev_display_detection_detailed_evaluation`.

Inference on new images This part covers the application of a DL object detection model. The single steps are also shown in the HDevelop example `detect_pills_deep_learning_4_infer.hdev`.

1. Request the requirements the network imposes on the images using the operator
   - `get_dl_model_param`

   or the procedure
   - `create_dl_preprocess_param_from_model`.
2. Set the model parameter described in the section "Model Parameters and Hyperparameters" below, using the operator
   - `set_dl_model_param`.

   The *'batch_size'* can be generally set independently from the number of images to be inferred. See `apply_dl_model` for details on how to set this parameter for greater efficiency.
3. Generate a data dictionary `DLSample` for each image. This can be done using the procedure
   - `gen_dl_samples_from_images`.
4. Every image has to be preprocessed as done for the training. For this, you can use the procedure
   - `preprocess_dl_samples`.
5. Apply the model using the operator
   - `apply_dl_model`.
6. Retrieve the results from the dictionary *'DLResultBatch'*.

Data

We distinguish between data used for training and evaluation, consisting of images with their information about the instances, and data for inference, which are bare images. For the first ones, you provide the information defining for each instance to which class it belongs and where it is in the image (via its bounding box). For instance segmentation the pixel-precise region of the objects is needed (provided via the masks).

As a basic concept, the model handles data over dictionaries, meaning it receives the input data over a dictionary `DLSample` and returns a dictionary `DLResult` and `DLTrainResult`, respectively. More information on the data handling can be found in the chapter Deep Learning / Model.

Data for training and evaluation The dataset consists of images and corresponding information. They have to be provided in a way the model can process them. Concerning the image requirements, find more information in the section "Images" below.

The training data is used to train and evaluate a network for your specific task. With the aid of this data the network can learn which classes are to be distinguished, how such examples look like, and how to find them. The necessary information is provided by telling for each object in every image to which class this object belongs to and where it is located. This is done by providing a class label and an enclosing bounding box for every object. In case of instance segmentation additionally a mask is needed for every instance. There are different ways possible, how to store and retrieve this information. How the data has to be formatted in HALCON for a DL model is explained in the chapter Deep Learning / Model. In short, a dictionary `DLDataset` serves as a database for the information needed by the training and evaluation procedures. You can label your data and directly create the dictionary `DLDataset` in the respective format using the MVTec Deep Learning Tool, available from the MVTec website. If you have your data already labeled in the standard COCO format, you can use the procedure `read_dl_dataset_from_coco` (for *'instance_type'* = *'rectangle1'* only). It formats the data and creates a dictionary `DLDataset`. For further information on the needed part of the COCO data format, please refer to the documentation of the procedure.

You also want enough training data to split it into three subsets, used for training, validation and testing the network. These subsets are preferably independent and identically distributed, see the section "Data" in the chapter Deep Learning.

Note, that in object detection the network has to learn how to find possible locations and sizes of the instances. That is why also the later important instance locations and sizes need to appear representatively in your training dataset.

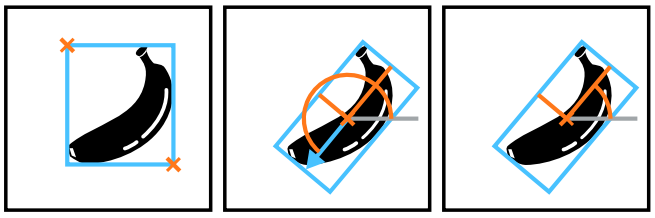
Images Regardless of the application, the network poses requirements on the images regarding e.g., the image dimensions. The specific values depend on the network itself and can be queried with `get_dl_model_param`. In order to fulfill these requirements, you may have to preprocess your images. Standard preprocessing of the entire dataset and therewith also the images is implemented in `preprocess_dl_dataset` and in `preprocess_dl_samples` for a single sample, respectively. This procedure also offers guidance on how to implement a customized preprocessing procedure.

Bounding boxes Depending on the instance type of object detection model, the bounding boxes are parametrized differently:

instance type *'rectangle1'* The bounding boxes are defined over the coordinates of upper left corner (*'bbox_row1'*, *'bbox_col1'*) and the lower right corner (*'bbox_row2'*, *'bbox_col2'*). This is consistent with `gen_rectangle1`.

instance type *'rectangle2'* The bounding boxes are defined over the coordinates of their center (*'bbox_row'*, *'bbox_col'*), the orientation *'bbox_phi'* and the half edge lengths *'bbox_length1'* and *'bbox_length2'*. The orientation is given in arc measure and indicates the angle between the horizontal axis and *'bbox_length1'* (mathematically positive). This is consistent with `gen_rectangle2`.

If in case of *'rectangle2'* you are interested in the oriented bounding box, but without considering the direction of the object within the bounding box, the parameter *'ignore_direction'* can be set to *'true'*. This is illustrated in the figure below.



(1) (2) (3)

Bounding box formats of the different object detection instance types: (1) Instance type *'rectangle1'*. (2) Instance type *'rectangle2'*, where the bounding box is oriented towards the banana end. (3) Instance type *'rectangle2'*, where the oriented bounding box is of interest without considering the direction of the banana within the bounding box.

Mask Instance segmentation requires, in addition to a tight bounding box, a mask for every object to be learned. Such a mask is given as a region. Note, these regions are given with respect to the image.

The masks for a single image are given as an object tuple of regions, see the illustration below. The order of the masks corresponds to the order of the bounding box annotations.

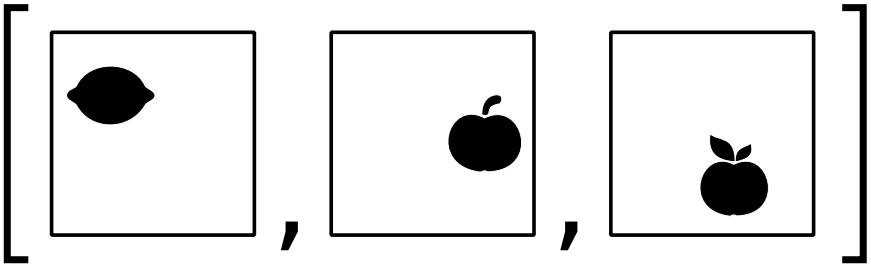


Illustration of the masks of an image: The tupel contains for every object to be learned an independent region.

Network output The network output depends on the task:

training As output, the operator `train_dl_model_batch` will return a dictionary `DLTrainResult` with the current value of the total loss as well as values for all other losses included in your model.

inference and evaluation As output, the operator `apply_dl_model` will return a dictionary `DLResult` for every image. For object detection, this dictionary will include for every detected instance its bounding box and confidence value of the assigned class as well as its mask in case of instance segmentation. Thereby several instances may be detected for the same object in the image, see the explanation to the non-maximum suppression above. The resulting bounding boxes are parametrized according to the instance type (specified over *'instance_type'*) and given in pixel centered sub-pixel accurate coordinates. For more information to the coordinate system, see the chapter Transformations / 2D Transformations. Further information on the output dictionary can be found in the chapter Deep Learning / Model.

Model Parameters and Hyperparameters

Next to the general DL hyperparameters explained in Deep Learning, there are further hyperparameters relevant for object detection:

- *'bbox_heads_weight'*
- *'class_heads_weight'*
- *'mask_head_weight'* (in case of instance segmentation)

These hyperparameters are explained in more detail in `get_dl_model_param` and set using `create_dl_model_detection`.

For an object detection model, there are two different types of model parameters:

- Parameter defining your architecture. They can not be changed anymore once your model is created. These parameters are all set using the operator `create_dl_model_detection` when creating your model.
- Parameter influencing your predictions and as a consequence the evaluation results. Those only relevant for object detection are
  - *'max_num_detections'*
  - *'max_overlap'*
  - *'max_overlap_class_agnostic'*
  - *'min_confidence'*

  They are explained in more detail in `get_dl_model_param`. To set them you can use `create_dl_model_detection` when creating your model or `set_dl_model_param` afterwards.

Evaluation measures for the Results from Object Detection

For object detection, the following evaluation measures are supported in HALCON. Note that for computing such a measure for an image, the related ground truth information is needed.

- Mean average precision, `mAP` and average precision (AP) of a class for an IoU threshold, `ap_iou_classname`

  The AP value is an average of maximum precision at different recall values. In simple words it tells us, if the objects predicted for this class are generally correct detections or not. Thereby we pay more attention to the predictions with high confidence values. The higher the value, the better.

  To count a prediction as a hit, we want both correct, its top-1 classification and its localization. The measure, telling us the correctness of the localization is the intersection over union, IoU: an instance is localized correctly if the IoU is higher than the demanded threshold. The IoU is explained in more detail below. For this reason, the AP value depends on the class and on the IoU threshold.

  You can obtain the specific AP values, the averages over the classes, the averages over the IoU thresholds, and the average over both, the classes and the IoU thresholds. The latter one is the mean average precision, mAP, a measure to tell us how well instances are found and classified.

- True Positives, False Positives, False Negatives

  The concept of true positive, false positives, and false negatives is explained in Deep Learning. It applies for object detection with the exception that there are different kinds of false positives, as e.g.:

  - An instance got classified wrongly.
  - An instance was found where there is only background.
  - An instance was localized badly, meaning the IoU between the instance and its ground truth is lower than the evaluation IoU threshold.
  - There is a duplicate, thus at least two instances overlap mainly with the same ground truth bounding box, but they overlap not more than *'max_overlap'* with each other, so none of them got suppressed.

  Note, these values are only available from the detailed evaluation. This means, in `evaluate_dl_model` the parameter `detailed_evaluation` has to be set to *'true'*.

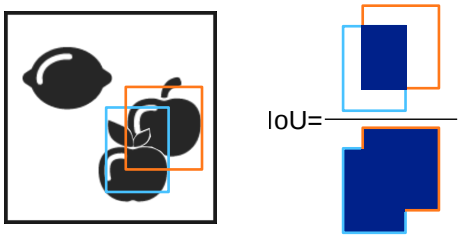
- Score of Angle Precision (SoAP)

  The SoAP value is a score for the precision of the inferred orientation angles. This score is determined by the angle differences between the inferred instances (I) and the corresponding ground truth annotations (GT):

  $\text{SoAP} = 1.0 - \frac{1}{\pi}\frac{1}{k} \cdot \sum_k \left|\varphi_k^{\text{I}} - \varphi_k^{\text{GT}}\right|,$

  where the index k runs over all inferred instances. This score only applies for detection models of *'instance_type' 'rectangle2'*.

Before mentioned measures use the intersection over union (IoU). The IoU is a measure for the accuracy of an object detection. For a proposed bounding box it compares the ratio between area of intersection and the area of overlap with the ground truth bounding box. A visual example is shown in the following schema.



(1) (2)

Visual example of the IoU, illustrated for instance type *'rectangle1'*. (1) The input image with the ground truth bounding box (orange) and the predicted bounding box (light blue). (2) The IoU is the ratio between the area intersection and the area overlap.

In case of instance segmentation the IoU is calculated (by default) based on the masks. It is possible to change the default and use the bounding boxes instead.

```
create_dl_model_detection ( : : Backbone, NumClasses,
    DLModelDetectionParam : DLModelHandle )
```

Create a deep learning network for object detection or instance segmentation.

With the operator `create_dl_model_detection` a deep learning network for object detection or instance segmentation is created. See the chapter Deep Learning / Object Detection and Instance Segmentation for further information on object detection and instance segmentation based on deep learning. The handle of this network is returned in `DLModelHandle`.

You can specify your model and its architecture over the parameters listed below. To successfully create a detection model, you need to specify its backbone and the number of classes the model shall be able to distinguish. The first information is handed over through the parameter `Backbone` which is explained below in the section "Possible Backbones". The second information is given through the parameter `NumClasses`. Note, this parameter fixes the number of classes the network will distinguish and therewith also the number of entries in *'class_ids'* and *'class_names'*.

The values of all other applicable parameters can be specified using the dictionary `DLModelDetectionParam`. Such a parameter is e.g., the *'instance_type'*, determining which kind of bounding boxes the model handles. To create a deep learning network for instance segmentation the parameter *'instance_segmentation'* has to be set to *'true'*. The full list of parameters that can be set is given below in the section "Settable Parameters". Some parameters are only available for instance segmentation. In case a parameter is not specified, the default value is taken to create the model. Note, parameters influencing the network architecture will not be changeable anymore once the network has been created. All the other parameters can still be set or changed using the operator `set_dl_model_param`. An overview, how parameters can be set is given in `get_dl_model_param`, where also a description of the specific parameters is provided. After creating the object detection model, the *'type'* will automatically be set to *'detection'*.

Possible Backbones

The parameter `Backbone` determines the backbone your network will use. See the chapter Deep Learning / Object Detection and Instance Segmentation for more information to the backbone. In short, the backbone consists of a pretrained classifier, from which only the layers necessary to generate the feature maps are kept. Hence, there are no fully connected layers anymore in the network. This implies that you read in a classifier as feature extractor for the subsequent detection network. For this you can read in a classifier in the HALCON format or a model in the ONNX format, see `read_dl_model` for more information.

`create_dl_model_detection` attaches the feature pyramid on different levels of the backbone. More precisely, the backbone has for different levels a layer specified as docking layer. When creating a detection model, the feature pyramid is attached on the corresponding docking layer. The pretrained classifiers provided by HALCON have already specified docking layers. But when you use a self-provided classifier as backbone, you have to specify them yourself. You can set `backbone_docking_layers` as part of the classifier using the operator `set_dl_model_param` or the backbone as such using this operator.

The docking layers are from different levels and therefore the feature maps used in the feature pyramid are of different size. More precisely, in the feature pyramid the feature map lengths are halved with every level. By implication, the input image lengths need to be halved for every level. This means, the network architectures allow changes concerning the image dimensions, but the dimensions *'image_width'* and *'image_height'* need to be an integer multiple of 2^{level}. Here, $level$ is the highest level up to which the feature pyramid is built. This value depends on the attached networks as well as on the docking layers. For the provided classifiers the list below mentions, up to which levels the feature pyramid is built using default settings.

HALCON provides the following pretrained classifiers you can read in as backbone:

'pretrained_dl_classifier_alexnet.hdl':

This neural network is designed for simple classification tasks. It is characterized by its convolution kernels in the first convolution layers, which are larger than in other networks with comparable classification performance (e.g., *'pretrained_dl_classifier_compact.hdl'*). This may be beneficial for feature extraction.

This backbone expects the images to be of the type `real`. Additionally, the backbone is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The default feature pyramid built on this backbone goes up to level 4.

'pretrained_dl_classifier_compact.hdl':
This neural network is designed to be memory and runtime efficient.

This backbone expects the images to be of the type `real`. Additionally, the backbone is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The default feature pyramid built on this backbone goes up to level 4.

'pretrained_dl_classifier_enhanced.hdl':
This neural network has more hidden layers than *'pretrained_dl_classifier_compact.hdl'* and is therefore assumed to be better suited for more complex tasks. But this comes at the cost of being more time and memory demanding.

This backbone expects the images to be of the type `real`. Additionally, the backbone is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The default feature pyramid built on this backbone goes up to level 5.

'pretrained_dl_classifier_mobilenet_v2.hdl':
This classifier is a small and low-power model, and hence it is more suitable for mobile and embedded vision applications.

This backbone expects the images to be of the type `real`. Additionally, the backbone is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The default feature pyramid built on this backbone goes up to level 4.

'pretrained_dl_classifier_resnet18.hdl':
As the network *'pretrained_dl_classifier_enhanced.hdl'*, this network is suited for more complex tasks. But its structure differs, bringing the advantage of making the training more stable and being internally more robust. Compared to the neural network *'pretrained_dl_classifier_resnet50.hdl'* it is less complex and has faster inference times.

This backbone expects the images to be of the type `real`. Additionally, the backbone is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224

'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The default feature pyramid built on this backbone goes up to level 5.

***'pretrained_dl_classifier_resnet50.hdl'*:**

As the network *'pretrained_dl_classifier_enhanced.hdl'*, this network is suited for more complex tasks. But its structure differs, bringing the advantage of making the training more stable and being internally more robust.

This backbone expects the images to be of the type `real`. Additionally, the backbone is designed for certain image properties. The corresponding values can be retrieved with `get_dl_model_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The default feature pyramid built on this backbone goes up to level 5.

Settable Parameters

Parameters you can set for your model when creating it using `create_dl_model_detection` (see `get_dl_model_param` for explanations):

- *'anchor_angles'*
- *'anchor_aspect_ratios'* (legacy: *'aspect_ratios'*)
- *'anchor_num_subscales'* (legacy: *'num_subscales'*)
- *'backbone_docking_layers'*
- *'bbox_heads_weight'*, *'class_heads_weight'*
- *'capacity'*
- *'class_ids'*
- *'class_ids_no_orientation'*
- *'class_names'*
- *'class_weights'*
- *'freeze_backbone_level'*
- *'ignore_direction'*
- *'image_dimensions'*
- *'image_height'*, *'image_width'*
- *'image_num_channels'*
- *'instance_segmentation'*
- *'instance_type'*
- *'mask_head_weight'*
  Restriction: only instance segmentation
- *'max_level'*, *'min_level'*
- *'max_num_detections'*
- *'max_overlap'*
- *'max_overlap_class_agnostic'*
- *'min_confidence'*
- *'optimize_for_inference'*

Attention

To successfully set *'gpu'* parameters, cuDNN and cuBLAS are required, i.e., to set the parameter `GenParamName` *'runtime'* to *'gpu'*. For further details, please refer to the "`Installation Guide`", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods".

Parameters

▷ **Backbone** (input_control) . . . . . filename.read ⇝ *string*
Deep learning classifier, used as backbone network.
Default: 'pretrained_dl_classifier_compact.hdl'
List of values: Backbone ∈ {'pretrained_dl_classifier_alexnet.hdl', 'pretrained_dl_classifier_compact.hdl', 'pretrained_dl_classifier_enhanced.hdl', 'pretrained_dl_classifier_mobilenet_v2.hdl', 'pretrained_dl_classifier_resnet18.hdl', 'pretrained_dl_classifier_resnet50.hdl'}
File extension: .hdl

▷ **NumClasses** (input_control) . . . . . integer ⇝ *integer*
Number of classes.
Default: 3

▷ **DLModelDetectionParam** (input_control) . . . . . dict ⇝ *handle*
Parameters for the object detection model.
Default: []

▷ **DLModelHandle** (output_control) . . . . . dl_model ⇝ *handle*
Deep learning model for object detection.

Result

If the parameters are valid, the operator create_dl_model_detection returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_dl_model_param, get_dl_model_param, apply_dl_model, train_dl_model_batch

Alternatives

read_dl_model

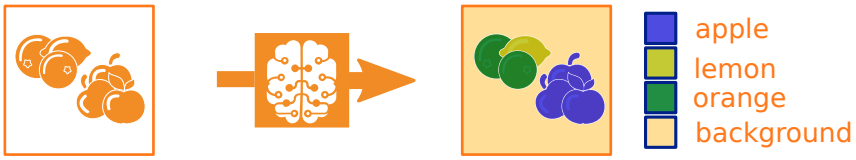
Module

Deep Learning Professional

9.7 Semantic Segmentation and Edge Extraction

This chapter explains how to use semantic segmentation based on deep learning, both for the training and inference phases.

With **semantic segmentation** we assign each pixel of the input image to a class using a deep learning (DL) network.

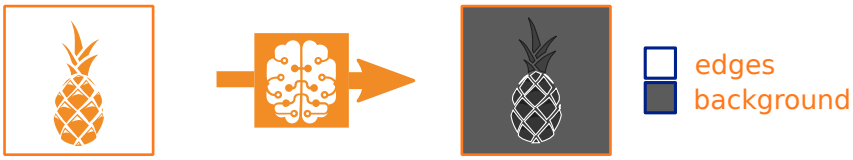


A possible example for semantic segmentation: Every pixel of the input image is assigned to a class, but neither the three different instances of the class 'apple' nor the two different instances of the class 'orange' are distinguished objects.

The result of semantic segmentation is an output image, in which the pixel value signifies the assigned class of the corresponding pixel in the input image. Thus, in HALCON the output image is of the same size as the input

image. For general DL networks the deeper feature maps, representing more complex features, are usually smaller than the input image (see the section “The Network and the Training Process” in Deep Learning). To obtain an output of the same size as the input, HALCON uses segmentation networks with two components: an encoder and a decoder. The encoder determines features of the input image as done, e.g., for deep-learning-based classification. As this information is ’encoded’ in a compressed format, the decoder is needed to reconstruct the information to the desired outcome, which, in this case, is the assignment of each pixel to a class. Note that, as pixels are classified, overlapping instances of the same class are not distinguished as distinct.

Edge extraction is a special case of semantic segmentation, where the model is trained to distinguish two classes: ’edge’ and ’background’. For more information, see “Solution Guide I - Basics”.



A possible example for edge extraction: A special case of semantic segmentation, where every pixel of the input image is assigned to one of the two classes: ’edge’ and ’background’.

Semantic segmentation with deep learning is implemented within the more general deep learning model of HALCON. For more information to the latter one, see the chapter Deep Learning / Model. For the specific system requirements in order to apply deep learning, please refer to the HALCON “Installation Guide”.

The following sections are introductions to the general workflow needed for semantic segmentation, information related to the involved data and parameters, and explanations to the evaluation measures.

General Workflow

In this paragraph, we describe the general workflow for a semantic segmentation task based on deep learning. It is subdivided into the four parts preprocessing of the data, training of the model, evaluation of the trained model, and inference on new images. Thereby we assume, your dataset is already labeled, see also the section “Data” below. Have a look at the HDevelop example series `segment_pill_defects_deep_learning` for an application. The example `segment_edges_deep_learning_with_retraining` shows the complete workflow for an edge extraction application.

Preprocess the data This part is about how to preprocess your data. The single steps are also shown in the HDevelop example `segment_pill_defects_deep_learning_1_preprocess.hdev`.

1. The information what is to be found in which image of your training dataset needs to be transferred. This is done by the procedure
   - `read_dl_dataset_segmentation`.

   Thereby a dictionary `DLDataset` is created, which serves as a database and stores all necessary information about your data. For more information about the data and the way it is transferred, see the section “Data” below and the chapter Deep Learning / Model.
2. Split the dataset represented by the dictionary `DLDataset`. This can be done using the procedure
   - `split_dl_dataset`.

   The resulting split will be saved over the key `split` in each sample entry of `DLDataset`.
3. Now you can preprocess your dataset. For this, you can use the procedure
   - `preprocess_dl_dataset`.

   This procedure also offers guidance on how to implement a customized preprocessing procedure.

   To use this procedure, specify the preprocessing parameters as e.g., the image size. For this latter one you should select the smallest possible image size at which the regions to segment are still well recognizable. Store all the parameter with their values in a dictionary `DLPreprocessParam`, wherefore you can use the procedure
   - `create_dl_preprocess_param`.

We recommend to save this dictionary `DLPreprocessParam` in order to have access to the preprocessing parameter values later during the inference phase.
During the preprocessing of your dataset also the images `weight_image` will be generated for the training dataset by `preprocess_dl_dataset`. They assign each class the weight ('class weights') its pixels get during training (see the section "Model Parameters and Hyperparameters" below).

Training of the model This part is about how to train a DL semantic segmentation model. The single steps are also shown in the HDevelop example `segment_pill_defects_deep_learning_2_train.hdev`.

1. A network has to be read using the operator
   - `read_dl_model`.
2. The model parameters need to be set via the operator
   - `set_dl_model_param`.

   Such parameters are e.g., *'image_dimensions'* and *'class_ids'*, see the documentation of `get_dl_model_param`.
   You can always retrieve the current parameter values using the operator
   - `get_dl_model_param`.
3. Set the training parameters and store them in the dictionary `TrainParam`. These parameters include:
   - the hyperparameters, for an overview see the section "Model Parameters and Hyperparameters" below and the chapter Deep Learning.
   - parameters for possible data augmentation (optional).
   - parameters for the evaluation during training.
   - parameters for the visualization of training results.
   - parameters for serialization.

   This can be done using the procedure
   - `create_dl_train_param`.
4. Train the model. This can be done using the procedure
   - `train_dl_model`.

   The procedure expects:
   - the model handle `DLModelHandle`
   - the dictionary with the data information `DLDataset`
   - the dictionary with the training parameter `TrainParam`
   - the information, over how many epochs the training shall run.

   In case the procedure `train_dl_model` is used, the total loss as well as optional evaluation measures are visualized.

Evaluation of the trained model In this part we evaluate the semantic segmentation model. The single steps are also shown in the HDevelop example `segment_pill_defects_deep_learning_3_evaluate.hdev`.

1. Set the model parameters which may influence the evaluation, as e.g., *'batch_size'*, using the operator
   - `set_dl_model_param`.
2. The evaluation can conveniently be done using the procedure
   - `evaluate_dl_model`.
3. The dictionary `EvaluationResult` holds the asked evaluation measures. You can visualize your evaluation results using the procedure
   - `dev_display_segmentation_evaluation`.

Inference on new images This part covers the application of a DL semantic segmentation model. The single steps are also shown in the HDevelop example `segment_pill_defects_deep_learning_4_infer.hdev`.

1. Set the parameters as e.g., *'batch_size'* using the operator
   - set_dl_model_param.
2. Generate a data dictionary DLSample for each image. This can be done using the procedure
   - gen_dl_samples_from_images.
3. Preprocess the image as done for the training. We recommend to do this using the procedure
   - preprocess_dl_samples.

   When you saved the dictionary DLPreprocessParam during the preprocessing step, you can directly use it as input to specify all parameter values.
4. Apply the model using the operator
   - apply_dl_model.
5. Retrieve the results from the dictionary *'DLResultBatch'*. The regions of the particular classes can be selected using e.g., the operator threshold on the segmentation image.

Data

We distinguish between data used for training and evaluation, and data for inference. The latter ones consist of bare images. The first ones consist of images with their information and ground truth annotations. You provide this information defining for each pixel, to which class it belongs (over the segmentation_image, see below for further explanations).

As basic concept, the model handles data over dictionaries, meaning it receives the input data over a dictionary DLSample and returns a dictionary DLResult and DLTrainResult, respectively. More information on the data handling can be found in the chapter Deep Learning / Model.

Data for training and evaluation The training data is used to train a network for your specific task. The dataset consists of images and corresponding information. They have to be provided in a way the model can process them. Concerning the image requirements, find more information in the section "Images" below. The information about the images and their ground truth annotations is provided over the dictionary DLDataset and for every sample the respective segmentation_image, defining the class for every pixel.

Classes The different classes are the sets or categories differentiated by the network. They are set in the dictionary DLDataset and are passed to the model via the operator set_dl_model_param.

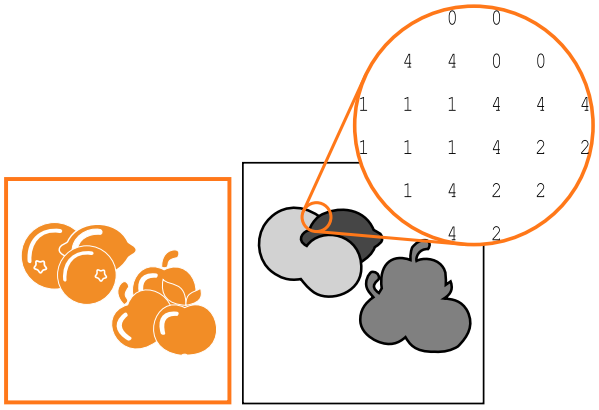
In **semantic segmentation**, we call your attention to two special cases: the class 'background' and classes declared as 'ignore':

- 'background' class: The networks treats the background class like any other class. It is also not necessary to have a background class. But if you have different classes in your dataset you are not interested in although they have to be learned by the network, you can set them all as 'background'. As a result, the class background will be more diverse. See the procedure preprocess_dl_samples for more information.
- 'ignore' classes: There is the possibility to declare one or multiple classes as 'ignore'. Pixels assigned to a 'ignore' class are ignored by the loss as well as for all measures and evaluations. Please see the section "The Network and the Training Process" in the chapter Deep Learning for more information about the loss. The network does not classify any pixel into a class declared as 'ignore'. Also, the pixels labeled to belong to such a class will be classified by the network like every other pixel into a non-'ignore' class. In the example given in the image below, this means the network will classify also the pixels of the class 'border', but it will not classify any pixel into the class 'border'. You can declare a class as 'ignore' using the parameter *'ignore_class_ids'* of set_dl_model_param.

In **edge extraction** only two classes are distinguished: 'edge' and 'background'. The class 'edge' is labeled just like a normal class. Thus, only one class is labeled and this class is called 'edge'.

DLDataset This dictionary serves as a database, this means, it stores all information about your data necessary for the network as, e.g., the names and paths to the images, the classes, ... Please see the documentation of Deep Learning / Model for the general concept and key entries. Keys only applicable for semantic segmentation concern the segmentation_image (see the entry below). Over the keys segmentation_dir and segmentation_file_name you provide the information how they are named and where they are saved.

segmentation_image In order that the network can learn, how the member of different classes look like, you tell for each pixel of every image in the training dataset to which class it belongs. This is done by storing for every pixel of the input image the class encoded as pixel value in the corresponding `segmentation_image`. These annotations are the ground truth annotations.



(1) (2)

Schema of `segmentation_image`. For visibility, gray values are used to represent numbers. (1) Input image. (2) The corresponding `segmentation_image` providing the class annotations, 0: background (white), 1: orange, 2: lemon, 3: apple, and 4: border (black) as a separate class so we can declare it as 'ignore'.

You need enough training data to split it into three subsets, one used for training, one for validation and one for testing the network. These subsets are preferably independent and identically distributed (see the section "Data" in the chapter Deep Learning. For the splitting you can use the procedure `split_dl_data_set`.

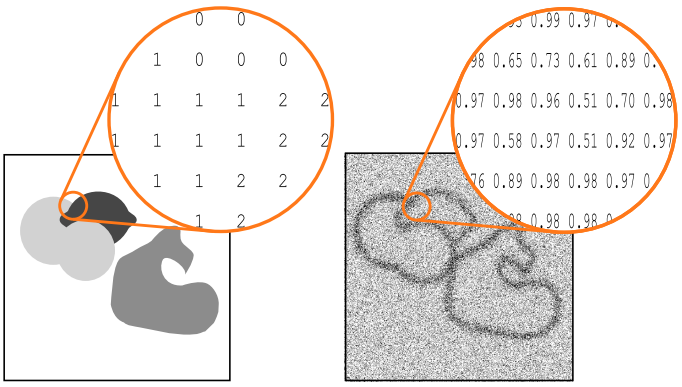
Images Regardless of the application, the network poses requirements on the images regarding the image dimensions, the gray value range, and the type. The specific values depend on the network itself, see the documentation of `read_dl_model` for the specific values of different networks. For a loaded network they can be queried with `get_dl_model_param`. In order to fulfill these requirements, you may have to preprocess your images. Standard preprocessing of an entire sample and therewith also the image is implemented in `preprocess_dl_samples`. This procedure also offers guidance on how to implement a customized preprocessing procedure.

Network output The network output depends on the task:

training As output, the operator will return a dictionary `DLTrainResult` with the current value of the total loss as well as values for all other losses included in your model.

inference and evaluation As output, the network will return a dictionary `DLResult` for every sample. For semantic segmentation, this dictionary will include for each input image the handles of the two following images:

- `segmentation_image`: An image where each pixel has a value corresponding to the class its corresponding pixel has been assigned to (see the illustration below).
- `segmentation_confidence`: An image, where each pixel has the confidence value out of the classification of the according pixel in the input image (see the illustration below).



(1) (2)

A schema over different data images. For visibility, gray values are used to represent numbers. (1) `segmentation_image`: also the pixels of classes declared as 'ignore' (see the figure above) get classified. (2) `segmentation_confidence`.

Model Parameters and Hyperparameters

Next to the general DL hyperparameters explained in Deep Learning, there is a further hyperparameter relevant for semantic segmentation:

- 'class weights', see the explanations below.

For a semantic segmentation model, the model parameters as well as the hyperparameters (with the exception of 'class weights') are set using `set_dl_model_param`. The model parameters are explained in more detail in `get_dl_model_param`.

Note, due to large memory usage, typically only small batch sizes are possible for training. As a consequence, training is rather slow and we advice to use a momentum higher than e.g., for classification. The HDevelop example `segment_pill_defects_deep_learning_2_train.hdev` provides good initial parameter values for the training of a segmentation network in HALCON.

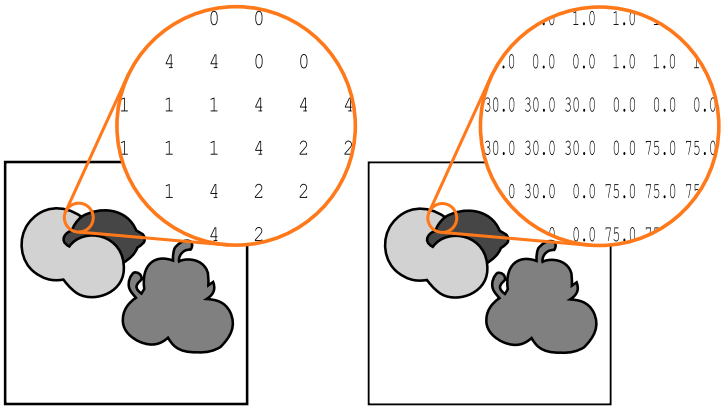
'class weights' With the hyperparameter 'class weights' you can assign each class the weight its pixels get during training. Giving the unique classes a different weight, it is possible to force the network to learn the classes with different importance. This is useful in cases where a class dominates the images, as e.g., defect detection, where the defects take up only a small fraction within an image. In such a case a network classifying every pixel as background (thus, 'not defect') would achieve generally good loss results. Assigning different weights to the distinct classes helps to re-balance the distribution. In short, you can focus the loss to train especially on those pixels you determine to be important.

The network obtains these weights over `weight_image`, an image which is created for every training sample. In `weight_image`, every pixel value corresponds to the weight the corresponding pixel of the input image gets during training. You can create these images with the help of the following two procedures:

- `calculate_dl_segmentation_class_weights` helps you to create the class weights. The procedure uses the concept of inverse class frequency weights.
- `gen_dl_segmentation_weight_images` uses the class weights and generates the `weight_image`.

This step has to be done before the training. Usually it is done during the preprocessing and it is part of the procedure `preprocess_dl_dataset`. Note, this hyperparameter is referred as `class_weights` or `ClassWeights` within procedures. An illustration, how such an image with different weights looks like, is shown in the figure below.

Note, giving a specific part of the image the weight 0.0, these pixels do not contribute to the loss (see the section "The network and its training" in Deep Learning for more information about the loss).



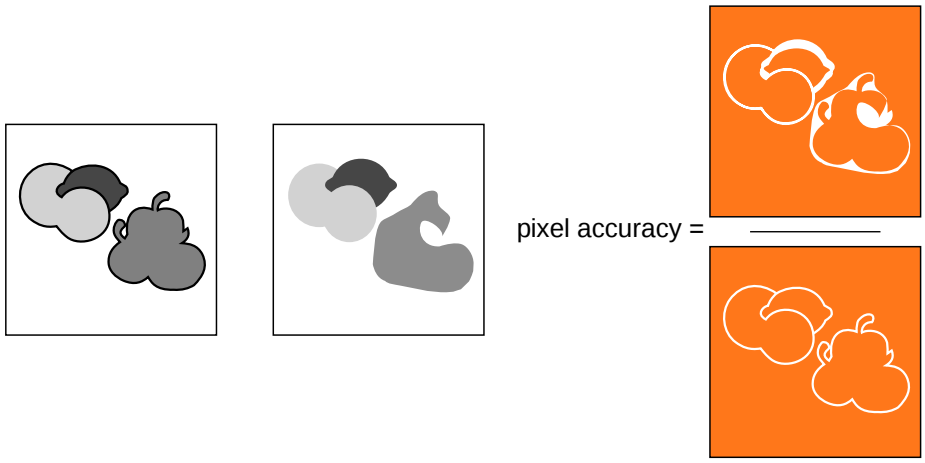
(1) (2)

Schema for `weight_image`. For visibility, gray values are used to represent numbers. (1) The `segmentation_image` defining the classes for every pixel within the image, 0: background (white), 1: orange, 2: lemon, 3: apple, and 4: border (black), declared as 'ignore'. (2) The corresponding `weight_image` providing class weights, background: 1.0, orange: 30.0, lemon: 75.0. Pixels of classes declared as 'ignore', here the class border, will be ignored and get the weight 0.0.

Evaluation measures for the Data from Semantic Segmentation

For semantic segmentation, the following evaluation measures are supported in HALCON. Note that for computing such a measure for an image, the related ground truth information is needed. All the measure values explained below for a single image (e.g., `mean_iou`) can also be calculated for an arbitrary number of images. For this, imagine a single, large image formed by the ensemble of the output images, for which the measure is computed. Note, all pixels of a class declared as 'ignore' are ignored for the computation of the measures.

`pixel_accuracy` The pixel accuracy is simply the ratio of all pixels that have been predicted with the correct class-label to the total number of pixels.



(1) (2) (3)

Visual example of the `pixel_accuracy`: (1) The `segmentation_image` defining the ground truth class for each pixel (see the section "Data" above). Pixels of a class declared as 'ignore' are drawn in black. (2) The output image, also the pixels of classes declared as 'ignore' get classified. (3) The pixel accuracy is the ratio between the total orange areas. Note, pixels labeled as part of a class declared as 'ignore' are ignored.

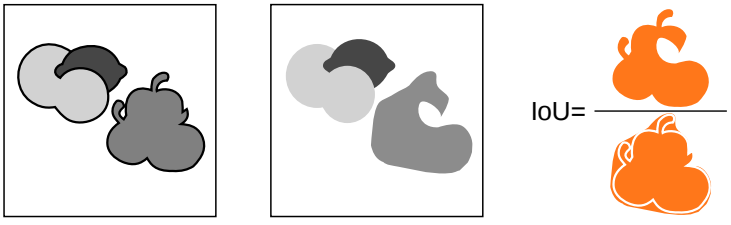
`class_pixel_accuracy` The per-class pixel accuracy considers only pixels of a single class. It is defined as the ratio between the correctly predicted pixels and the total number of pixels labeled with this class.

In case a class does not occur it gets a `class_pixel_accuracy` value of *-1* and does not contribute to the average value, `mean_accuracy`.

`mean_accuracy` The mean accuracy is defined as the averaged per-class pixel accuracy, `class_pixel_accuracy`, of all occurring classes.

`class_iou` The per-class intersection over union (IoU) gives for a specific class the ratio of correctly predicted pixels to the union of annotated and predicted pixels. Visually this is the ratio between the intersection and the union of the areas, see the image below.

In case a class does not occur it gets a `class_iou` value of *-1* and does not contribute to the `mean_iou`.



(1) (2) (3)

Visual example of the per-class IoU, `class_iou`, here for the class apple only. (1) The `segmentation_image` defining the ground truth class for each pixel (see the section “Data”). Pixels of a class declared as 'ignore' are drawn in black. (2) The output image, also the pixels of classes declared as 'ignore' get classified. (3) The intersection over union is the ratio between the intersection and the union of the areas of pixels denoted as apple. Note, pixels labeled as part of a class declared as 'ignore' are ignored.

`mean_iou` The mean IoU is defined as the averaged per-class intersection over union, `class_iou`, of all occurring classes. Note that every occurring class has the same impact on this measure, independent of the number of pixels they contain.

`frequency_weighted_iou` As for the mean IoU, the per-class IoU is calculated first. But the contribution of each occurring class to this measure is weighted by the ratio of pixels that belong to that class. Note that classes with many pixels can dominate this measure.

`pixel_confusion_matrix` The concept of a confusion matrix is explained in the section “Supervising the training” within the chapter Deep Learning. It applies for semantic segmentation, where the instances are single pixels.

Chapter 10

Develop

dev_clear_obj (Objects : : :)

Delete an iconic object from the HALCON database.

dev_clear_obj deletes iconic objects, which are no longer needed, from the HALCON database.

Attention

Never use clear_obj to clear objects in HDevelop. The operator dev_clear_obj has to be used instead.

Parameters

▷ **Objects** (input_object) ..... object(-array) ⇝ *object*
Objects to be deleted.

Result

dev_clear_obj returns always 2 (H_MSG_TRUE).

See also

clear_obj, dev_set_check, reset_obj_db

Module

Foundation

dev_clear_window (: : :)

Clear the contents of the active graphics window.

dev_clear_window clears the contents of the active graphics window including the history of the displayed iconic objects. The output parameters that have been set for this window via the context menu, the visualization parameters dialog, or the appropriate operators (e.g., with dev_set_color, dev_set_draw, etc.) remain unmodified.

The operator is equivalent to pressing the Clear button of the active graphics window.

A graphics window can be activated by calling dev_set_window or by pressing the Activate button in the tool bar of the selected graphics window.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
```

```
Number := |Regions|
dev_update_window ('off')
for i := 1 to Number by 1
  select_obj (Regions, RegionSelected, i)
  dev_clear_window ()
  dev_display (RegionSelected)
  * stop ()
endfor
```

Result

dev_clear_window returns always 2 (H_MSG_TRUE).

Possible Predecessors

dev_set_window, dev_open_window, dev_display

Possible Successors

dev_display

See also

clear_window

Module

Foundation

dev_close_inspect_ctrl (: : Variable :)

Close inspect window(s) of one or more control variables.

dev_close_inspect_ctrl is the counterpart operator to dev_inspect_ctrl. It closes the variable inspect window or windows corresponding to the variables specified in Variable. The parameter Variable can contain an expression (e.g., tuple) with several variable names. First, the operator dev_close_inspect_ctrl tries to find an inspect window with an exact match of the listed variables. If it is found, it will be closed. If no exact match is found, the operator tries to remove for every listed variable one column from the first matching inspect window. By removing the last column of an inspect window the window is closed.

As an alternative, the inspect windows can be closed by pressing the Close-button in the title bar.

Attention

This operator only works for single floating inspect windows, i.e., inspect windows that are neither docked nor tabbed.

This operator is not supported for code export.

Parameters

▷ **Variable** (input_control) ..... real(-array) ⇝ *real* / integer / string
Name of the variable which inspect window has to be closed.

Example

```
Var := 1
dev_inspect_ctrl (Var)
Var := [1,2,3,9,5,6,7,8]
Var[3] := 4
stop ()
dev_close_inspect_ctrl (Var)
```

Result

dev_close_inspect_ctrl returns always 2 (H_MSG_TRUE).

Possible Predecessors

dev_inspect_ctrl

Module

Foundation

dev_close_tool (: : ToolId :)

Close the specified floating tool window.

dev_close_tool closes the tool with the ToolId. Instead of using the ToolId, given during opening the tool, you can use the name that has to be used in dev_open_tool. In that case a arbitrary tool, which fits the parameter ToolId, is used.

Attention

This operator only works for single floating tools, i.e., tools that are neither docked nor tabbed.

This operator is not supported for code export.

Parameters

▷ **ToolId** (input_control) . . . . . dev_tool ⇝ *string*
Tool identifier.

Example

```
dev_open_tool ('matching_assistant', 0, 0, 'default','default', ToolId)
dev_close_tool (ToolId)
```

Result

If the values of the specified parameters are correct, dev_open_tool returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

dev_set_tool_geometry, dev_open_tool

Possible Successors

dev_open_tool

See also

dev_show_tool

Module

Foundation

dev_close_window (: : :)

Close the active floating graphics window.

dev_close_window closes the active floating graphics window.

The operator is equivalent to pressing the Close button in the title bar of the active window or selecting the appropriate menu entry from the Visualization menu.

A graphics window can be activated by calling dev_set_window or by pressing the Activate button in the tool bar of the selected graphics window.

Attention

This operator only works for single floating graphics windows, i.e., graphics windows that are neither docked nor tabbed.

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Example

```
* close all windows
for i := 1 to 10 by 1
  dev_close_window ()
endfor
```

```
read_image (For5, 'for5')
get_image_size (For5, Width, Height)
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_display (For5)
```

Result

`dev_close_window` returns always 2 (`H_MSG_TRUE`).

Possible Predecessors

`dev_set_window`, `dev_open_window`

Possible Successors

`dev_open_window`, `dev_get_window`

See also

`close_window`

Module

Foundation

dev_disp_text (: : String, CoordSystem, Row, Column, Color, GenParamName, GenParamValue :)

Display text in the current graphics window.

`dev_disp_text` displays text in the current graphics window at the position (`Row`,`Column`).

If only a single position is defined, one text line is displayed for each element of `String`. Also, *'\n'* will be interpreted as a newline character, i.e., a line break is performed.

If multiple positions are defined, only a single string or one string for each position is allowed in `String`. In this case, line breaks have to be forced with *'\n'*.

Newlines (*'\n'*) at the end of `String` are ignored.

The position of the text may be specified in window coordinates (`CoordSystem` = *'window'*) or in image coordinates (`CoordSystem` = *'image'*), which is useful when using zoomed images.

In addition to supplying (`Row`,`Column`) coordinates, it is also possible to pass predefined values to `Row` and `Column` to display the text at a fixed position in the window (only if `CoordSystem` = *'window'*).:

'top', *'left'*	*'top'*, *'center'*	*'top'*, *'right'*
'center', *'left'*	*'center'*, *'center'*	*'center'*, *'right'*
'bottom', *'left'*	*'bottom'*, *'center'*	*'bottom'*, *'right'*

The parameter `Color` also accepts tuples of values. In that case, the specified colors are used cyclically for every new text position or for every new line of text if a single position is used.

Generic Parameters

`dev_disp_text` may display the `String` within a box (default). This behavior and the look of the box are defined with the generic parameters in `GenParamName` and `GenParamValue`.

'box' If *'box'* is set to *'true'*, the text is written within a box. The look of the box and its optional shadow can be configured with the generic parameters below.

List of values: *'true'* and *'false'*

Default: *'true'*

'box_color' Sets the color of the box.

List of values: a string containing the color name (e.g., *'white'*, *'red'*, or *'#aa00bba0'*)

Default: *'#fce9d4'* (which is a light orange)

'shadow' If *'shadow'* is set to *'true'*, an additional shadow is displayed beneath the box (if *'box'* is *'true'*).

List of values: *'true'* and *'false'*

Default: *'true'* if *'box_color'* is set to a color without alpha value, *'false'* otherwise

'shadow_color' Sets the color of the shadow if *'shadow'* is *'true'*.

List of values: a string containing the color name (e.g., *'black'*, *'red'*, or *'#aa00bba0'*)

Default: *'#f28d26'* (which is a darker orange) if *'box_color'* is not set, *'white'* otherwise

'border_radius' Controls the roundness of the box's corners. For sharp corners set it to *0*, for smoother corners to higher values.

List of values: positive real numbers or *0*

Default: 2

'box_padding' Controls to which amount in pixels the box is extended around the text.

List of values: positive real number

Default: 0

'shadow_sigma' Controls to which amount the shadow beneath the box is blurred. Set it to *0* for a sharp shadow.

List of values: positive real number or *0*

Default: 1.5

'shadow_dx' and 'shadow_dy' Controls the offset of the shadow in column (*'shadow_dx'*) and row (*'shadow_dy'*) direction in pixels.

List of values: any real number

Default: 2

'backdrop_blur_sigma' Controls to which amount the background of the box is blurred. s only an effect if *'box'* is *'true'* and *'box_color'* is not opaque (i.e. alpha is not 255). It is recommended to use *'backdrop_blur_sigma'* with *'shadow'* set to *'false'*.

List of values: positive real number or *0*

Default: 0

'rotate_phi' Angle in degrees to rotate the displayed text.

List of values: any real number

Default: 0

'rotate_col' Column coordinate of the rotation center if *'rotate_phi'* is different to zero. Choose either a column coordinate by setting a real number or determine the rotation center on the left edge (*'text_left'*), the right edge (*'text_right'*), or in the center (*'text_center'*) of the text box.

List of values: any real number or *'text_left'*, *'text_center'*, *'text_right'*

Default: *'text_center'*

'rotate_row' Row coordinate of the rotation center if *'rotate_phi'* is different to zero. Choose either a row coordinate by setting a real number or determine the rotation center on the top edge (*'text_top'*), the bottom edge (*'text_bottom'*), or in the center (*'text_center'*) of the text box.

List of values: any real number or *'text_top'*, *'text_center'*, *'text_bottom'*

Default: *'text_center'*

'border_color' Sets the color of the border of the text box.

List of values: a string containing the color name (e.g., *'black'*, *'red'*, or *'#aa00bba0'*)

Default: *'#ffffff'*

'border_width' Controls the width of the text box border.

List of values: positive real numbers or *0*

Default: 0

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

- ▷ **String** (input_control) . . . . . string(-array) ⇝ *string*
  A tuple of strings containing the text message to be displayed. Each value of the tuple will be displayed in a single line.
  Default: 'hello'
- ▷ **CoordSystem** (input_control) . . . . . string ⇝ *string*
  If set to 'window', the text position is given with respect to the window coordinate system. If set to 'image', image coordinates are used (this may be useful in zoomed images).
  Default: 'window'
  List of values: `CoordSystem` ∈ {'window', 'image'}
- ▷ **Row** (input_control) . . . . . point.y(-array) ⇝ *integer* / real / string
  The vertical text alignment or the row coordinate of the desired text position.
  Default: 12
  List of values: `Row` ∈ {12, 'bottom', 'center', 'top'}
- ▷ **Column** (input_control) . . . . . point.x(-array) ⇝ *integer* / real / string
  The horizontal text alignment or the column coordinate of the desired text position.
  Default: 12
  List of values: `Column` ∈ {12, 'center', 'left', 'right'}
- ▷ **Color** (input_control) . . . . . string(-array) ⇝ *string*
  A tuple of strings defining the colors of the texts.
  Default: 'black'
  List of values: `Color` ∈ {'black', 'blue', 'yellow', 'red', 'green', 'cyan', 'magenta', 'forest green', 'lime green', 'coral', 'slate blue'}
- ▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
  Generic parameter names.
  Default: []
  List of values: `GenParamName` ∈ {'backdrop_blur_sigma', 'box', 'shadow', 'box_color', 'shadow_color', 'border_radius', 'box_padding', 'shadow_sigma', 'shadow_dx', 'shadow_dy', 'rotate_phi', 'rotate_col', 'rotate_row', 'border_color', 'border_width'}
- ▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
  Generic parameter values.
  Default: []
  List of values: `GenParamValue` ∈ {'true', 'false', 'white', 'red', 'forest green', 'black', 'blue', 5.0}

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
dev_disp_text ('Display some text in a box', 'window', 12, 12, \
               'black', [], [])
```

Result

If the values of the specified parameters are correct, `dev_disp_text` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

`dev_open_window`, `set_font`, `get_string_extents`

Alternatives

`disp_text`, `write_string`

See also

`get_string_extents`, `set_font`

Module

Foundation

dev_display (Object : : :)

Displays image objects in the current graphics window.

dev_display displays an iconic object (image, region, or XLD) in the active graphics window. This is equivalent to a double click on an icon variable inside the variable window.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **Object** (input_object) . . . . . object(-array) ⇝ *object*
Image objects to be displayed.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
dev_clear_window ()
dev_display (Image)
dev_set_colored (12)
dev_set_draw ('margin')
dev_display (Regions)
```

Result

If the values of the specified parameters are correct, dev_display returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Alternatives

disp_obj, disp_image, disp_region, disp_xld

See also

dev_set_color, dev_set_colored, dev_set_draw, dev_set_line_width

Module

Foundation

dev_error_var (: : ErrorVar, Mode :)

Define or undefine an error variable.

dev_error_var defines an error variable, i.e., a variable which contains the result state of the last operator call. ErrorVar will be 2 (H_MSG_TRUE) if no error had occurred. The parameter Mode specifies whether the error variable should be used (*1*) or not (*0*). If an error variable is active it will be updated each time an operator execution is finished. Thus a value is only valid until the next call of an operator. The value can be saved by assigning it to another variable (see example) or by calling dev_error_var(ErrorVar,0).

Parameters

▷ **ErrorVar** (input_control) . . . . . integer ⇝ *integer*
Name of the variable which shall contain the error status.
Default: 'ErrorVar'

▷ **Mode** (input_control) . . . . . integer ⇝ *integer*
Switch the error variable on or off.
Default: 1
List of values: Mode ∈ {0, 1}

Example

```
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
dev_error_var (Error, 1)
dev_set_check ('~give_error')
```

```
FileName := 'wrong_name'
read_image (Image, FileName)
ReadError := Error
if (ReadError != H_MSG_TRUE)
  write_string (WindowHandle, 'wrong file name: '+FileName)
endif
```

Result

If the values of the specified parameters are correct, dev_error_var returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

dev_set_check

Possible Successors

dev_set_check, if, elseif, else, assign

See also

dev_set_check, set_check

Module

Foundation

dev_get_exception_data (: : Exception, Name : Value)

Access the elements of an exception tuple.

The operator dev_get_exception_data enables to access specific items of an exception tuple that was caught by the operator catch. Except the error code that is always passed as the first element of the exception tuple all other data of the exception tuple has to be accessed exclusively via the operator dev_get_exception_data. This is due to the fact that the order and the extent of the provided data may change in future versions and may vary for the different code exports.

If an operator error occurred within HDevelop or HDevEngine, the caught exception tuple contains the data items listed below. This tuple has to be passed to the parameter Exception. The name of the requested data slot has to be passed to the parameter Name. The requested data is returned in the parameter Value. By passing a tuple of slot names to Name it is possible to request several exception data items in a single call of dev_get_exception_data. In that case a corresponding data tuple is returned in Value. For requested items that are not available an empty string ('') is returned.

Note that the operator dev_get_exception_data is meant to be used within HDevelop or HDevEngine. Some information is not available when the operator is exported.

Supported data slots:

'error_code': HALCON or user defined error code. A list of HALCON error codes (codes < 10000) can be found in the appendix of the "Extension Package Programmer's Manual".

'add_error_code': Additional HDevelop specific error code. These error codes specify whether an error was caught within the HALCON operator (code = 21000) or outside the operator, e.g., during the evaluation and assignment of the parameter expressions. In the latter case the error code specifies the error type more precisely. The relevant error codes are listed in the "HDevelop User's Guide".

'error_message': HALCON error message.

'add_error_message': Additional error message that describes the HDevelop specific error more precisely.

'program_line': Number of the program line, where the error occurred.

'operator': Name of the operator that threw the exception (if the exception was thrown in a protected procedure, '--protected--' is returned instead of the operator name).

'call_stack_depth': Depth of the call stack (if the error occurred in 'main' a depth of 1 is returned).

'procedure': Name of the procedure where the error occurred.

'user_data': User defined exception tuple. If an operator exception was caught and rethrown, this is the tuple that was appended to the caught exception tuple in the call of `throw`. In case of a completely user defined exception all tuple items except the first element containing the error code ($\geq$ 30000) are returned. If no user data was passed with the exception tuple, an empty tuple is returned. Note, that the user data tuple cannot be requested together with any other exception data in one `dev_get_exception_data` call.

Parameters

▷ **`Exception`** (input_control) . . . . . exception-array ⇝ *integer* / string
Tuple containing the exception data or user defined error codes.

▷ **`Name`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name(s) of the requested exception data.
Default: 'error_code'
List of values: `Name` ∈ {'error_code', 'add_error_code', 'error_message', 'add_error_message', 'operator', 'procedure', 'program_line', 'call_stack_depth', 'user_data'}

▷ **`Value`** (output_control) . . . . . attribute.value-array ⇝ *integer* / string
Requested exception data.

Result

If the values of the specified parameters are correct, `dev_get_exception_data` (as operator) returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

`catch`

See also

`try`, `catch`, `endtry`, `throw`, `dev_set_check`

Module

Foundation

`dev_get_preferences ( : : PreferenceNames : PreferenceValues )`

Query HDevelop preferences within a program.

`dev_get_preferences` allows to query selected preferences of HDevelop within a program. Until now, the following preferences are supported:

'graphics_window_context_menu': Returns whether a right click into the graphics window opens a context menu or not. By default the context menu is enabled.

'graphics_window_mouse_wheel': Returns whether the mouse wheel can be used to zoom the contents of the graphics window or not. By default the moue wheel is enabled.

'graphics_window_tool_tip': Returns whether pressing the Ctrl key over the graphics window shows a tool tip with the current pixel position and the gray values under the mouse cursor or not. By default the tool tip is enabled.

'suppress_handled_exceptions_dlg': Returns whether the error dialog is suppressed that is by default opened for exceptions that are thrown during program execution and that are enclosed by a `try-catch` block and can therefore be handled by an exception handler. This option is persistently stored in the HDevelop.ini file and can be configured via the Preferences dialog / General Options / Experienced User.

Attention

This operator is not supported for code export.

Parameters

▷ **`PreferenceNames`** (input_control) . . . . . attribute.name-array ⇝ *string*
Selection of the preferences.
Default: 'graphics_window_context_menu'
List of values: `PreferenceNames` ∈ {'graphics_window_context_menu', 'graphics_window_mouse_wheel', 'graphics_window_tool_tip', 'suppress_handled_exception_dlg'}

▷ **`PreferenceValues`** (output_control) . . . . . attribute.value-array ⇝ *string*
Values of the selected preferences.

See also

dev_set_preferences

Module

Foundation

dev_get_system (: : SystemQueries : SystemInformations)

Query the HDevelop system within a program.

The operator dev_get_system returns information concerning the HDevelop system parameters.

Read-Only Parameters

The following system parameters can be queried:

'call_stack': Returns the call stack, that is the procedure names and lines of the calls. See also "Thread View / Call Stack" in the HDevelop User's Guide.

Attention: This option only works when executing without JIT compilation.

'engine_environment': Returns *'HDevelop'* if it is called within HDevelop, else *'HDevEngine'*.

'jit_enabled': Returns *'true'* if the JIT Compiler is enabled, *'false'* otherwise. Not all procedures can be JIT compiled.

Attention

This operator is not supported for code export.

Parameters

▷ **SystemQueries** (input_control) ..... attribute.name-array ⇝ *string*
Desired system parameters.
Default: 'engine_environment'
List of values: SystemQueries ∈ {'call_stack', 'engine_environment', 'jit_enabled'}

▷ **SystemInformations** (output_control) ..... attribute.value-array ⇝ *string*
Current value of the system parameters.

Module

Foundation

dev_get_window (: : : WindowHandle)

Return the handle of the active graphics window.

dev_get_window returns the window handle of the active graphics window. If currently no graphics window is open, the return value is *-1*.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **WindowHandle** (output_control) ..... window ⇝ *handle*
Window handle.

Example

```
read_image (Image,'mreut')
threshold (Image, Region, 100, 200)
dev_open_window (1, 1, 200, 200, 'black', WindowID1)
dev_open_window (1, 220, 200, 200, 'black', WindowID2)
```

```
dev_get_window (CurrentWindowID)
dev_set_window (WindowID1)
dev_set_color ('blue')
dev_display (Image)
dev_display (Region)
dev_set_window(CurrentWindowID)
```

Result

If the values of the specified parameters are correct, `dev_get_window` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

`dev_close_window`

Possible Successors

`dev_set_window`

Module

Foundation

dev_inspect_ctrl (: : Variable :)

Open a window to inspect one or more control variables.

`dev_inspect_ctrl` opens an inspection window to display the contents of one or more control variables.

The standard inspection window displays the variable values with some optional statistics in a table. The window title is generated from the given parameter names. For some semantic types a special representation exists, e.g., functions are displayed as function plot by default. For more information, see the chapter "`Inspecting Variables`" in the "`HDevelop User's Guide`".

You can pass a tuple of variables in `Variable` to open an inspection window for multiple variables. It is also possible to override the default inspection mode by including a string literal as the first element of the tuple:

'tuple': Show the values in the default inspection window even if the semantic type of the variable has a special representation.

'plot': Plot the values of numeric variables against their tuple index.

'plot_xy': Plot the values of numeric variables as X-Y pairs. The number of variables must be even, with each pair of variables having the same length.

Note that no expressions are evaluated inside the call of `dev_inspect_ctrl`, but only the contained parameter names are parsed. Therefore, operations like, e.g., accessing a specific tuple range or arithmetic tuple operations need to be done beforehand.

Normally, the window contents will be updated whenever the variables change. However, the update behavior can be influenced by the operator `dev_update_var` and the `Update Variable` preference.

The window can be closed by pressing the `Close` button or by calling `dev_close_inspect_ctrl`.

Attention

This operator is not supported for code export.

Parameters

▷ **Variable** (input_control) ..... integer(-array) ⇝ *integer* / real / string
Name of the variable to be checked.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
area_center (Regions, Area, Row, Column)
dev_inspect_ctrl (Area)
dev_inspect_ctrl (['plot_xy', Row, Column])
```

Result

If the values of the specified parameters are correct, `dev_inspect_ctrl` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

See also

`dev_update_var`

Module

Foundation

dev_open_dialog (: : DialogName :)

Open a modal HDevelop dialog.

`dev_open_dialog` opens a modal HDevelop dialog type `DialogName`. `DialogName` contains the name of the dialog which should be opened.

The following dialogs, are supported:

DialogName	Dialog
'about_dialog'	About dialog
'duplicate_procedure_dialog'	Duplicate Procedure dialog
'export_dialog'	Export dialog
'open_graphics_windows_dialog'	Open Graphics Window dialog
'insert_visualization_code_dialog'	Insert Visualization Code dialog
'read_image_dialog'	Read Image dialog
'print_dialog'	Print dialog
'properties_dialog'	Properties dialog
'save_window_dialog'	Save Graphic Window dialog

Attention

This operator is not supported for code export.

Parameters

▷ **DialogName** (input_control) . . . . . string ⇝ *string*
Name of the dialog to be opened.

Example

```
dev_open_dialog ('print_dialog')
```

Result

If the values of the specified parameters are correct, `dev_open_tool` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Module

Foundation

dev_open_file_dialog (: : Filter, Mode, Path : Selection)

Opens a file selection dialog.

`dev_open_file_dialog` opens a file selection dialog for reading one file (`Mode` = *'read'*), reading one or more files (`Mode` = *'read_multi'*), writing (`Mode` = *'write'*) a file or choosing (`Mode` = *'dir'*) a directory. If `Mode` is set to *'default'* and `Filter` is set to an HALCON operator the type is set to the value the operator would use.

The `Filter` can be used to select the types of the files which shall be selected for reading or writing. The `Filter` can be set to an HALCON operator, for example `read_image`, which should read or write the selected file. In

that case, the filter is set appropriate to the files used by the HALCON operator. To select images of type JPEG and TIFF the filter could be set to *'Images (*.jpg *.tif *.tiff)'* or *'JPEG Files (*.jpg);;TIFF-Files (*.tif *.tiff)'*. In the first case, by selecting the entry 'Images' all supported images files are displayed together. In the second case it is possible to choose between JPEG Files or TIFF Files. The Filter is not case sensitive, therefore the same results are supplied by *'JPEG Files (*.jpg)'* and *'JPEG Files (*.JPG)'*. The *'All Files (*)'* is always added. To combine more than one filter you have to separate them by *';;'*.

Examples:

The filter *'Images (*.jpg *.tiff *.tif)'* lists all files with the extensions jpg, tiff and tif.

The filter *'JPEG Files (*.jpg)'* lists all files with the extension jpg.

Combination of 3 filters: *'Images (*.jpg *.tiff *.tif);;JPEG Files (*.jpg);;TIFF Files (*.tiff *.tif)'*

The `Path` can be used to set the working directory. If `Path` is set to *'default'* the working directory is set to the last one used for this type of `Mode`.

After a successful execution the `Selection` contains the selected files or file.

Attention

This operator is not supported for code export.

Parameters

- ▷ **`Filter`** (input_control) . . . . . string ⇝ *string*
  Type of file or files to select.
- ▷ **`Mode`** (input_control) . . . . . string ⇝ *string*
  Mode of the file selection dialog.
  Default: 'default'
  List of values: `Mode` ∈ {'default', 'read', 'read_multi', 'write', 'dir'}
- ▷ **`Path`** (input_control) . . . . . string ⇝ *string*
  Working directory.
  Default: 'default'
- ▷ **`Selection`** (output_control) . . . . . string(-array) ⇝ *string*
  Selected file names.
  Default: 'default'

Example

```
dev_open_file_dialog ('read_image', 'default','c:/',Selection)
read_image(ToolId, Selection)
```

Result

If the values of the specified parameters are correct, `dev_open_file_dialog` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Alternatives

`dev_open_dialog`

See also

`dev_open_dialog`

Module

Foundation

`dev_open_tool` (: : `ToolName`, `Row`, `Column`, `Width`, `Height`, `GenParamName`, `GenParamValue` : `ToolId`)

Open a HDevelop tool, a non-modal dialog, or assistant.

`dev_open_tool` opens a HDevelop tool, a non-modal dialog, or assistant. The parameter `ToolName` contains the name of the tool to be opened. This operator returns the `ToolId` of the newly created tool, which can be used by operators like `dev_set_tool_geometry`, `dev_show_tool`, `dev_close_tool` to address a specific tool. The opened tool automatically becomes active.

The tool is closed by pressing the `Close` button of the window frame, or by calling `dev_close_tool`.

The parameters GenParamName and GenParamValue are optional. For certain ToolName they can be used to specify which page is displayed and more, see below. If they are set to *[]* the tool opens with default settings.

Following the supported tools are listed, whereby they are sorted thematically.

Concerning the **Main Windows** supported values for ToolName:

- *'graphics_window'*: Graphics Window
- *'operator_window'*: Operator Window
- *'program_window'*: Program Window
- *'variable_window'*: Variable Window

Concerning the **Assistants** supported ToolName:

- *'calibration_assistant'*: Camera Calibration Assistant
  Supported values for GenParamName:
  - *'page'*: Page to be displayed. Supported values for GenParamValue:
    - *'setup'*: Setup
    - *'calibration'*: Calibration
    - *'result'*: Results
    - *'code_generation'*: Code Generation

    Example: `dev_open_tool('calibration_assistant', ..., 'page', 'result', ...)` opens the calibration assistant and shows the tab card `Results`.
- *'image_acquisition_assistant'*: Image Acquisition Assistant
  Supported values for GenParamName:
  - *'page'*: Page to be displayed. Supported values for GenParamValue:
    - *'source'*: Source
    - *'connection'*: Connection
    - *'parameters'*: Parameters
    - *'code_generation'*: Code Generation

    Example: `dev_open_tool('image_acquisition', ...,'page', 'source', ...)` opens the image acquisition assistant and shows the tab card `Source`.
- *'matching_assistant'*: Matching Assistant
  Supported values for GenParamName:
  - *'page'*: Page to be displayed. Supported values for GenParamValue:
    - *'model_creation'*: Model Creation
    - *'model_parameter'*: Model Parameter
    - *'model_use'*: Model Use
    - *'inspect'*: Inspect
    - *'code_generation'* & Code Generation

    Example: `dev_open_tool('matching_assistant', ...,'page', 'inspect', ...)` opens the matching assistant and shows the tab card `Inspect`.
- *'measure1d_assistant'*: Measure Assistant
  Supported values for GenParamName:
  - *'page'*: Page to be displayed. Supported values for GenParamValue:
    - *'input'*: Input
    - *'edges'*: Edges
    - *'fuzzy'*: Fuzzy
    - *'result'*: Results
    - *'code_generation'*& Code Generation

    Example: `dev_open_tool('measure1d_assistant', ..., 'page', 'edges', ...)` opens the measure assistant and shows the tab card `Edges`.
- *'ocr_assistant'*: OCR Assistant

Concerning the **Tools** supported values for `ToolName`:

- *'call_stack'*: Call Stack
- *'canvas'*: Canvas Window
- *'feature_histogram'*: Feature Histogram
- *'feature_inspection'*: Feature Inspection
- *'gray_histogram'*: Gray Histogram
- *'line_profile'*: Line Profile
- *'zoom_window'*: Zoom Window
- *'ocr_training_file_browser'*: OCR Training File Browser

Concerning the **Dialogs** supported values for `ToolName`:

- *'browse_examples_dialog'*: Browse Examples Dialog
- *'create_procedure_dialog'*: Create Procedure Dialog
- *'edit_procedure_interface_dialog'*: Edit Procedure Interface Dialog
- *'find_replace_dialog'*: Find Replace Dialog
- *'output_console'*: Output Console
- *'quick_navigation'*: Quick Navigation
  Supported values for `GenParamName`:
  - *'page'*: Page to be displayed. Supported values for `GenParamValue`:
    - *'invalid_lines'*: Invalid Lines
    - *'find_results'*: Find Results
    - *'breakpoints'*: Breakpoints
    - *'bookmarks'*: Bookmarks

    Example: `dev_open_tool('quick_navigation', ..., 'page', 'breakpoints', ...)` opens the quick navigation window and shows the tab card `Breakpoints`.
- *'breakpoints_dialog'*: Quick Navigation / Breakpoints
- *'bookmarks_dialog'*: Quick Navigation / Bookmarks
- *'invalid_lines_dialog'*: Quick Navigation / Invalid Lines
- *'visualization_parameters_dialog'*: Visualization Parameters Dialog
- *'help'*: Help Browser
  Supported values for `GenParamName`:
  - *'page'*: Page to be displayed. Supported values for `GenParamValue`:
    - *'contents'*: Contents
    - *'operator'*: Operator
    - *'search'*: Search
    - *'index'*: Index
    - *'bookmarks'*: Bookmarks

    Example: `dev_open_tool('help', ..., 'page', 'search', ...)` opens the help tool and selects the tab card `Search`.
  - *'operator'*: Operator whose HTML page should be displayed.
    Supported values for `GenParamValue`: Operator names (written in snake case).
    Example: `dev_open_tool('help', ...,['page','operator'], ['search','read_image'], ...)` opens the help tool, selects the tab card `Search` and displays the description of the operator `read_image`.
  - *'manual'*: Manual whose HTML page should be displayed.
    Supported values for `GenParamValue`: Manual names (including their path, both written in snake case).
    Example: `dev_open_tool('help', 'default', 'default', 'default', 'default', 'manual', 'hdevelop_users_guide/hdevelop_users_guide_0000', ToolId)` opens the manual "`HDevelop User's Guide`".

- *'preferences'*: Preferences Dialog
  Supported values for GenParamName:
  - *'page'*: Page to be displayed. Supported values for GenParamValue are listed in groups of menu items.
    For the menu User Interface supported values for GenParamValue:
    * *'user_interface/program_window'*: Program Window
    * *'user_interface/fonts'*: Fonts
    * *'user_interface/language'*: Language
    * *'user_interface/themes'*: Themes

    For the menu Procedures supported values for GenParamValue:
    * *'procedures/directories'*: Directories
    * *'procedures/external_procedures'*: External Procedures
    * *'procedures/manage_procedure_libraries'*: Manage Procedure Libraries
    * *'procedures/manage_passwords'*: Manage Passwords
    * *'procedures/procedure_use'*: Procedure Use
    * *'procedures/unresolved_procedure_calls'*: Unresolved Procedure Calls

    For the menu General Options supported values for GenParamValue:
    * *'general_options/general_options'*: General Options
    * *'general_options/experienced_user'*: Experienced User

    For the menu Visualization Settings, supported values for GenParamValue:
    * *'visualization_settings/pen'*: Pen
    * *'visualization_settings/lut'*: LUT
    * *'visualization_settings/paint'*: Paint

    For the menu Runtime Settings supported values for GenParamValue:
    * *'runtime_settings/runtime_settings'*: Runtime Settings
    * *'runtime_settings/override_operator_behavior'*: Override Operator Behavior

    For the menu Telemetry Settings supported values for GenParamValue:
    * *'telemetry_settings'*: Telemetry Settings

    Example: `dev_open_tool('preferences', ..., 'page', 'user_interface/language', ...)` shows the page language from the group User Interface.
- *'halcon_news'*: HALCON News website - an empty ToolId is returned because this simply opens the news page.

The parameters Row and Column can be used to open the tool at a specific position. Note that the offset values specified under Edit ▷ Preferences ▷ General Options ▷ General Options ▷ Window open offset are added to the row and the column index, respectively. For more information, see the chapter "Menu Edit" in the "HDevelop User's Guide". In order to apply the standard behavior, that is in general opening the tool at the last position, *'default'* can be passed.

The parameters Width and Height can be used to open the tool with a specific size. In order to apply the standard behavior, that is in general opening the tool with the last size, *'default'* can be passed. If Width and Height are less than the minimum size of the tool the minimum size is used.

Attention

This operator is not supported for code export.

Parameters

▷ **ToolName** (input_control) . . . . . string ⇝ *string*
Name of the tool to be opened and additional parameters.
List of values: ToolName ∈ {'graphics_window', 'operator_window', 'program_window', 'variable_window', 'calibration_assistant', 'image_acquisition_assistant', 'matching_assistant', 'measure1d_assistant', 'ocr_assistant', 'call_stack', 'canvas', 'feature_histogram', 'feature_inspection', 'gray_histogram', 'line_profile', 'zoom_window', 'ocr_training_file_browser', 'browse_examples_dialog', 'create_procedure_dialog', 'edit_procedure_interface_dialog', 'find_replace_dialog', 'output_console', 'quick_navigation', 'breakpoints_dialog', 'bookmarks_dialog', 'invalid_lines_dialog', 'visualization_parameters_dialog', 'help', 'preferences', 'halcon_news'}

▷ **Row** (input_control) . . . . . rectangle.origin.y ⇝ *integer* / string
Row index of upper left corner.
Default: 'default'
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . rectangle.origin.x ⇝ *integer* / string
Column index of upper left corner.
Default: 'default'
Minimum increment: 1
Recommended increment: 1

▷ **Width** (input_control) . . . . . rectangle.extent.x ⇝ *integer* / string
Width of the tool.
Default: 'default'
Minimum increment: 1
Recommended increment: 1
Restriction: Width > 0

▷ **Height** (input_control) . . . . . rectangle.extent.y ⇝ *integer* / string
Height of the tool.
Default: 'default'
Minimum increment: 1
Recommended increment: 1
Restriction: Height > 0

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters.
Default: []

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string*
Values of the generic parameters.
Default: []

▷ **ToolId** (output_control) . . . . . dev_tool ⇝ *string*
Tool identifier.

Example

```
dev_open_tool ('zoom_window',0, 0, Width, Height, [], [], ToolId)
dev_close_tool (ToolId)
```

Result

If the values of the specified parameters are correct, `dev_open_tool` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Possible Successors

`dev_set_tool_geometry`, `dev_show_tool`, `dev_close_tool`

Module

Foundation

dev_open_window (: : Row, Column, Width, Height, Background : WindowHandle)

Open a new graphics window.

`dev_open_window` opens a new graphics window, which can be used to display iconic objects like images, regions, and lines as well as to perform textual output. This window automatically becomes active, which means that all output (`dev_display` and automatic display of operator results) is redirected to this window. This is shown by the lucent lamp on the `Active` button.

Each graphics window is identified by its `WindowHandle`. This logical number is displayed in the title bar of the graphics window and is required as input parameter for some operators like `disp_image`, `disp_region`, `draw_circle`, `get_mbutton`, `write_string`, and `set_rgb`.

The graphics window is closed by pressing the Close button of the window frame, via the Visualization menu, or by calling dev_close_window.

Graphics Window Position

By default, the graphics window opens within the canvas window. If this is not desired, select Restore Default Layout use ▷ Docked Graphics Window from the Window menu in HDevelop. See also "Canvas Window" in the "HDevelop User's Guide".

The parameters Row and Column are used to pass the position of the window. Its reference point (0, 0) on the canvas is visualized by a cross.

For floating windows, the reference point is by default the upper left corner of the HDevelop main window. You can change this reference point, so that, for example, the graphics window position aligns with the upper left corner of the screen. To do so, open the HDevelop preferences and choose the desired Origin of coordinates under General options.

Moreover, you can adapt the window position via the setting Window open offset. These offset values are added to the row and the column index, respectively. See also "Menu Window open offset" in the "HDevelop User's Guide".

Graphics Window Visualization

The background of the created graphics window is set to the color specified in Background. This parameter is not available for the operator open_window. There, the same behavior can be achieved by calling set_window_attr(::'background_color',Background:) in advance.

As a default, the visible image part in the graphics window (viewport) is set in a way that images are displayed without clipping and fitted completely into the window. The image part is adapted to the window's size according to this rule for the first image that is displayed after a program reset or the loading of a new program or if the current image has a different image size than the image that was displayed before. The size of the window is not adapted automatically, hence, if the aspect ratio of the image differs from that of the window, the image is distorted to fit into the window. This can be changed via the Window Size menu.

The visible image part can be changed interactively by spinning the mouse wheel, using the Move or Zoom mode, via the Image Size menu, with the help of the Zoom tab card on the Visualization Parameters dialog, or with the operator dev_set_part.

The display parameters of the graphics window can be specified via its context menu, the Visualization menu, the Visualization Parameters dialog, or the appropriate HDevelop operators like dev_set_color, dev_set_line_width, dev_set_draw. Depending on the Apply Immediately preference, the parameter changes are applied to the lastly displayed object or apply from now on for all following objects. In contrast to the standard HALCON window operators, the new settings are also used for all new graphics windows.

Graphics Window History

Each graphics window manages a history that contains the

- objects and
- display parameters

that have been displayed or changed since the most recent clear action or display of a full image. This history is used if a redraw of the window is triggered, e.g., after a change of the window's size, in order to reconstruct the complete window contents. Other iconic output that was displayed using HALCON operators like disp_image or disp_region, text (write_string), or geometric objects (disp_line, disp_circle, etc.) are **not** part of the history, and can therefore **not** be redrawn. Only the object classes image, region, and XLD that are displayed with the HDevelop operator dev_display or by HDevelop actions like double clicking on an icon are part of the history.

Pressing the Clear button clears the graphics window contents and the history of the window. This can also be achieved by using the operator dev_clear_window. This will not affect the current display parameters.

Further Information

Additional information about the underlying HALCON window can be found at open_window. For more information about the graphics window, please have a look at the "HDevelop User's Guide".

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics

operators into the different programming languages, in the "HDevelop User's Guide", see the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **Row** (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Row index of upper left corner.
Default: 0
Value range: 0 ≤ Row
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column index of upper left corner.
Default: 0
Value range: 0 ≤ Column
Minimum increment: 1
Recommended increment: 1

▷ **Width** (input_control) . . . . . rectangle.extent.x ⇝ *integer*
Width of the window.
Default: 512
Minimum increment: 1
Recommended increment: 1
Restriction: Width > 0 || Width == -1

▷ **Height** (input_control) . . . . . rectangle.extent.y ⇝ *integer*
Height of the window.
Default: 512
Minimum increment: 1
Recommended increment: 1
Restriction: Height > 0 || Height == -1

▷ **Background** (input_control) . . . . . integer ⇝ *integer* / string
Color of the background of the new window.
Default: 'black'

▷ **WindowHandle** (output_control) . . . . . window ⇝ *handle*
Window handle.

Example

```
dev_close_window ()
read_image (For5, 'for5')
get_image_size (For5, Width, Height)
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_display (For5)
dev_set_lut ('rainbow')
dev_display (For5)
stop ()
dev_set_lut ('default')
dev_display (For5)
stop ()
dev_set_part (100, 100, 300, 300)
dev_display (For5)
```

Result

If the values of the specified parameters are correct, dev_open_window returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Successors

dev_display, dev_set_lut, dev_set_color, dev_set_draw, dev_set_part

Alternatives

open_window

See also

query_color

Module

Foundation

dev_set_check (: : Mode :)

Specify the error handling in HDevelop.

dev_set_check specifies how HDevelop has to react in case of an error, i.e., if the return state of an operator is not 2 (H_MSG_TRUE).

If Mode has the value *'give_error'*—which is the system default—an erroneous operator call will throw an exception, that can be caught within the HDevelop program by the catch statement. However, if there is no surrounding try-catch block in the HDevelop program and the program is executed within HDevelop, the program execution stops at the erroneous operator and an error message box is opened to display the error text. In addition, the appropriate operator call is entered into the *Operator Window*, so that the user can easily edit and possibly correct the parameters of the erroneous operator call. If the procedure was called from HDevEngine and the exception is not caught within the HDevelop program, an HDevEngineException object is thrown and the procedure is left.

If Mode is set to *'~give_error'*, the error will be ignored and the program continues with the next operator. dev_set_check('~give_error') is intended to be used in connection with dev_error_var, which allows to check the result state that is returned by the operator calls.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **Mode** (input_control) ..... string ⇝ *string*
Mode of error handling.
Default: 'give_error'

Example

```
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
dev_error_var (Error, 1)
dev_set_check ('~give_error')
FileName := 'wrong_name'
read_image (Image, FileName)
ReadError := Error
if (ReadError != H_MSG_TRUE)
  write_string (WindowHandle, 'wrong file name: '+FileName)
endif
* Now the program will stop with an exception
dev_set_check ('give_error')
read_image (Image, FileName)
```

Result

If the values of the specified parameters are correct, dev_set_check returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Successors

dev_error_var

See also

set_check, try, catch, endtry

Module

Foundation

dev_set_color (: : ColorName :)

Set one or more output colors.

dev_set_color defines the color(s) that are used to display regions, XLDs, and other geometrical objects in the graphics windows. The available colors can be queried with the operator query_color. In addition, the ColorName may be specified as hexadecimal RGB triplet or RGBA quadruplet in the form *'#rrggbb'* and *'#rrggbbaa'*. 'rr', 'gg', 'bb', and 'aa' are hexadecimal numbers between '00' and 'ff', respectively. 'aa' denotes the alpha value of a color and can be used to display transparent regions.

For more information see the description of the operator set_color. However, in contrast to that operator the color setting is also used for all new graphics windows that are opened afterwards.

These color settings remain valid until dev_set_color or dev_set_colored is called or until the color settings are modified interactively.

Color name	75% alpha	50% alpha	25% alpha
'black'	'#000000c0'	'#00000080'	'#00000040'
'white'	'#ffffffc0'	'#ffffff80'	'#ffffff40'
'red'	'#ff0000c0'	'#ff000080'	'#ff000040'
'green'	'#00ff00c0'	'#00ff0080'	'#00ff0040'
'blue'	'#0000ffc0'	'#0000ff80'	'#0000ff40'
'dim gray'	'#696969c0'	'#69696980'	'#69696940'
'gray'	'#bebebec0'	'#bebebe80'	'#bebebe40'
'light gray'	'#d3d3d3c0'	'#d3d3d380'	'#d3d3d340'
'cyan'	'#00ffffc0'	'#00ffff80'	'#00ffff40'
'magenta'	'#ff00ffc0'	'#ff00ff80'	'#ff00ff40'
'yellow'	'#ffff00c0'	'#ffff0080'	'#ffff0040'
'medium slate blue'	'#7b68eec0'	'#7b68ee80'	'#7b68ee40'
'coral'	'#ff7f50c0'	'#ff7f5080'	'#ff7f5040'
'slate blue'	'#6a5acdc0'	'#6a5acd80'	'#6a5acd40'
'spring green'	'#00ff7fc0'	'#00ff7f80'	'#00ff7f40'
'orange red'	'#ff4500c0'	'#ff450080'	'#ff450040'
'dark olive green'	'#556b2fc0'	'#556b2f80'	'#556b2f40'
'pink'	'#ffc0cbc0'	'#ffc0cb80'	'#ffc0cb40'
'cadet blue'	'#5f9ea0c0'	'#5f9ea080'	'#5f9ea040'
'goldenrod'	'#daa520c0'	'#daa52080'	'#daa52040'
'orange'	'#ffa500c0'	'#ffa50080'	'#ffa50040'
'gold'	'#ffd700c0'	'#ffd70080'	'#ffd70040'
'forest green'	'#228b22c0'	'#228b2280'	'#228b2240'
'cornflower blue'	'#6495edc0'	'#6495ed80'	'#6495ed40'
'navy'	'#000080c0'	'#00008080'	'#00008040'
'turquoise'	'#40e0d0c0'	'#40e0d080'	'#40e0d040'
'dark slate blue'	'#483d8bc0'	'#483d8b80'	'#483d8b40'
'light blue'	'#add8e6c0'	'#add8e680'	'#add8e640'
'indian red'	'#cd5c5cc0'	'#cd5c5c80'	'#cd5c5c40'
'violet red'	'#d02090c0'	'#d0209080'	'#d0209040'
'light steel blue'	'#b0c4dec0'	'#b0c4de80'	'#b0c4de40'
'medium blue'	'#0000cdc0'	'#0000cd80'	'#0000cd40'
'khaki'	'#f0e68cc0'	'#f0e68c80'	'#f0e68c40'
'violet'	'#ee82eec0'	'#ee82ee80'	'#ee82ee40'
'firebrick'	'#b22222c0'	'#b2222280'	'#b2222240'
'midnight blue'	'#191970c0'	'#19197080'	'#19197040'

Examples of possible color strings

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **ColorName** (input_control) ..... string(-array) ⇝ *string*
Output color names.
Default: 'white'
Suggested values: ColorName ∈ {'white', 'black', 'gray', 'red', 'green', 'blue', '#003075', '#e53019', '#ffb529'}

Example

```
read_image(Image,'mreut')
dev_set_draw('fill')
dev_set_color('red')
threshold(Image,Region,180,255)
dev_set_color('green')
threshold(Image,Region,0,179)
```

Result

If the values of the specified parameters are correct, dev_set_color returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

dev_open_window, query_color, query_all_colors

Possible Successors

dev_display

Alternatives

dev_set_colored

See also

dev_set_draw, dev_set_line_width, set_color

Module

Foundation

dev_set_colored (: : NumColors :)

Set multiple output colors.

dev_set_colored allows to display tuples of regions, XLDs, and other geometrical objects in the graphics windows in different colors using a set of NumColors predefined colors. Valid values for NumColors can be queried with the operator query_colored.

For more information see the description of the operator set_colored. However, in contrast to that operator the color setting is also used for all new graphics windows that are opened afterwards.

These color settings remain valid until dev_set_color or dev_set_colored is called or until the color setting are modified interactively.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **NumColors** (input_control) . . . . . integer ⇝ *integer*
Number of output colors.
Default: 6
List of values: NumColors ∈ {3, 6, 12}

Example

```
read_image(Image,'monkey')
threshold(Image,Region,128,255)
dev_set_colored(6)
connection(Region,Regions)
```

Result

If the values of the specified parameters are correct, dev_set_colored returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

dev_open_window

Possible Successors

dev_display

Alternatives

dev_set_color

See also

dev_set_draw, dev_set_line_width, set_colored

Module

Foundation

dev_set_contour_style (: : Style :)

Define the contour display fill style.

dev_set_contour_style defines the fill style of contour displays. The following values are supported for Style:

- *'stroke'*: Only the line of the contour gets displayed.
- *'fill'*: The area enclosed by the contour is filled.
- *'stroke_and_fill'*: The line of the contour gets displayed and the enclosed area filled. especially makes a difference for larger line widths, see set_line_width.

For all styles the current drawing color is used.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **Style** (input_control) . . . . . string ⇝ *string*
Fill style for contour display.
Default: 'stroke'
List of values: Style ∈ {'stroke', 'fill', 'stroke_and_fill'}

Result

If the values of the specified parameters are correct, dev_set_contour_style returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Successors

dev_display

See also

set_contour_style

Module

Foundation

dev_set_draw (: : DrawMode :)

Define the region fill mode.

dev_set_draw defines the fill mode for regions. If DrawMode is set to *'fill'*, regions are displayed filled, if set to *'margin'*, only contours are displayed. In the *'margin'* mode, the appearance of the contours can be affected by dev_set_line_width and set_line_style.

For more information see the description of the operator set_draw. However, in contrast to that operator the draw mode is also used for all new graphics windows that are opened afterwards.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **DrawMode** (input_control) ..... string ⇝ *string*
Fill mode for region output.
Default: 'fill'
List of values: DrawMode ∈ {'fill', 'margin'}

Example

```
read_image(Image,'monkey')
threshold(Image,Region,128,255)
dev_clear_window ()
dev_set_color('red')
dev_set_draw('fill')
dev_display(Region)
dev_set_color('white')
dev_set_draw('margin')
dev_display(Region)
```

Result

If the values of the specified parameters are correct, dev_set_draw returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Successors

dev_set_line_width, dev_display

See also

set_draw

Module

Foundation

dev_set_line_width (: : LineWidth :)

Define the line width for region contour output.

dev_set_line_width defines the line width (in pixel) that is used to display region contours (in *'margin'* mode), XLDs, and other geometric output (e.g., disp_region, disp_line, etc.).

For more information see the description of the operator set_line_width. However, in contrast to that operator the new line width is also used for all new graphics windows that are opened afterwards.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **LineWidth** (input_control) ..... integer ⇝ *integer*
Line width for region output in contour mode.
Default: 1
Restriction: LineWidth >= 1

Example

```
read_image(Image,'monkey')
threshold(Image,Region,128,255)
dev_set_draw('margin')
dev_set_line_width(5)
dev_clear_window ()
dev_display(Region)
```

Result

If the values of the specified parameters are correct, dev_set_line_width returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Successors

dev_display

See also

set_line_width, query_line_width

Module

Foundation

dev_set_lut (: : LutName :)

Set "look-up-table" (lut).

dev_set_lut sets the look-up-table of the active graphics window. A look-up-table defines the transformation of a "gray value" from an one-channel-image into a gray value or color on the screen. query_lut lists the names of all look-up-tables.

For more information see the description of the operator set_lut. However, in contrast to that operator the new look-up-table is also used for all new graphics windows that are opened afterwards.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **LutName** (input_control) ..... filename.read(-array) ⇝ *string* / real / integer
Name of look-up-table, values of look-up-table (RGB) or file name.
Default: 'default'
Suggested values: LutName ∈ {'default', 'linear', 'inverse', 'sqr', 'inv_sqr', 'cube', 'inv_cube', 'sqrt', 'inv_sqrt', 'cubic_root', 'inv_cubic_root', 'color1', 'color2', 'color3', 'color4', 'three', 'six', 'twelve',

'twenty_four', 'rainbow', 'temperature', 'cyclic_gray', 'cyclic_temperature', 'hsi', 'change1', 'change2', 'change3', 'jet', 'inverse_jet', 'batlow', 'inverse_batlow'}
File extension: .lut

Example

```
read_image(Image,'mreut')
dev_set_lut('inverse')
* For true color only:
dev_display(Image)
```

Result

If the values of the specified parameters are correct, dev_set_lut returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Successors

dev_display

See also

set_lut

Module

Foundation

dev_set_paint (: : Mode :)

Define the gray value output mode.

dev_set_paint defines the output mode that is used for displaying image objects in the graphics window.

For a detailed description of all possible options see set_paint. However, in contrast to that operator the display mode is also used for all new graphics windows that are opened afterwards.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **Mode** (input_control) . . . . . string-array ⇝ *string* / integer
Grayvalue output name. Additional parameters possible.
Default: 'default'
Suggested values: Mode ∈ {'default', '3d_plot'}

Example

```
read_image(Image,'fabrik')
dev_set_paint('3d_plot')
dev_display(Image)
```

Result

If the values of the specified parameters are correct, dev_set_paint returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

dev_open_window

Possible Successors

dev_set_color, dev_display

See also

set_paint

Module

Foundation

dev_set_part (: : Row1, Column1, Row2, Column2 :)

Modify the displayed image part.

dev_set_part is used to set the part of the image that is displayed in the graphics window. The parameters Row1 and Column1 specify the upper left corner, Row2 and Column2 the lower right corner of the image part to display.

For more information see the description of the operator set_part. In addition, if Row1 is larger than Row2 or Column1 larger than Column2, the zooming in the particular dimension will be reset to show the complete height and/or width of the image. Please note that this is not possible with the operator set_part outside HDevelop.

In addition, note that the part is automatically reset, if a new program is loaded, a program reset is performed, or a new image with a different image size is displayed.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **Row1** (input_control) ..... rectangle.origin.y ⇝ *integer*
Row of the upper left corner of the chosen image part.
Default: 0

▷ **Column1** (input_control) ..... rectangle.origin.x ⇝ *integer*
Column of the upper left corner of the chosen image part.
Default: 0

▷ **Row2** (input_control) ..... rectangle.corner.y ⇝ *integer*
Row of the lower right corner of the chosen image part.
Default: 128

▷ **Column2** (input_control) ..... rectangle.corner.x ⇝ *integer*
Column of the lower right corner of the chosen image part.
Default: 128

Example

```
read_image (Image, 'fabrik')
for i := 1 to 240 by 10
  dev_set_part (i, i, 511-i, 511-i)
  dev_display (Image)
endfor
dev_set_part (1, 1, -1, -1)
dev_display (Image)
```

Result

If the values of the specified parameters are correct, dev_set_part returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Successors

dev_display

See also

set_part

Module

Foundation

dev_set_preferences (: : PreferenceNames, PreferenceValues :)

Set HDevelop preferences within a program.

dev_set_preferences allows to set selected preferences of HDevelop within a program. The following preferences are supported:

'graphics_window_context_menu': Controls whether a right click into the graphics window opens a context menu or not. By default the context menu is enabled. Disabling the context menu may be sensible if the right mouse button is used for controlling some kind of navigation in the graphics window, e.g., for moving or zooming 3D objects.

List of values: *'false'*, *'true'*.

Default: *'true'*.

'graphics_window_mouse_wheel': Controls whether the mouse wheel can be used to zoom the contents of the graphics window or not. By default the mouse wheel is enabled. Disabling the mouse wheel may be sensible if 3D objects are displayed and if the zooming into the objects is realized interactively with the help of 3D display operators.

List of values: *'false'*, *'true'*.

Default: *'true'*.

'graphics_window_tool_tip': Controls whether pressing the Ctrl key over the graphics window shows a tool tip with the current pixel position and the gray values under the mouse cursor or not. By default the tool tip is enabled. Disabling the tool tip may be sensible if the state of the Ctrl key shall be evaluated for program based user interactions, e.g., for manipulating the view on 3D objects.

List of values: *'false'*, *'true'*.

Default: *'true'*.

'suppress_handled_exceptions_dlg': Controls whether the error dialog should be suppressed that is by default opened for exceptions that are thrown during program execution and that are enclosed by a try-catch block and can therefore be handled by an exception handler. This option is persistently stored into the HDevelop.ini file and can also be configured via the Preferences dialog / General Options / Experienced User. If an exception is thrown by a program line that is not enclosed by a try-catch block, an error message dialog is always opened.

List of values: *'false'*, *'true'*.

Default: *'false'* (Any changes are persistently saved).

Attention

This operator is not supported for code export.

Parameters

▷ **PreferenceNames** (input_control) . . . . . attribute.name-array ⇝ *string*
Selection of the preferences.
Default: 'graphics_window_context_menu'
List of values: PreferenceNames ∈ {'graphics_window_context_menu', 'graphics_window_mouse_wheel', 'graphics_window_tool_tip', 'suppress_handled_exception_dlg'}

▷ **PreferenceValues** (input_control) . . . . . attribute.value-array ⇝ *string*
New values for the selected preferences.
Default: 'false'
List of values: PreferenceValues ∈ {'true', 'false'}

See also

dev_get_preferences

Module

Foundation

dev_set_shape (: : Shape :)

Define the region output shape.

dev_set_shape defines the shape that is used for displaying regions. The available shapes can be queried with query_shape.

These shapes are supported:

'original': The shape is displayed unchanged. Nevertheless modifications via parameters like dev_set_line_width can take place. This is also true for all other modes.

'outer_circle': Each region is displayed by the smallest surrounding circle. (See smallest_circle.)

'inner_circle': Each region is displayed by the largest included circle. (See inner_circle.)

'ellipse': Each region is displayed by an ellipse with the same moments and orientation (See elliptic_axis.)

'rectangle1': Each region is displayed by the smallest surrounding rectangle parallel to the coordinate axes. (See smallest_rectangle1.)

'rectangle2': Each region is displayed by the smallest surrounding rectangle. (See smallest_rectangle2.)

'convex': Each region is displayed by its convex hull (See shape_trans.)

'icon': Each region is displayed by the icon set with set_icon in the center of gravity.

For more information see the description of the operator set_shape. However, in contrast to that operator the selected shape is also used for all new graphics windows that are opened afterwards.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **Shape** (input_control) ..... string ⇝ *string*
Region output mode.
Default: 'original'
List of values: Shape ∈ {'original', 'convex', 'outer_circle', 'inner_circle', 'rectangle1', 'rectangle2', 'ellipse', 'icon'}

Example

```
read_image(Image,'monkey')
threshold(Image,Region,128,255)
connection(Region,Regions)
dev_set_shape('rectangle1')
dev_set_draw('margin')
dev_set_line_width(5)
dev_clear_window()
dev_display(Regions)
```

Result

If the values of the specified parameters are correct, dev_set_shape returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Successors

dev_display, dev_set_color

See also

set_shape, dev_set_line_width

Module

Foundation

dev_set_system (: : SystemParameter, Value :)

Change HDevelop system parameter within a program.

The operator dev_set_system sets HDevelop system parameters.

Available options for SystemParameter:

'jit_enabled': Determines whether the JIT Compiler is enabled (*'true'*) or not (*'false'*).
Note, not all procedures can be JIT compiled.
List of values: *'true'*, *'false'*
Default: *'false'*

'all': Reset all system parameters to their default value. To do so, set `Value` to *'default'*.
List of values: *'default'*

Attention

This operator is not supported for code export.

Parameters

▷ **`SystemParameter`** (input_control) . . . . . attribute.name ⇝ *string*
Name of the system parameter to be changed.
Default: 'jit_enabled'
List of values: `SystemParameter` ∈ {'jit_enabled', 'all'}

▷ **`Value`** (input_control) . . . . . attribute.value ⇝ *integer* / string
New value of the system parameter.
Default: 'false'
List of values: `Value` ∈ {'false', 'true', 'default'}

Module

Foundation

`dev_set_tool_geometry` `( : : ToolId, Row, Column, Width, Height : )`

Sets the position and size of the specified tool.

`dev_set_tool_geometry` sets the position and dimension of the tool with the id `ToolId`.

Instead of using the `ToolId`, given during opening the tool, you can use the name that has to be used in `dev_open_tool`. In that case a arbitrary tool, which fits the parameter `ToolId`, is used.

The parameters `Row` and `Column` can be used to modify the position of the tool. Note that the offset values specified under `Edit ▷ Preferences ▷ General Options ▷ General Options ▷ Window open offset` are added to the row and the column index, respectively. For more information, see the chapter "Menu `Edit`" in the "`HDevelop User's Guide`". In order to unmodify the position *'default'* can be passed.

The parameters `Width` and `Height` can be used to modify the size of the tools. In order to unmodify the size *'default'* can be passed. If `Width` and `Height` are less than the minimum size of the tool the minimum size is used.

Attention

This operator is not supported for code export.

Parameters

▷ **`ToolId`** (input_control) . . . . . dev_tool ⇝ *string*
Tool identifier.

▷ **`Row`** (input_control) . . . . . rectangle.origin.y ⇝ *integer* / string
Row index of upper left corner.
Default: 'default'
Minimum increment: 1
Recommended increment: 1

▷ **`Column`** (input_control) . . . . . rectangle.origin.x ⇝ *integer* / string
Column index of upper left corner.
Default: 'default'
Minimum increment: 1
Recommended increment: 1

▷ **Width** (input_control) . . . . . rectangle.extent.x ⇝ *integer* / string
Width of the tool.
Default: 'default'
Minimum increment: 1
Recommended increment: 1
Restriction: Width > 0

▷ **Height** (input_control) . . . . . rectangle.extent.y ⇝ *integer*
Height of the tool.
Default: 'default'
Minimum increment: 1
Recommended increment: 1
Restriction: Height > 0

Example

```
dev_open_tool ('zoom_window', 0, 0, Width, 'default', ToolId)
dev_close_tool (ToolId)
```

Result

If the values of the specified parameters are correct, dev_open_tool returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

dev_open_tool

Possible Successors

dev_show_tool, dev_close_tool

Alternatives

dev_show_tool

See also

dev_show_tool

Module

Foundation

dev_set_window (: : WindowHandle :)

Activate a graphics window.

dev_set_window activates the graphics window with the passed window handle. This is equivalent to pressing the Active button in the tool bar of the graphics window.

Attention

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "HDevelop User's Guide" the chapter Code Export ▷ General Aspects of Code Generation ▷ Graphics Windows.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

Example

```
dev_open_window (1, 1, 200, 200, 'black', WindowID1)
dev_open_window (1, 220, 200, 200, 'black', WindowID2)
read_image(Image,'monkey')
dev_set_window(WindowID1)
dev_display(Image)
dev_set_window(WindowID2)
dev_display(Image)
```

Result

If the values of the specified parameters are correct, `dev_set_window` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

`dev_open_window`

Possible Successors

`dev_display`

Module

Foundation

dev_set_window_extents (: : Row, Column, Width, Height :)

Change position and size of the active floating graphics window.

`dev_set_window_extents` changes the position and/or the size of the currently active floating graphics window.

The parameters `Row` and `Column` specify the new position (upper left corner) of the window. Note that the offset values specified under `Edit` ▷ `Preferences` ▷ `General Options` ▷ `General Options` ▷ `Window open offset` are added to the row and the column index, respectively. For more information, see the chapter "Menu `Edit`" in the "`HDevelop User's Guide`". Negative coordinates of the position are ignored, i.e., in this direction the window will not be moved.

The parameters `Width` and `Height` specify the new size of the window. This is the size of the inner part that actually displays the iconic objects. If one of the two values is negative, this dimension will remain unchanged.

Attention

This operator only works for single floating graphics windows, i.e., graphics windows that are neither docked nor tabbed.

Never use `set_window_extents` to change the size and position of an HDevelop graphics window. The operator `dev_set_window_extents` has to be used instead.

Using the code export feature of HDevelop, the code that is generated for this operator may have a different behavior than the related HALCON operator. For a detailed description of the code export of HDevelop graphics operators into the different programming languages see in the "`HDevelop User's Guide`" the chapter `Code Export` ▷ `General Aspects of Code Generation` ▷ `Graphics Windows`.

Parameters

▷ **Row** (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Row index of upper left corner.
Default: 0
Minimum increment: 1
Recommended increment: 1
Restriction: Row >= 0 || Row == -1

▷ **Column** (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column index of upper left corner.
Default: 0
Minimum increment: 1
Recommended increment: 1
Restriction: Column >= 0 || Column == -1

▷ **Width** (input_control) . . . . . rectangle.extent.x ⇝ *integer*
Width of the window.
Default: 256
Minimum increment: 1
Recommended increment: 1
Restriction: Width > 0 || Width == -1

▷ **Height** (input_control) ..... rectangle.extent.y ⇝ *integer*
Height of the window.
Default: 256
Minimum increment: 1
Recommended increment: 1
Restriction: Height > 0 || Height == -1

Example

```
dev_close_window ()
read_image (For5, 'for5')
get_image_size (For5, Width, Height)
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_display (For5)
stop ()
dev_set_window_extents (-1,-1,Width/2,Height/2)
dev_display (For5)
stop ()
dev_set_window_extents (200,200,-1,-1)
```

Result

If the values of the specified parameters are correct, `dev_set_window_extents` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

Possible Successors

`dev_display`, `dev_set_lut`, `dev_set_color`, `dev_set_draw`, `dev_set_part`

See also

`set_window_extents`

Module

Foundation

dev_show_tool (: : ToolId, Action :)

Shows the specified tool.

`dev_show_tool` executes the action `Action` on the tool `ToolId`

Instead of using the `ToolId`, given during opening the tool, you can use the name that has to be used in `dev_open_tool`. In that case an arbitrary tool, which fits the parameter `ToolId`, is used.

`Action`

'show': Activates the tool.

'normal': Shows the tool, brings it to the front and activates it.

'minimize': Minimizes the tool.

'maximize': Maximizes the tool.

'hide': Hides the tool.

Attention

This operator is not supported for code export.

Parameters

▷ **ToolId** (input_control) ..... dev_tool ⇝ *string*
Tool identifier.

▷ **Action** (input_control) ..... string ⇝ *string*
Action to execute.

Example

```
dev_open_tool ('matching_assistant', 0, 0, 'default','default', ToolId)
dev_show_tool (ToolId, 'minimize')
```

Result

If the values of the specified parameters are correct, dev_show_tool returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Predecessors

dev_set_tool_geometry, dev_open_tool

See also

dev_set_tool_geometry

Module

Foundation

dev_update_pc (: : DisplayMode :)

Switches the update of the PC during program execution on or off.

dev_update_pc specifies the behavior of the PC during program execution. If DisplayMode is set to *'on'*, which is the default, within the selected procedure the PC is always displayed left to the currently executed operator. In addition the program text is scrolled—if necessary—so that the current operator is visible.

If the mode is *'off'* the PC is not visible during program execution and the program text will not be scrolled automatically. When the program stops the PC becomes visible again and the listing is scrolled to the current PC position.

For measuring the execution time of a sequence of operators all update options should be switched off in order to reduce the influence of the runtime of GUI updates in HDevelop. For this the operators dev_update_pc, dev_update_time, dev_update_var, and dev_update_window or the procedures dev_update_on and dev_update_off can be used.

This option can also be controlled via the Preferences dialog: Edit ▷ Preferences ▷ Runtime Settings ▷ Update Program Counter.

Attention

This operator is not supported for code export.

Parameters

▷ **DisplayMode** (input_control) . . . . . string ⇝ *string*
Mode for runtime behavior.
Default: 'off'
List of values: DisplayMode ∈ {'on', 'off'}

Result

If the values of the specified parameters are correct, dev_update_pc returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

See also

dev_update_time, dev_update_window, dev_update_var

Module

Foundation

dev_update_time (: : DisplayMode :)

Switch time measurement for operators on or off.

dev_update_time controls whether the execution time of an operator is displayed.

For measuring the execution time of a sequence of operators all `update` options should be switched off in order to reduce the influence of the runtime of GUI updates in HDevelop. For this the operators `dev_update_pc`, `dev_update_time`, `dev_update_var`, and `dev_update_window` or the procedures `dev_update_on` and `dev_update_off` can be used.

This option can also be controlled via the `Preferences` dialog: `Edit` ▷ `Preferences` ▷ `Runtime Settings` ▷ `Show Processing Time`.

Attention

This operator is not supported for code export.

Parameters

▷ **`DisplayMode`** (input_control) . . . . . string ⇝ *string*
Mode for graphic output.
Default: 'off'
List of values: `DisplayMode` ∈ {'on', 'off'}

Result

If the values of the specified parameters are correct, `dev_update_time` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

See also

`dev_update_pc`, `dev_update_window`, `dev_update_var`

Module

Foundation

`dev_update_var ( : : DisplayMode : )`

Switches the update of the variable window during program execution on or off.

`dev_update_var` specifies the behavior of the variable window during program execution. If `DisplayMode` is set to *'on'*, which is the default, the contents of the variable window (iconic and control variables) is updated each time a variable is modified by the program.

If the mode is *'off'* the values of the control variables and the icons of the iconic variables are not updated before the execution stops.

For measuring the execution time of a sequence of operators all `update` options should be switched off in order to reduce the influence of the runtime of GUI updates in HDevelop. For this the operators `dev_update_pc`, `dev_update_time`, `dev_update_var`, and `dev_update_window` or the procedures `dev_update_on` and `dev_update_off` can be used.

This option can also be controlled via the `Preferences` dialog: `Edit` ▷ `Preferences` ▷ `Runtime Settings` ▷ `Update Variables`.

Attention

This operator is not supported for code export.

Parameters

▷ **`DisplayMode`** (input_control) . . . . . string ⇝ *string*
Mode for graphic output.
Default: 'off'
List of values: `DisplayMode` ∈ {'on', 'off'}

Result

If the values of the specified parameters are correct, `dev_update_var` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised and an error code returned.

See also

`dev_update_pc`, `dev_update_window`, `dev_update_time`

Module

Foundation

dev_update_window (: : DisplayMode :)

Switches the automatic output of iconic output objects into the graphics window during program execution on or off.

dev_update_window specifies whether all iconic objects that are returned by an operator call have to be displayed in the active graphics window (DisplayMode = *'on'*—default) or not (DisplayMode = *'off'*).

This option has no influence on the object output in single step mode. After the execution of a single operator the iconic output objects are always displayed in the active graphics window.

This option should be set to *'off'*, if only selected objects should to be displayed in the graphics window. In this case the objects should be displayed by dev_display.

For measuring the execution time of a sequence of operators all update options should be switched off in order to reduce the influence of the runtime of GUI updates in HDevelop. For this the operators dev_update_pc, dev_update_time, dev_update_var, and dev_update_window or the procedures dev_update_on and dev_update_off can be used.

This option can also be controlled via the Preferences dialog: Edit ▷ Preferences ▷ Runtime Settings ▷ Update Window.

Attention

This operator is not supported for code export.

Parameters

▷ **DisplayMode** (input_control) . . . . . string ⇝ *string*
Mode for graphic output.
Default: 'off'
List of values: DisplayMode ∈ {'on', 'off'}

Result

If the values of the specified parameters are correct, dev_update_window returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Possible Successors

dev_display

See also

dev_update_pc, dev_update_var, dev_update_time

Module

Foundation

Chapter 11

File

11.1 Access

close_file (: : FileHandle :)

Closing a text file.

The operator close_file closes a file which was opened via the operator open_file.

Parameters

▷ **FileHandle** (input_control) ..... file(-array) ⇝ *handle*
File handle.

Example

```
open_file ('standard', 'output', FileHandle)
* ....
close_file(FileHandle)
```

Result

If the file handle is correct close_file returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_file

See also

open_file

Module

Foundation

fnew_line (: : FileHandle :)

Write a line break and clear the output buffer.

The operator fnew_line writes a line break into the output file defined by the handle FileHandle. The input file must have been opened with open_file in text format. The call of fnew_line also empties possibly retained data from the output buffer into the file (see fwrite_string and set_system(::'flush_file', <boolean-value>:)).

Which characters are written as line break depends on the operating system: under Windows the sequence '$\backslash r \backslash n$' (carriage return + line feed) is used as the standard line break, under Linux just '$\backslash n$' (line feed).

Parameters

▷ **FileHandle** (input_control) ..... file ⇝ *handle*
File handle.

Example

```
fwrite_string(FileHandle,'Good Morning')
fnew_line(FileHandle)
```

Result

If an output file is open and it can be written to the file, the operator `fnew_line` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fwrite_string`

See also

`fwrite_string`

Module

Foundation

fread_bytes (: : FileHandle, NumberOfBytes : ReadData, IsEOF)

Read bytes from a binary file.

The operator `fread_bytes` reads bytes from the input file defined by `FileHandle`. The input file must have been opened with `open_file` in binary format.

The number of bytes to be read, greater than *0*, is specified as `NumberOfBytes`.

The bytes that are read are returned in `ReadData`. `IsEOF` is set to *1* if end of file is reached while reading the bytes from the input binary file. Otherwise, it is set to *0*.

When the number of bytes to be read is larger than the number of bytes in the input binary file, the operator `fread_bytes` returns all bytes read till the end of the file in `ReadData` and parameter `IsEOF` is set to *1*.

If no byte can be read because the end of the file is reached, `ReadData` is empty and `IsEOF` is set to *1*.

Parameters

▷ **FileHandle** (input_control) ..... file ⇝ *handle*
File handle.

▷ **NumberOfBytes** (input_control) ..... integer ⇝ *integer*
Number of bytes to be read.

▷ **ReadData** (output_control) ..... integer-array ⇝ *integer*
Bytes read from the input binary file.

▷ **IsEOF** (output_control) ..... integer ⇝ *integer*
Indicates if end of file is reached while reading the file.

Example

```
* Read a binary file 5 bytes at a time till EOF is reached.
open_file(Filename,'input_binary',FileHandle)
repeat
  fread_bytes(FileHandle, 5, BytesRead, IsEOF)
until (IsEOF)
close_file (FileHandle)
```

Result

If an input file is open in binary mode and no file read error occurs,the operator `fread_bytes` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_file`

Possible Successors

`close_file`

Alternatives

`fread_char`

See also

`open_file`, `close_file`, `fwrite_bytes`

Module

Foundation

fread_char (: : FileHandle : Char)

Read one character from a text file.

The operator `fread_char` reads a character from the input file defined by `FileHandle`. The input file must have been opened with `open_file` in text format.

The read character or the control character sequence 'eof' is returned in parameter `Char`. The operator `fread_char` respects the encoding of the characters, unless the file was opened with the option *'ignore_encoding'* (see `open_file`). Thus, when opened with the correct encoding the operator `fread_char` returns multi-byte characters at once. If necessary, the character is transcoded into the current encoding of the HALCON library (see `set_system(::'filename_encoding',<encoding>:)`).

When the file was opened with encoding mode *'ignore_encoding'*, `Char` always returns one byte without any interpretation or transcoding. This is useful for reading a file with special control bytes in a kind of raw mode. If no character can be read because the end of the file is reached, `fread_char` returns the control character sequence *'eof'* in `Char`.

The operator `fread_char` emits a low-level error message, when the next byte or byte sequence does not represent a valid code point in the specified encoding. Despite of the low-level error message, the operator will not fail and `Char` will contain the next byte. Furthermore, the operator also emits a low-level error message, when the read character cannot be transcoded without loss of information into the current encoding of the HALCON library. This can only happen when the file is UTF-8 encoded and the current encoding of the HALCON library is *'locale'* (see `set_system(::'filename_encoding','locale':)`).

Parameters

▷ **FileHandle** (input_control) ..... file ⇝ *handle*
File handle.

▷ **Char** (output_control) ..... string ⇝ *string*
Read character, which can be multi-byte or the control string *'eof'*.

Example

```
* Read a text file character by character.
open_file (FileName, 'input', FileHandle)
repeat
    fread_char (FileHandle, Char)
until (Char == 'eof')
close_file (FileHandle)
```

Result

If an input file is open, the operator `fread_char` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised. Encoding errors have no influence on the result state.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_file`

Possible Successors

`close_file`

Alternatives

`fread_string`, `read_string`, `fread_line`

See also

`open_file`, `close_file`, `fread_string`, `fread_line`

Module

Foundation

fread_line (: : FileHandle : OutLine, IsEOF)

Read a character line from a text file.

The operator `fread_line` reads a whole line (including the line break character) from the input file defined by the handle `FileHandle`. The input file must have been opened with `open_file` in text format.

The read line is returned in parameter `OutLine`. It starts at the current file position and ends at the end of the file or with the first line break character found. A subsequent read operation on the file would start after the line break, i.e., at the beginning of the next line. If needed, the output string is transcoded into the current encoding of the HALCON library (the default is UTF-8).

The range of control characters that are handled as line break depends on the file encoding, which can be specified when the file is opened with `open_file`. The standard line break characters are *'\n'* (line feed), *'\r'* (carriage return), and *'\f'* (form feed). These characters are accepted on all encodings or when the encoding has to be ignored. The line break sequence *'\r\n'* (carriage return + line feed), which is the default line break under Windows, is handled (on all systems) as one line break and returned as *'\n'* in the output string. In UTF-8 encoded files, the following Unicode control code points will also terminate the read line: *U+0085* (next line), *U+2028* (line separator), and *U+2029* (paragraph separator).

If the end of the file is reached before any character was written to the output string, the parameter `IsEOF` returns the value *1*, otherwise *0*.

The operator fread_line emits a low-level error message, when it encounters bytes that do not represent a valid code point in the specified encoding. Despite of the low-level error message, the operator will not fail and OutLine will contain potentially invalid bytes (invalid within the specified encoding). Furthermore, the operator also emits a low-level error message, when the output string cannot be transcoded without loss of information into the current encoding of the HALCON library. For a correctly encoded string this can only happen when the file is UTF-8 encoded and the current encoding of the HALCON library is *'locale'* (see set_system('filename_encoding','locale')).

Parameters

- ▷ **FileHandle** (input_control) ..... file ⇝ *handle*
  File handle.
- ▷ **OutLine** (output_control) ..... string ⇝ *string*
  Read line.
- ▷ **IsEOF** (output_control) ..... integer ⇝ *integer*
  Reached end of file before any character was read.

Example

```
do {
  fread_line(FileHandle,&Line,&IsEOF);
} while(IsEOF==0);
```

Result

If the file is open and a suitable line is read, fread_line returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised. Encoding errors have no influence on the result state.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_file

Possible Successors

close_file

Alternatives

fread_char, fread_string

See also

open_file, close_file, fread_char, fread_string

Module

Foundation

fread_string (: : FileHandle : OutString, IsEOF)

Read a string from a text file.

The operator fread_string reads a string from the input file defined by the handle FileHandle. The input file must have been opened with open_file in text format.

A string begins with the first character that is not a separator, i.e., that is no white space or line break: all other characters, mainly letters, numbers, and all other printable characters, but also all other control characters that are no separators are added to the output string. The string ends at the first separator character after one or more accepted characters, or when the end of the file was reached. The terminating separator character is not added to the output string, but the file position indicator remains after it. Thus, a subsequent read operation on the file would start after the terminating separator character. The read character sequence is returned in parameter OutString. If needed, the output string is transcoded into the current encoding of the HALCON library (the default is UTF-8).

The range of characters that are handled as separator depends on the file encoding, which can be specified when the file is opened. The character ' ' (space) and the standard ASCII whitespace control characters '\t' (tab), '\f' (form feed), '\n' (line feed), '\r' (carriage return), and '\v' (vertical tab) are all accepted as separator on all locales or when the encoding has to be ignored. Other encodings may provide additional separator characters, e.g., Latin-1 defines *'0xA0'* (no-break space) or Shift-JIS (as many other Asian encodings) defines *'0x8140'* (ideographic space) as valid separator. UTF-8 handles these characters also as a separator just as all other code points which in the Unicode standard are defined as white spaces in the categories space separators *Zs*, line separators *Zl*, and paragraph separators *Zp*.

If the end of the file is reached before any character was written to the output string, `IsEOF` returns the value *1*, otherwise *0*.

The operator `fread_string` emits a low-level error message, when it encounters bytes that do not represent a valid code point in the specified encoding. Despite of the low-level error message, the operator will not fail and `OutString` will contain potentially invalid bytes (invalid within the specified encoding). Furthermore, the operator also emits a low-level error message, when the output string cannot be transcoded without loss of information into the current encoding of the HALCON library. For a correctly encoded string this can only happen when the file is UTF-8 encoded and the current encoding of the HALCON library is *'locale'* (see `set_system(::'filename_encoding','locale':)`).

Parameters

- ▷ **`FileHandle`** (input_control) ..... file ⇝ *handle*
  File handle.
- ▷ **`OutString`** (output_control) ..... string ⇝ *string*
  Read character sequence.
- ▷ **`IsEOF`** (output_control) ..... integer ⇝ *integer*
  Reached end of file before any character was added to the output string.

Example

```
fwrite_string(FileHandle,'Please enter text and return: ..')
fread_string(FileHandle,String,IsEOF)
fwrite_string(FileHandle,['here it is again: ',String])
fnew_line(FileHandle)
```

Result

If a file is open and a suitable string is read, `fread_string` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception is raised. Encoding errors have no influence on the result state.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_file`

Possible Successors

`close_file`

Alternatives

`fread_char`, `read_string`, `fread_line`

See also

`open_file`, `close_file`, `fread_char`, `fread_line`

Module

Foundation

`fwrite_bytes ( : : FileHandle, DataToWrite : NumberOfBytesWritten )`

Write bytes to a binary file.

The operator fwrite_bytes writes bytes to the output file defined by FileHandle. The output file must have been opened with open_file in binary format.

The data to be written to the file is specified as DataToWrite.

The number of bytes that are written to the file is returned in NumberOfBytesWritten.

Parameters

▷ **FileHandle** (input_control) ..... file ⇝ *handle*
File handle.

▷ **DataToWrite** (input_control) ..... integer-array ⇝ *integer*
Data to be written to the file.

▷ **NumberOfBytesWritten** (output_control) ..... integer ⇝ *integer*
Number of bytes written to the output binary file.

Example

```
* Write a binary file byte by byte.
open_file (Filename, 'append_binary', FileHandle)
fwrite_bytes(FileHandle, [0x97, 99, 102], BytesWritten)
close_file (FileHandle)
```

Result

If an output file is open in binary mode and no file write error occurs, the operator fwrite_bytes returns 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_file

Possible Successors

close_file

Alternatives

fwrite_string

See also

open_file, close_file, fread_bytes

Module

Foundation

fwrite_string (: : FileHandle, String :)

Write strings and numbers into a text file.

The operator fwrite_string writes one or more strings or numbers to the output file defined by the handle FileHandle. The output file must have been opened with open_file in text format.

Strings and numbers that are to be written into the file are passed via the input parameter String to the operator fwrite_string. The parameter String accepts also tuples where strings and numbers are mixed. All elements of the tuple are written consecutively into the file without blanks or other separators in between. Numbers are converted into a string before they are written.

The operator fwrite_string emits a low-level error message, when the string cannot be transcoded without loss of information into the specified file encoding. This can only happen when the file encoding is *'locale_encoding'* and the current encoding of the HALCON library is *'utf8'* (see set_system(::'filename_encoding','utf8':)).

The call set_system(::'flush_file', <boolean-value>:) determines whether the output characters are immediately written to the file or not. If the value *'flush_file'* is set to *'false'*, the characters (especially in case of screen output) show up only after the operator fnew_line is called.

Parameters

▷ **FileHandle** (input_control) ..... file ⇝ *handle*
File handle.

▷ **String** (input_control) ..... string(-array) ⇝ *string* / integer / real
Values to be written into the file.
Default: 'hallo'

Example

```
fwrite_string(FileHandle,['text with numbers:',5,' and ',1.0])
* results in the following output:
* 'text with numbers:5 and 1.00000'
```

Result

If the writing procedure was carried out successfully, the operator fwrite_string returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised. Encoding errors have no influence on the result state.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_file

Possible Successors

close_file

Alternatives

write_string

See also

open_file, close_file, set_system

Module

Foundation

open_file (: : FileName, FileType : FileHandle)

Open a file in text or binary format.

open_file opens a file in text format or in binary format. The name of the file is defined by the parameter FileName. The operator open_file returns a handle to the file in the output parameter FileHandle. The parameter FileType determines the type of the file. For text files this parameter can also be used to specify the encoding used for strings in that file, where besides UTF-8 only the local-8-bit encoding is supported.

The following settings for FileType are supported:

'input': An already existing input file is opened for reading in text format.

'output': A new output file is opened for writing in text format.

'append': An already existing output file is opened for writing at the end of the file in text format.

'input_binary': An already existing input file is opened for reading in binary format.

'output_binary': A new output file is opened for writing in binary format.

'append_binary': An already existing output file is opened for writing at the end of the file in binary format.

For text files the tuple passed to `FileType` can be extended by one of the following encoding settings:

'utf8_encoding': Strings in the file are encoded in UTF-8. This is the default, so for UTF-8 encoded files and all files which use only pure 7-bit US-ASCII characters this value can be omitted.

'locale_encoding': Strings in the file are encoded in the local-8-bit encoding which depends on the system's current locale setting. Valid encodings are defined, e.g., under Windows by the code pages 1252 (a Microsoft dialect of Latin-1) or 932 (Shift-JIS) or on Linux by the locales en_US.utf8, de_DE.iso885915, or ja_JP.sjis.

'ignore_encoding': The encoding of the strings that are read from the file or written into it is not handled. In that mode multi-byte characters are neither processed nor interpreted, the operator `fread_char` returns always one byte, and separator characters that depend on a specific locale are not handled in `fread_line` and `fread_string`. Furthermore, the strings are not transcoded into or from the current encoding of the HALCON library.

Note: Strings are still transcoded in the HALCON C/C++ interface, when the encoding of the HALCON C/C++ interface differs from the encoding of the HALCON library. For HDevelop scripts, the encoding is always set to UTF-8. To avoid transcoding of strings to HDevelop, the encoding of the HALCON library should be set to UTF-8.

For the standard terminal input and output streams, the file names *'standard'* (*'input'* and *'output'*) and *'error'* (only *'output'*) are reserved.

Parameters

▷ **`FileName`** (input_control) ..... filename ⇝ *string*
Name of file to be opened.
Default: 'standard'
Suggested values: `FileName` ∈ {'standard', 'error', '/tmp/dat.dat'}

▷ **`FileType`** (input_control) ..... string(-array) ⇝ *string*
Type of file access and optional the string encoding.
Default: 'output'
List of values: `FileType` ∈ {'input', 'output', 'append', 'input_binary', 'output_binary', 'append_binary', 'utf8_encoding', 'locale_encoding', 'ignore_encoding'}

▷ **`FileHandle`** (output_control) ..... file ⇝ *handle*
File handle.

Example

```
* Creating an output text file with the name '/tmp/log.txt' and writing
* of a string:
   open_file('/tmp/log.txt','output',FileHandle)
   fwrite_string(FileHandle,'these are the first and last lines')
   fnew_line(FileHandle)
   close_file(FileHandle)
```

Result

If the parameters are valid, the operator `open_file` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`fwrite_string`, `fread_char`, `fread_string`, `fread_line`, `fread_serialized_item`, `fwrite_serialized_item`, `fread_bytes`, `fwrite_bytes`, `close_file`

See also

close_file

Module

Foundation

11.2 Images

deserialize_image (: Image : SerializedItemHandle :)

Deserialize a serialized image object.

deserialize_image deserializes an image object, that was serialized by serialize_image (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized image object is defined by the parameter SerializedItemHandle. The deserialized image is stored in the image object defined by the parameter Image.

Parameters

▷ **Image** (output_object) ..... image(-array) ⇝ *object :* byte / direction / cyclic / int1 / complex / int2 / uint2 / vector_field / int4 / int8 / real
Image object.

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator deserialize_image returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_image

Possible Successors

disp_image, threshold, regiongrowing, count_channels, decompose3, class_ndim_norm, gauss_filter, fill_interlace, zoom_image_size, zoom_image_factor, crop_part, write_image, rgb1_to_gray

See also

serialize_image

Module

Foundation

image_to_memory_block (Image : : Format, FillColor : MemoryBlockHandle)

Write an image to a memory block in various graphic formats.

The operator image_to_memory_block saves the input image Image to the memory block MemoryBlockHandle in the format Format. If the domain (region) cannot be saved in the specified Format (this is the case for 'jpeg'), all pixels outside the region receive the color defined by FillColor. For gray value images a value between *0* (black) and *255* (white) must be passed. For RGB color images the RGB values can be passed directly as a hexadecimal value: e.g., *0xffff00* for a yellow background (red=255, green=255, blue=0).

The following formats can be set in Format:

'jpeg': JPEG format (lossy compression), file extension *.jpg This format can only store images with one channel (gray value image) or three channels (RGB image).

Only images with the pixel type byte are supported for this file format.

Together with the format string the quality value determining the compression rate can be provided. Large values (maximum 100) create a large memory block with high image quality. Small values significantly decrease the image quality and the size of the memory block.

Possible values: 'jpeg', 'jpeg 30', 'jpeg 60'.

Attention: Images stored for being processed later should not be stored in this format due to the loss of information during compression.

'png': PNG format (lossless compression), file extension *.png This format can only store images with one channel (gray value image) or three channels (RGB image). The maximal supported image size (width x height) for PNG is $2^{31} - 1$, also in HALCON XL.

Images of type byte and uint2 can be stored in PNG format.

Together with the format string, a compression level between 0 and 9 can be specified, where 0 corresponds to no compression and 9 to the best possible compression. Alternatively, the compression can be selected by using string constants.

Possible values: 'png', 'png 7', 'png none', 'png best', 'png fastest'.

If an image with a reduced domain is written, the region is stored as the alpha channel, where the points within the domain are stored as the maximum gray value of the image type and the points outside the domain are stored as the gray value 0. If an image with a full domain is written, no alpha channel is stored.

Parameters

▷ **`Image`** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / complex / int2 / uint2 / vector_field / int4 / int8 / real

Input image.

▷ **`Format`** (input_control) ..... string ⇝ *string*

Graphic format.
Default: 'jpeg'
Suggested values: `Format` ∈ {'jpeg', 'png'}

▷ **`FillColor`** (input_control) ..... number ⇝ *integer* / real

Gray value for pixels not belonging to the image domain (region).
Default: 0
Suggested values: `FillColor` ∈ {-1, 0, 255, 65280, 16711680}

▷ **`MemoryBlockHandle`** (output_control) ..... memory_block ⇝ *handle*

Memory block handle.

Result

If the parameter values are correct the operator `image_to_memory_block` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`read_image`

Possible Successors

`get_memory_block_ptr`, `write_memory_block`, `serialize_tuple`

See also

`memory_block_to_image`

Module

Foundation

memory_block_to_image (: Image : MemoryBlockHandle :)

Read an image from a memory block with different file formats.

The operator memory_block_to_image reads image data from the indicated memory block MemoryBlockHandle and generates the image Image.

The formats JPEG and PNG can be read. The file formats are recognized by the internal structure of the memory block. In case of colored images an image with three color channels is created, the red channel being stored in the first, the green channel in the second and the blue channel in the third component (channel number).

For the PNG image format binary alpha channels are interpreted as domains. Otherwise, the domain of the generated image object (all pixels of the matrix) is chosen maximal.

Attention

If CMYK or YCCK JPEG memory blocks are read, HALCON assumes that these follow the Adobe Photoshop convention that the CMYK channels are stored inverted, i.e., 0 represents 100% ink coverage, rather than 0% ink as one would expect. The images are converted to RGB images using this convention. If the JPEG memory block does not follow this convention, but stores the CMYK channels in the usual fashion, invert_image must be called after reading the image.

If PNG images that contain an alpha channel are read, the alpha channel is returned as the second or fourth channel of the output image, unless the alpha channel contains exactly two different gray values, in which case a one or three channel image with a reduced domain is returned, in which the points in the domain correspond to the points with the higher gray value in the alpha channel.

Parameters

▷ **Image** (output_object) ..... image ⇝ *object :* byte / direction / cyclic / int1 / complex / int2 / uint2 / vector_field / int4 / int8 / real
Read image.

▷ **MemoryBlockHandle** (input_control) ..... memory_block ⇝ *handle*
Memory block handle.

Result

If the parameters are correct the operator memory_block_to_image returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_memory_block, deserialize_tuple

Possible Successors

disp_image, threshold, regiongrowing, count_channels, decompose3, class_ndim_norm, gauss_filter, fill_interlace, zoom_image_size, zoom_image_factor, crop_part, write_image, rgb1_to_gray

See also

image_to_memory_block

Module

Foundation

read_image (: Image : FileName :)

Read an image with different file formats.

The operator read_image reads the indicated image files from the background storage and generates the image. One or more file names can be passed in FileName. If more than one file name is passed, an image object tuple with the corresponding number of image objects is returned.

HALCON formats (HOBJ and IMA)

For images in the HALCON Iconic Object format (HOBJ), multiple images saved in one file are returned as a tuple of images. If regions or XLDs are contained within the file, an exception is raised. See write_object for more information about the HOBJ format.

All images files written by the operator write_image (format 'ima') have the extension '.ima'. A description file can be available for every image in IMA format (same file name with extension '.exp'). The type of the pixel data (byte, int4, real) can also be taken from the description file. If the description file is not available, the type byte is used by default as well as a default data size ('height' x 'width') that can be queried via get_system. If the data size of the '.ima' file and the default data size are not equal, an exception is raised.

Other formats

Besides the HALCON formats, TIFF, GIF, BMP, JPEG, JPEG-2000, JPEG-XR, PNG, PCX, SUN-Raster, PGM, PPM, PBM, and XWD files can also be read. The gray values of PBM images are set at the values 0 and 255. The file formats are either recognized by the extension (if indicated) or because of the internal structure of the files. If the extension is indicated the image can be found faster. If no extension is indicated, files with an extension are preferred to files without extension. In case of PGM, PPM, and PBM the corresponding extension (e.g., 'pgm') or the general value 'pnm' can be used. In case of TIFF 'tiff' and 'tif' are accepted. In case of JPEG-XR 'jxr', 'wdp', 'wmp', and 'hdp' are accepted. In case of JPEG-2000 only 'jp2' is accepted. In case of colored images an image with three color channels is created, the red channel being stored in the first, the green channel in the second and the blue channel in the third component (channel number).

For the image formats TIFF, PNG, JPEG-XR, and JPEG-2000, binary alpha channels are interpreted as domains. For TIFF files, additionally binary SubIFDs with PhotometricInterpretation = TransparencyMask are interpreted as domains. Otherwise, the domain of the generated image object (= all pixels of the matrix) is chosen maximal.

For TIFF images, multipage TIFF files are returned as image object tuples. SubIFDs in a TIFF file are only read if their dimensions are equal to the dimensions of the main image.

Images of type 'int8' can be read on 64 bit systems only. Furthermore, only the IMA and the TIFF format support this image type.

Image files are searched in the current directory and in the image directory of HALCON (determined by the environment variable). The image directory of HALCON is preset at '.' and '/usr/local/halcon/images' in a Unix-like environment and can be set via the operator set_system. More than one image directory can be indicated. This is done by separating the individual directories by a colon.

Furthermore, the search path can be set via the environment variable HALCONIMAGES (same structure as 'image_dir'). Example:

```
setenv HALCONIMAGES "/usr/images:/usr/local/halcon/images"
```

HALCON also searches images in the subdirectory 'images' (Images for the program examples). The environment variable HALCONROOT is used for the HALCON directory.

Attention

If CMYK or YCCK JPEG/JPEG-XR files are read, HALCON assumes that these files follow the Adobe Photoshop convention that the CMYK channels are stored inverted, i.e., 0 represents 100% ink coverage, rather than 0% ink as one would expect. The images are converted to RGB images using this convention. If the JPEG file does not follow this convention, but stores the CMYK channels in the usual fashion, invert_image must be called after reading the image.

If PNG images that contain an alpha channel are read, the alpha channel is returned as the second or fourth channel of the output image, unless the alpha channel contains exactly two different gray values, in which case a one or three channel image with a reduced domain is returned, in which the points in the domain correspond to the points with the higher gray value in the alpha channel.

Parameters

▷ **Image** (output_object) ..... image(-array) ⇝ *object :* byte / direction / cyclic / int1 / complex / int2 / uint2 / vector_field / int4 / int8 / real

Read image.

▷ **FileName** (input_control) . . . . . filename.read(-array) ⇝ *string*
Name of the image to be read.
Default: 'printer_chip/printer_chip_01'
Suggested values: FileName ∈ {'fabrik', 'fuse', 'mreut', 'multiple_dies_01', 'particle', 'patras', 'printer_chip/printer_chip_01', 'rings_and_nuts', 'tooth_rim'}
File extension: .hobj, .ima, .tif, .tiff, .gif, .bmp, .jpg, .jpeg, .jp2, .jxr, .png, .pcx, .ras, .xwd, .pbm, .pnm, .pgm, .ppm

Example

```
* Reading an image:
  read_image(Image,'mreut')

* Reading 3 images into an image array:
  read_image(Images,['ic0','ic1','ic2'])

* Setting of search path for images on '/mnt/images' and '/home/images':
  set_system('image_dir','/mnt/images:/home/images')
```

Result

If the parameters are correct the operator read_image returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

disp_image, threshold, regiongrowing, count_channels, decompose3, class_ndim_norm, gauss_filter, fill_interlace, zoom_image_size, zoom_image_factor, crop_part, write_image, rgb1_to_gray

Alternatives

read_sequence

See also

set_system, write_image

Module

Foundation

read_image_metadata (: : Format, TagName, FileName : TagValue)

Read metadata from image files.

The operator read_image_metadata reads the metadata of the file FileName and returns the information in TagValue. With TagName you specify, for which single tag or tuple of tags you want to retrieve the information. The parameter Format specifies, in which format the metadata is encoded.

This operators supports for Format the TIFF metadata formats *'tiff'* and *'bigtiff'*. The file extension .tif is expected.

The following tags are readable:

TagName	TagValue **data type**
'tiff_aperture_value'	double
'tiff_copyright'	string
'tiff_date_time'	string
'tiff_exposure_time'	double
'tiff_image_description'	string
'tiff_light_source'	integer
'tiff_make'	string
'tiff_software'	string

Note, the operator returns an error in case the tag specified by TagName has no value set.

You can retrieve all readable tags with the string *'tiff_tags_supported_for_reading'* for TagName as well as retrieve all writable tags with the string *'tiff_tags_supported_for_writing'*. Each of both strings can be used only as single value for TagName.

Parameters

▷ **Format** (input_control) . . . . . string ⇝ *string*
Graphic format.
Default: 'tiff'
Suggested values: Format ∈ {'tiff', 'bigtiff'}

▷ **TagName** (input_control) . . . . . tuple ⇝ *string*
Name of the tag to be written in the image file.
Default: 'tiff_image_description'
Suggested values: TagName ∈ {'tiff_image_description', 'tiff_make', 'tiff_software', 'tiff_date_time', 'tiff_copyright', 'tiff_exposure_time', 'tiff_aperture_value', 'tiff_light_source', 'tiff_tags_supported_for_reading', 'tiff_tags_supported_for_writing'}

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Name of image file.
File extension: .tif, .tiff

▷ **TagValue** (output_control) . . . . . tuple ⇝ *string* / integer / real
Output tag value read from the image file.

Result

If TagValue can be correctly read for the specified input parameters the operator read_image_metadata returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. In this case, an extended error information, as e.g., the causative tag, may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

write_image_metadata

Possible Successors

read_image

Module

Foundation

read_sequence (: Image : HeaderSize, SourceWidth, SourceHeight, StartRow, StartColumn, DestWidth, DestHeight, PixelType, BitOrder, ByteOrder, Pad, Index, FileName :)

Read images.

The operator read_sequence reads unformatted image data, from a file and returns a "suitable" image. The image data must be filled consecutively pixel by pixel and line by line.

Any file headers (with the length HeaderSize bytes) are skipped. The parameters SourceWidth and SourceHeight indicate the size of the filled image. DestWidth and DestHeight indicate the size of the image. In the simplest case these parameters are the same. However, areas can also be read. The upper left corner of the required image area can be determined via StartRow and StartColumn.

The pixel types *'bit'*, *'byte'*, *'short'* (16 bits, unsigned), *'signed_short'* (16 bits, signed), *'long'* (32 bits, signed), *'swapped_long'* (32 bits, with swapped segments), and *'real'* (32 bit floating point numbers) are supported. Furthermore, the operator read_sequence enables the extraction of components of a RBG image, if a triple of three bytes (in the sequence "red", "green", "blue") was filed in the image file. For the red component the pixel type *'r_byte'* must be chosen, and correspondingly for the green and blue components *'g_byte'* or *'b_byte'*, respectively.

'MSBFirst' (most significant bit first) or the inversion thereof (*'LSBFirst'*) can be chosen for the bit order (BitOrder). The byte orders (ByteOrder) *'MSBFirst'* (most significant byte first) or *'LSBFirst'*, respectively, are processed analogously. Finally an alignment (Pad) can be set at the end of the line: *'byte'*, *'short'* or *'long'*. If a whole image sequence is stored in the file a single image (beginning at Index 1) can be chosen via the parameter Index.

Image files are searched in the current directory (determined by the environment variable) and in the image directory of HALCON. The image directory of HALCON is preset at '.' and '/usr/local/halcon/images' in a Unix-like environment and can be set via the operator set_system. More than one image directory can be indicated. This is done by separating the individual directories by a colon.

Furthermore the search path can be set via the environment variable HALCONIMAGES (same structure as *'image_dir'*). Example:

```
setenv HALCONIMAGES "/usr/images:/usr/local/halcon/images"
```

HALCON also searches images in the subdirectory 'images' (Images for the program examples). The environment variable HALCONROOT is used for the HALCON directory.

Attention

If files of pixel type *'real'* are read and the byte order is chosen incorrectly (i.e., differently from the byte order in which the data is stored in the file) program error and even crashes because of floating point exceptions may result.

Parameters

▷ **Image** (output_object) . . . . . image ⇝ *object :* byte / int2 / uint2 / int4
Image read.

▷ **HeaderSize** (input_control) . . . . . integer ⇝ *integer*
Number of bytes for file header.
Default: 0
Value range: 0 ≤ HeaderSize

▷ **SourceWidth** (input_control) . . . . . extent.x ⇝ *integer*
Number of image columns of the filed image.
Default: 512
Value range: 1 ≤ SourceWidth

▷ **SourceHeight** (input_control) . . . . . extent.y ⇝ *integer*
Number of image lines of the filed image.
Default: 512
Value range: 1 ≤ SourceHeight

▷ **StartRow** (input_control) . . . . . point.y ⇝ *integer*
Starting point of image area (line).
Default: 0
Value range: 0 ≤ StartRow
Restriction: StartRow < SourceHeight

▷ **StartColumn** (input_control) . . . . . point.x ⇝ *integer*
Starting point of image area (column).
Default: 0
Value range: 0 ≤ StartColumn
Restriction: StartColumn < SourceWidth

▷ **DestWidth** (input_control) . . . . . extent.x ⇝ *integer*
Number of image columns of output image.
Default: 512
Value range: 1 ≤ DestWidth
Restriction: DestWidth <= SourceWidth

▷ **DestHeight** (input_control) . . . . . extent.y ⇝ *integer*
Number of image lines of output image.
Default: 512
Value range: 1 ≤ DestHeight
Restriction: DestHeight <= SourceHeight

▷ **PixelType** (input_control) . . . . . string ⇝ *string*
Type of pixel values.
Default: 'byte'
List of values: `PixelType` ∈ {'bit', 'byte', 'r_byte', 'g_byte', 'b_byte', 'short', 'signed_short', 'long', 'swapped_long', 'real'}

▷ **BitOrder** (input_control) . . . . . string ⇝ *string*
Sequence of bits within one byte.
Default: 'MSBFirst'
List of values: `BitOrder` ∈ {'MSBFirst', 'LSBFirst'}

▷ **ByteOrder** (input_control) . . . . . string ⇝ *string*
Sequence of bytes within one 'short' unit.
Default: 'MSBFirst'
List of values: `ByteOrder` ∈ {'MSBFirst', 'LSBFirst'}

▷ **Pad** (input_control) . . . . . string ⇝ *string*
Data units within one image line (alignment).
Default: 'byte'
List of values: `Pad` ∈ {'byte', 'short', 'long'}

▷ **Index** (input_control) . . . . . integer ⇝ *integer*
Number of images in the file.
Default: 1
Value range: 1 ≤ Index (lin)

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Name of input file.

Result

If the parameter values are correct the operator `read_sequence` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`disp_image`, `count_channels`, `decompose3`, `write_image`, `rgb1_to_gray`

Alternatives

`read_image`

See also

`read_image`

Module

Foundation

serialize_image (Image : : : SerializedItemHandle)

Serialize an image object.

serialize_image serializes an image object (see fwrite_serialized_item for an introduction of the basic principle of serialization). The image object is defined by the parameter Image. The serialized image object is returned by the handle SerializedItemHandle and can be deserialized by deserialize_image.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / complex / int2 / uint2 / vector_field / int4 / int8 / real

Image object.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*

Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_image returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_image

Module

Foundation

write_image (Image : : Format, FillColor, FileName :)

Write images in graphic formats.

The operator write_image saves the input image Image in the file FileName in the format Format. If the domain (region) cannot be saved in the specified Format (this is the case for 'bmp', 'jpeg', and 'ima'), all pixels outside the region receive the color defined by FillColor. For gray value images a value between 0 (black) and 255 (white) must be passed. For RGB color images the RGB values can be passed directly as a hexadecimal value: e.g., 0xffff00 for a yellow background (red=255, green=255, blue=0).

The following formats are currently supported:

'tiff', 'bigtiff': TIFF format, file extension *.tif

All HALCON pixel types are supported. Image object tuples with up to 65535 image objects and up to 65535 channels can be written. Image objects with channels of mixed pixel types can be written.

Compression is possible with

'deflate [num]':	Adobe deflate compression (lossless)
'jpeg [num]':	JPEG (lossy)
'lzw'	LZW (lossless)
'packbits'	PackBits (lossless)

Here, *'[num]'* denotes an optional specification of a compression parameter. For *'deflate'*, a number between *0* (no compression) and *9* (maximum compression) can be specified. For *'jpeg'*, a number between *0* and *100* can be specified. The semantics are identical to the semantics of 'jpeg' described below. *'jpeg'* can only be used for images of type byte and int1 with up to four channels.

The domain (region) of each image object can be saved in compressed form via *'mask'* (default setting) or as an additional alpha channel via *'alpha'*. The domain is only stored if it does not comprise the full image. *'jpeg'* and

'alpha' cannot be used together because owing to the lossy compression of JPEG the domain of the image cannot be reconstructed correctly. The different options can be accumulated by appending them separated by a space character.

Examples:

'tiff deflate 9':	Adobe deflate compression, the domain is stored as a mask image if necessary
'tiff jpeg 90':	JPEG compression with high quality, the domain is stored as a mask image if necessary
'tiff lzw alpha':	LZW compression, the domain is stored as an alpha channel if necessary
'tiff' or *'tiff none'*:	no compression, the domain is stored as a mask image if necessary

Attention: Note that *'bigtiff'* denotes TIFF files that can be larger than 4 GB, while *'tiff'* denotes TIFF files that are limited to 4 GB. The file size depends on the image size and the selected compression. Therefore, *'bigtiff'* should be selected if there is a possibility that the compressed file will grow to more than 4 GB.

'bmp': Windows-BMP format, file extension *.bmp

Restriction:

- This format can only store images with one channel (gray value image) or three channels (RGB image).
- Only images with the pixel type byte are supported for this file format.

'jpeg': JPEG format (lossy compression), file extension *.jpg

Together with the format string the quality value determining the compression rate can be provided, e.g., 'jpeg 30'.

Restriction:

- This format can only store images with one channel (gray value image) or three channels (RGB image).
- Only images with the pixel type byte are supported for this file format.

Attention: Images stored for being processed later should not be stored in this format due to the loss of information during compression.

'jp2': JPEG-2000 format (lossless and lossy compression), file extension *.jp2

Together with the format string the quality value determining the compression rate can be provided (e.g., *'jp2 40'*). This value corresponds to the ratio of the size of the compressed image and the size of the uncompressed image (in percent). Since lossless JPEG-2000 compression already reduces the file size significantly, only smaller values (typically smaller than 50) influence the file size. If no value is provided for the compression (and only then), the image is compressed without loss.

The image can contain an arbitrary number of channels. Possible types are byte, cyclic, direction, int1, uint2, int2, and int4. In the case of int4 it is only possible to store images with less or equal to 24 bits precision (otherwise an exception is raised). If an image with a reduced domain is written, the region is stored as 1-bit alpha channel.

Restriction: The maximal supported image size (width x height) for JPEG-2000 is $2^{31} - 1$ also in HALCON-XL.

Attention: Note that the JPEG-2000 encoding of an image requires a lot of memory. For large images, it is therefore recommended to use a different format (e.g., *'tiff'*).

'jpegxr': JPEG-XR format (lossless and lossy compression), file extension *.jxr

Together with the format string the quality value determining the compression rate can be provided, e.g., *'jpegxr 30'* (use *'jpegxr 100'* or *'jpegxr'* for lossless encoding). Image object tuples with an arbitrary number of image objects can be written. In the case of int4 and real images, the numeric range is compressed to 24 bit accuracy. Note that this may induce loss regardless of the quality setting. If an image with a reduced domain is written, the region is stored without loss as 1-bit alpha channel. Complex images, vector fields and regular images with two gray value channels are padded with an empty third channel to be compliant with the standard.

Restriction:

- This format cannot be written in parallel.

- Images can have up to 8 channels.
- All HALCON pixel types except int8 are supported.

Attention: Note that you need to have write permission in your current working directory in order to save an image in JPEG-XR format, regardless of the target directory.

'png': PNG format (lossless compression), file extension *.png

Together with the format string, a compression level between 0 and 9 can be specified, where 0 corresponds to no compression and 9 to the best possible compression. Alternatively, the compression can be selected with the following strings: *'best'*, *'fastest'*, and *'none'*. Hence, examples for correct parameters are *'png'*, *'png 7'*, and *'png none'*.

Images of type byte and uint2 can be stored in PNG files. If an image with a reduced domain is written, the region is stored as the alpha channel, where the points within the domain are stored as the maximum gray value of the image type and the points outside the domain are stored as the gray value 0. If an image with a full domain is written, no alpha channel is stored.

Restriction:

- This format can only store images with one channel (gray value image) or three channels (RGB image).
- The maximal supported image size (width x height) for PNG is $2^{31} - 1$ also in HALCON-XL.

'hobj': HALCON Iconic Object (HOBJ), file extension *.hobj

All types of HALCON images are supported. See `write_object` for more information about the HOBJ format.

'ima': HALCON format, file extension *.ima and *.exp

This file format is now legacy and the HOBJ format should be used instead. The data is written in binary form line by line (without header or carriage return). The size of the image and the pixel type are stored in the description file "`FileName`.exp". All HALCON pixel types except *'complex'* and *'vector_field'* can be written.

Restriction: This format can only store images with one channel (gray value image).

Parameters

▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / complex / int2 / uint2 / vector_field / int4 / int8 / real

Input images.

▷ **`Format`** (input_control) ..... string ⇝ *string*

Graphic format.
Default: 'tiff'
Suggested values: `Format` ∈ {'tiff', 'tiff mask', 'tiff alpha', 'tiff deflate 9', 'tiff deflate 9 alpha', 'tiff jpeg 90', 'tiff lzw', 'tiff lzw alpha ', 'tiff packbits', 'bigtiff', 'bigtiff mask', 'bigtiff alpha', 'bigtiff deflate 9', 'bigtiff deflate 9 alpha', 'bigtiff jpeg 90', 'bigtiff lzw', 'bigtiff lzw alpha ', 'bigtiff packbits', 'bmp', 'jpeg', 'jpeg 100', 'jpeg 80', 'jpeg 60', 'jpeg 40', 'jpeg 20', 'jp2', 'jp2 50', 'jp2 40', 'jp2 30', 'jp2 20', 'jpegxr', 'jpegxr 50', 'jpegxr 40', 'jpegxr 30', 'jpegxr 20', 'png', 'png best', 'png fastest', 'png none', 'ima', 'hobj'}

▷ **`FillColor`** (input_control) ..... number ⇝ *integer* / real

Fill gray value for pixels not belonging to the image domain (region).
Default: 0
Suggested values: `FillColor` ∈ {-1, 0, 255, 65280, 16711680}

▷ **`FileName`** (input_control) ..... filename.write(-array) ⇝ *string*

Name of image file.
File extension: `.hobj`, `.ima`, `.tif`, `.tiff`, `.bmp`, `.jpg`, `.jpeg`, `.jp2`, `.jxr`, `.png`

Result

If the parameter values are correct the operator `write_image` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

read_image

Module

Foundation

write_image_metadata (: : Format, TagName, TagValue, FileName :)

Write metadata of image files.

The operator write_image_metadata saves the information in TagValue with the tag TagName into the file FileName. In doing so, TagValue and TagName can be single values or tuples of values. The parameter Format specifies, in which format the metadata is to be written.

This operators supports for Format the TIFF metadata formats *'tiff'* and *'bigtiff'*. A file extension .tif is expected.

A list of the supported tag names for TagName and the according data types for TagValue is given in read_image_metadata.

Parameters

▷ **Format** (input_control) . . . . . string ⇝ *string*
Graphic format.
Default: 'tiff'
Suggested values: Format ∈ {'tiff', 'bigtiff'}

▷ **TagName** (input_control) . . . . . tuple ⇝ *string*
Name of the tag to be written in the image file.
Default: 'tiff_image_description'
Suggested values: TagName ∈ {'tiff_image_description', 'tiff_make', 'tiff_software', 'tiff_date_time', 'tiff_copyright', 'tiff_exposure_time', 'tiff_aperture_value', 'tiff_light_source'}

▷ **TagValue** (input_control) . . . . . tuple ⇝ *string* / integer / real
Value of the tag to be written in the image file.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of image file.
File extension: .tif, .tiff

Result

If TagValue can be correctly written for the specified parameters the operator write_image_metadata returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. In this case, an extended error information, as e.g., the causative tag, may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

write_image

Possible Successors

read_image_metadata

Module

Foundation

11.3 Misc

copy_file (: : SourceFile, DestinationFile :)

Copy a file to a new location.

copy_file copies SourceFile to DestinationFile. Existing files are silently overwritten. The file attributes of SourceFile are kept.

Parameters

▷ **SourceFile** (input_control) ..... filename.read ⇝ *string*
File to be copied.

▷ **DestinationFile** (input_control) ..... filename.write ⇝ *string*
Target location.

Result

copy_file returns the value 2 (H_MSG_TRUE) if the file could be copied. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

delete_file (: : FileName :)

Delete a file.

delete_file deletes the file given by FileName.

Parameters

▷ **FileName** (input_control) ..... filename ⇝ *string*
File to be deleted.

Result

delete_file returns the value 2 (H_MSG_TRUE) if the file exists and could be deleted. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

file_exists (: : FileName : FileExists)

Check whether file exists.

The operator file_exists checks whether the indicated file already exists. If this is the case, the parameter FileExists is set to TRUE, otherwise to FALSE.

Parameters

▷ **FileName** (input_control) . . . . . filename ⇝ *string*
Name of file to be checked.
Default: '/bin/cc'

▷ **FileExists** (output_control) . . . . . integer ⇝ *integer*
Boolean number.

Result

If the parameters are correct the operator `file_exists` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`open_file`

Alternatives

`open_file`

Module

Foundation

get_current_dir (: : : DirName)

Get the current working directory.

`get_current_dir` returns the current working directory in `DirName`.

Parameters

▷ **DirName** (output_control) . . . . . filename.dir ⇝ *string*
Name of current working directory.

Result

`get_current_dir` returns the value `2` (`H_MSG_TRUE`) if the current working directory could be determined. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

list_files (: : Directory, Options : Files)

List all files in a directory.

`list_files` returns all files in the directory given by `Directory` in the parameters `Files`. The current directory can be specified with '' or '.'. The parameter `Options` can be used to specify different processing options by passing a tuple of values.

If `Options` contains *'files'* only the files present in `Directory` are returned. If *'directories'* is passed, only the directories present in `Directory` are returned. On Unix-like systems *'symlinks'* may be passed to list symbolic

links. Directories are marked by a trailing '\' (Windows) or a trailing '/' (Unix-like systems). If files as well as directories should be returned, *['files','directories']* must be passed. If none of *'files'*, *'directories'* or *'symlinks'* is passed, `list_files` returns an empty tuple.

By passing *'recursive'*, it can be specified that the directory tree should be searched recursively by examining all sub-directories. On Unix-like systems, *'follow_links'* can be used to specify that symbolic links to files or directories should be followed. In the default setting, symbolic links are not dereferenced, and hence are not searched if they point to directories, and not returned if they point to files. The option *'follow_links'* can not be used in conjunction with the option *'symlinks'*.

For the recursive search, a maximum search depth can be specified with *'max_depth <d>'* , where *'<d>'* is a number that specifies the maximum depth. Hence, *'max_depth 2'* specifies that `Directory` and all immediate sub-directories should be searched. If symbolic links should be followed it might happen that the search does not terminate if the symbolic links lead to a cycle in the directory structure. Because of this, at most 1000000 files (and directories) are returned in `Files`. By specifying a different number with *'max_files <d>'*, this value can be modified.

Parameters

▷ **`Directory`** (input_control) . . . . . filename.dir ⇝ *string*
Name of directory to be listed.

▷ **`Options`** (input_control) . . . . . string(-array) ⇝ *string*
Processing options.
Default: 'files'
Suggested values: `Options` ∈ {'files', 'directories', 'symlinks', 'recursive', 'follow_links', 'max_depth 5', 'max_files 1000'}

▷ **`Files`** (output_control) . . . . . string-array ⇝ *string*
Found files (and directories).

Result

`list_files` returns the value `2` (`H_MSG_TRUE`) if the directory exists and could be read. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`tuple_regexp_select`

Module

Foundation

`make_dir` (: : DirName :)

Make a directory.

`make_dir` creates the directory given by `DirName`.

Parameters

▷ **`DirName`** (input_control) . . . . . filename.dir ⇝ *string*
Name of directory to be created.

Result

`make_dir` returns the value `2` (`H_MSG_TRUE`) if the directory could be created. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

read_world_file (: : FileName : WorldTransformation)

Read the geo coding from an ARC/INFO world file.

read_world_file reads a geocoding from an ARC/INFO world file with the file name FileName and returns it as a homogeneous 2D transformation matrix in WorldTransformation. To find the file FileName, all directories contained in the HALCON system variable *'image_dir'* (usually this is the content of the environment variable HALCONIMAGES) are searched (see read_image). This transformation matrix can be used to transform XLD contours to the world coordinate system before writing them with write_contour_xld_arc_info. If the matrix WorldTransformation is inverted by calling hom_mat2d_invert, the resulting matrix can be used to transform contours that have been read with read_contour_xld_arc_info to the image coordinate system.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the ARC/INFO world file.

▷ **WorldTransformation** (output_control) ..... hom_mat2d ⇝ *real*
Transformation matrix from image to world coordinates.

Result

If the parameters are correct and the world file could be read, the operator read_world_file returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

hom_mat2d_invert, affine_trans_contour_xld, affine_trans_polygon_xld

See also

write_contour_xld_arc_info, read_contour_xld_arc_info, write_polygon_xld_arc_info, read_polygon_xld_arc_info

Module

Foundation

remove_dir (: : DirName :)

Delete an empty directory.

remove_dir deletes the empty directory given by DirName.

Parameters

▷ **DirName** (input_control) ..... filename.dir ⇝ *string*
Name of directory to be deleted.

Result

remove_dir returns the value 2 (H_MSG_TRUE) if the directory could be deleted. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

set_current_dir (: : DirName :)

Set the current working directory.

set_current_dir sets the current working directory to the directory DirName.

Parameters

▷ **DirName** (input_control) ..... filename.dir ⇝ *string*
Name of current working directory to be set.

Result

set_current_dir returns the value 2 (H_MSG_TRUE) if the current working directory could be set. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

11.4 Object

deserialize_object (: Object : SerializedItemHandle :)

Deserialize a serialized iconic object.

deserialize_object deserializes an iconic object, that was serialized by serialize_object (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized iconic object is defined by the parameter SerializedItemHandle. The deserialized iconic object is stored in the iconic object defined by the parameter Object.

Parameters

▷ **Object** (output_object) ..... object(-array) ⇝ *object*
Iconic object.

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator deserialize_object returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_object

See also

serialize_object

Module

Foundation

read_object (: Object : FileName :)

Read an iconic object.

read_object reads an iconic object from the file FileName. The file must be written in the HALCON Iconic Object format (HOBJ), which is supported by the operators write_object, write_image, and write_region. See write_object for more information about the HOBJ format.

Parameters

▷ **Object** (output_object) ..... object(-array) ⇝ *object*
Iconic object.

▷ **FileName** (input_control) ..... filename ⇝ *string*
Name of file.
File extension: .hobj

Result

If the parameters are valid, the operator read_object returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

write_image, write_object

See also

write_object, write_image, read_image

Module

Foundation

serialize_object (Object : : : SerializedItemHandle)

Serialize an iconic object.

serialize_object serializes an iconic object (see fwrite_serialized_item for an introduction of the basic principle of serialization). The iconic object is a tuple of image objects, region objects, or XLD objects, and is defined by the parameter Object. The serialized iconic object is returned by the handle SerializedItemHandle and can be deserialized by deserialize_object.

Parameters

▷ **Object** (input_object) ..... object(-array) ⇝ *object*
Iconic object.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_object returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_object

See also

deserialize_object

Module

Foundation

write_object (Object : : FileName :)

Write an iconic object.

write_object writes the iconic Object to the file FileName. The iconic object is a (possibly mixed) tuple of images, regions, or XLDs. If no extension is specified in FileName, the extension '.hobj' will be appended. The iconic data is written in the HALCON Iconic Object format described below.

HALCON Iconic Object (HOBJ): HOBJ is a binary file format, which provides the functionality to write and read all kinds of iconic HALCON objects (images, regions, and XLDs). Since data is written with neither compression nor conversion, writing this file format is faster than other supported file formats in most circumstances. Hence, if an application needs to read and write all kinds of iconic HALCON objects as fast as possible and no compression is required, this format should be used. The default file extension for this file format is '.hobj'. For images, all HALCON pixel types can be written. Multi-channel images are supported. The channels can have mixed pixel types but must have the same width and height. The domain of an image and it's creation date are stored in the file as well. An object tuple is written into a single file.

Parameters

▷ **Object** (input_object) ..... object(-array) ⇝ *object*
Iconic object.

▷ **FileName** (input_control) ..... filename ⇝ *string*
Name of file.
File extension: .hobj

Result

If the parameters are valid, the operator write_object returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

read_object, read_image

See also

read_object, write_image, read_image, write_region, read_region

Module

Foundation

11.5 Region

deserialize_region (: Region : SerializedItemHandle :)

Deserialize a serialized region.

deserialize_region deserializes a region, that was serialized by serialize_region (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized region is defined by the parameter SerializedItemHandle. The deserialized region is stored in the region defined by the parameter Region.

Parameters

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Region.

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator deserialize_region returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, fread_serialized_item, receive_serialized_item, serialize_region

Possible Successors

reduce_domain, disp_region

See also

serialize_region

Module

Foundation

read_region (: Region : FileName :)

Read binary images or HALCON regions.

The operator read_region reads regions from a binary file.

The following formats are currently supported:

HALCON Iconic Object (HOBJ): File format for iconic HALCON objects. The HOBJ file must contain *only* regions to be readable by read_region. If other objects (e.g., images or XLDs) are stored in the file, an exception is raised. In this case read_object can be used to read the corresponding HOBJ file. The HOBJ format is the default file format for regions. Therefore, the extension '.hobj' does not have to be specified when reading or writing the file. See write_object for more information about the HOBJ format.

HALCON regions: File format for HALCON regions. This file format supports multiple regions in a single file. The file extension is '.reg'. This format is now legacy and the HOBJ format should be used instead.

Tiff: Binary Tiff images with extension '.tiff' or '.tif'. A tiff image can contain any number of regions. The color white is used as foreground.

PNG: Binary PNG images with extension '.png'. The result is always *one* region. The color white is used as foreground.

BMP: Binary Windows bitmap images with extension '.bmp'. The result is always *one* region. The color white is used as foreground.

A search path (`'image_dir'`) can be defined analogous to the operator `read_image`.

Attention

The clipping based on the current image format is set via the operator `set_system('clip_region',<'true'/'false'>)`. Consequently, if no image of sufficient size has been created before the call to `read_region`, `set_system('clip_region','false')` should be called before calling `read_region` to ensure that the region is not being clipped.

Parameters

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Read region.

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the region to be read.
File extension: `.hobj`, `.reg`, `.tif`, `.tiff`, `.png`, `.bmp`

Example

```
* Reading of regions and giving them gray values.
  read_image(Img,'ima_test')
  read_region(Regs,'reg_test')
  reduce_domain(Img,Regs,Res)
```

Result

If the parameter values are correct the operator `read_region` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_image`

Possible Successors

`reduce_domain`, `disp_region`

See also

`write_region`, `read_image`, `write_object`

Module

Foundation

serialize_region (Region : : : SerializedItemHandle)

Serialize a region.

`serialize_region` serializes a region (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The region is defined by the parameter `Region`. The serialized region is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_region`.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Region.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_region` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, read_region, threshold, regiongrowing

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_region

See also

deserialize_region

Module

Foundation

write_region (Region : : FileName :)

Write regions to a file.

The operator write_region writes the regions in Region into a binary file. The regions can be written in the HALCON Iconic Object format (HOBJ), as TIFF files or as HALCON region files. The format is selected via the file name extension of FileName. If no extension is specified or if the file extension is '.hobj', the regions are stored in the HOBJ format. If no extension is specified in FileName, the extension '.hobj' will be appended. See write_object for more information about the HOBJ format. If the extension '.tif' or '.tiff' is specified, the regions are stored as TIFF files. If the extension '.reg' is specified, the regions are stored as a HALCON region file. The HALCON region file format is now legacy and the file format HOBJ should be used instead unless the files must be readable with HALCON versions older than 12.0. Iconic object tuples with an arbitrary number of regions can be written. The output data can be read via the operator read_region.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Region of the images which are returned.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Name of region file.
Default: 'region.hobj'
File extension: .hobj, .reg, .tif, .tiff

Example

```
regiongrowing(Img,Segments,3,3,5,10)
write_region(Segments,'result1')
```

Result

If the parameter values are correct the operator write_region returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, read_region, read_object, threshold, regiongrowing

See also

read_region

Module

Foundation

11.6 Tuple

deserialize_handle (: : SerializedItem : Handle)

Deserialize a serialized item.

deserialize_handle deserializes the content of SerializedItem and returns the deserialized item in Handle (see fwrite_serialized_item for an introduction of the basic principle of serialization).

The serialized item must have been created by serialize_handle, or by the type specific serialization operators, such as serialize_matrix or serialize_shape_model.

Parameters

▷ **SerializedItem** (input_control) ..... serialized_item ⇝ *handle*
Handle containing the serialized item to be deserialized.

▷ **Handle** (output_control) ..... handle ⇝ *handle*
Handle containing the deserialized item.

Example

```
create_matrix (3, 3, 0, MatrixID)
serialize_handle (MatrixID, SerializedMatrix)
deserialize_handle (SerializedMatrix, MatrixID2)
```

Result

If the parameters are valid, the operator deserialize_handle returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

serialize_handle, fread_serialized_item, receive_serialized_item

Module

Foundation

deserialize_tuple (: : SerializedItemHandle : Tuple)

Deserialize a serialized tuple.

deserialize_tuple deserializes the data of a tuple, that was serialized by serialize_tuple (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized data of the tuple is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created tuple with the handle Tuple.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **Tuple** (output_control) ..... tuple(-array) ⇝ *real* / integer / string / handle
Tuple.

Result

If the parameters are valid, the operator deserialize_tuple returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_tuple

Module

Foundation

read_tuple (: : FileName : Tuple)

Read a tuple from a file.

The operator read_tuple reads the contents of FileName and converts it into Tuple. The default HALCON file extension for the tuple is 'tup'. The file has to be generated by write_tuple.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the file to be read.
File extension: .tup

▷ **Tuple** (output_control) ..... tuple(-array) ⇝ *real* / integer / string
Tuple with any kind of data.

Result

If the parameters are correct the operator read_tuple returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

fwrite_string

See also

write_tuple, write_image, write_region, open_file

Module

Foundation

serialize_handle (: : Handle : SerializedItem)

Serialize the content of a handle.

serialize_handle serializes the content of Handle and returns the serialized item in SerializedItem (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized item can later be deserialized with deserialize_handle.

Note that not all handle types support serialization. To check if a handle can be serializes, use tuple_is_serializable or tuple_is_serializable_elem. Handles that have already been cleared cannot be serialized.

Also note that the serialized item created by this operator is compatible with the type specific deserialization operators, such as deserialize_matrix or deserialize_shape_model.

Parameters

▷ **Handle** (input_control) . . . . . handle ⇝ *handle*
Handle that should be serialized.

▷ **SerializedItem** (output_control) . . . . . serialized_item ⇝ *handle*
Handle containing the serialized item.

Example

```
create_matrix (3, 3, 0, MatrixID)
serialize_handle (MatrixID, SerializedMatrix)
deserialize_handle (SerializedMatrix, MatrixID2)
```

Result

If the parameters are valid, the operator `serialize_handle` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`tuple_is_serializable`, `tuple_is_serializable_elem`

Possible Successors

`deserialize_handle`, `fwrite_serialized_item`, `send_serialized_item`

Module

Foundation

serialize_tuple (: : Tuple : SerializedItemHandle)

Serialize a tuple.

`serialize_tuple` serializes the data of a tuple (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The same data that is written in a file by `write_tuple` is converted to a serialized item. The tuple is defined by the handle `Tuple`. The serialized data of a tuple is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_tuple`.

Note that not all handle types can be serialized. If `Tuple` contains a handle that can not be serialized or that has been freed already, an exception is raised. The operators `tuple_is_serializable` and `tuple_is_serializable_elem` can be used to find out if a tuple or its elements can be serialized.

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string / handle
Tuple.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_tuple` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_tuple

Module

Foundation

tuple_is_serializable (: : Tuple : IsSerializable)

Test if a tuple is serializable.

tuple_is_serializable checks if Tuple can be serialized with serialize_tuple. If yes, *1* is returned in IsSerializable. Otherwise, *0* is returned.

A tuple can be serialized if it contains only integers, strings, reals, and serializable handles. Handles that are already freed cannot be serialized. To check all elements of a tuple individually, use tuple_is_serializable_elem.

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *handle* / integer / real / string
Tuple to check for serializability.

▷ **IsSerializable** (output_control) . . . . . number ⇝ *integer*
Boolean value indicating if the input can be serialized.

Result

If the parameters are valid, the operator tuple_is_serializable returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

serialize_tuple, write_tuple

Alternatives

tuple_is_serializable_elem

See also

tuple_is_serializable_elem, serialize_tuple, serialize_handle, deserialize_tuple, deserialize_handle, write_tuple

Module

Foundation

tuple_is_serializable_elem (: : Tuple : IsSerializableElem)

Test if the elements of a tuple are serializable.

tuple_is_serializable checks if the elements of Tuple can be serialized with serialize_tuple. The output IsSerializableElem has the same length as the input Tuple and contains at each position either *0* if the corresponding entry in Tuple cannot be serialized, or *1* if it can.

A tuple element can be serialized if it is an integer, a string, a real value or a serializable handle. Handles that are already freed cannot be serialized. To check the complete tuple for serializability, use tuple_is_serializable.

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *handle* / integer / real / string
Tuple to check for serializability.

▷ **IsSerializableElem** (output_control) . . . . . number(-array) ⇝ *integer*
Boolean value indicating if the input elements can be serialized.

Example

```
* Serialize all serializable elements of a tuple
tuple_is_serializable_elem (Tuple, IsSerializableElem)
if (sum(IsSerializableElem[=]0)>0)
  Tuple[find(IsSerializableElem,0)] := HNULL
endif
serialize_tuple (Tuple, SerializedItem)
```

Result

If the parameters are valid, the operator `tuple_is_serializable` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`serialize_tuple`, `write_tuple`

Alternatives

`tuple_is_serializable`

See also

`tuple_is_serializable`, `serialize_tuple`, `serialize_handle`, `deserialize_tuple`, `deserialize_handle`, `write_tuple`

Module

Foundation

`write_tuple ( : : Tuple, FileName : )`

Write a tuple to a file.

The operator `write_tuple` writes the contents of `Tuple` to a file. The data is written in an ASCII format. Therefore, the file can be exchanged between different architectures (see also Tuple / String Operations). The default HALCON file extension for the tuple is 'tup'.

Exception: Handles

Note that `write_tuple` does not support handles, since binary data is not supported by the underlying ASCII format. Any handle contained in `Tuple` is replaced by the integer *0*. To write a tuple that contains handles, use `serialize_tuple` and `fwrite_serialized_item`.

Parameters

▷ **`Tuple`** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string
Tuple with any kind of data.

▷ **`FileName`** (input_control) . . . . . filename.write ⇝ *string*
Name of the file to be written.
File extension: `.tup`

Result

If the parameters are correct the operator `write_tuple` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Alternatives

fwrite_string

See also

read_tuple, write_image, write_region, open_file

Module

Foundation

11.7 XLD

deserialize_xld (: XLD : SerializedItemHandle :)

Deserialize a serialized XLD object.

deserialize_xld deserializes an XLD object, that was serialized by serialize_xld (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized XLD object is defined by the parameter SerializedItemHandle. The deserialized XLD object is stored in the XLD object defined by the parameter XLD.

Parameters

- ▷ **XLD** (output_object) ..... xld(-array) ⇝ *object*
  XLD object.
- ▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
  Handle of the serialized item.

Result

If the parameters are valid, the operator deserialize_xld returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, fread_serialized_item, receive_serialized_item, serialize_xld

Possible Successors

clip_contours_xld, split_contours_xld

See also

serialize_xld

Module

Foundation

read_contour_xld_arc_info (: Contours : FileName :)

Read XLD contours to a file in ARC/INFO generate format.

read_contour_xld_arc_info reads the lines stored in ARC/INFO generate format in the file FileName, and returns them as XLD contours in Contours. To find the file FileName, all directories contained in the HALCON system variable *'image_dir'* (usually this is the content of the environment variable HALCONIMAGES) are searched (see read_image). The returned contours are in world coordinates. They can be transformed to the image coordinate system with the operator affine_trans_contour_xld. The necessary transformation matrix can be generated by using read_world_file to read the transformation matrix from image to world coordinates, and inverting this matrix by calling hom_mat2d_invert.

Parameters

▷ **Contours** (output_object) ..... xld_cont(-array) ⇝ *object*
Read XLD contours.

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the ARC/INFO file.

Example

```
* Read the transformation and invert it
read_world_file ('image.tfw', WorldTransformation)
hom_mat2d_invert (WorldTransformation, ImageTransformation)
* Read the image
read_image (Image, 'image.tif')
* Read the line data
read_contour_xld_arc_info (LinesWorld, 'lines.gen')
* Transform the line data to image coordinates
affine_trans_contour_xld (LinesWorld, Lines, ImageTransformation)
```

Result

If the parameters are correct and the file could be read, the operator `read_contour_xld_arc_info` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`hom_mat2d_invert`, `affine_trans_contour_xld`

See also

`read_world_file`, `write_contour_xld_arc_info`, `read_polygon_xld_arc_info`

Module

Foundation

read_contour_xld_dxf (: Contours : FileName, GenParamName, GenParamValue : DxfStatus)

Read XLD contours from a DXF file.

`read_contour_xld_dxf` reads the contents of the DXF file `FileName` (DXF version AC1009, AutoCAD Release 12) and converts them to the XLD contours `Contours`. If no absolute path is given in `FileName` the DXF file is searched in the current directory of the HALCON process.

The output parameter `DxfStatus` contains information about the number of contours that were read and, if necessary, warnings that parts of the DXF file could not be interpreted.

The operator `read_contour_xld_dxf` supports the following DXF entities:

- POLYLINE
  - 2D curves made up of line segments
  - Closed 2D curves made up of line segments
- LWPOLYLINE
- LINE
- POINT
- CIRCLE

- ARC
- ELLIPSE
- SPLINE
- BLOCK
- INSERT

The x and y coordinates of the DXF entities are stored in the column and row coordinates, respectively, of the XLD contours `Contours`. The z coordinates of the DXF entities are ignored.

If the file has been created with the operator `write_contour_xld_dxf`, all attributes and global attributes that were originally defined for the XLD contours are read. This means that `read_contour_xld_dxf` supports all the extended data written by the operator `write_contour_xld_dxf`. The reading of these attributes can be switched off by setting the generic parameter *'read_attributes'* to *'false'*. Generic parameters are set by specifying the parameter name(s) in `GenParamName` and the corresponding value(s) in `GenParamValue`.

DXF entities of the type CIRCLE, ARC, ELLIPSE, and SPLINE are approximated by XLD contours. The accuracy of this approximation can be controlled with the two generic parameters *'min_num_points'* and *'max_approx_error'*. The parameter *'min_num_points'* defines the minimum number of sampling points that are used for the approximation. Note that the parameter *'min_num_points'* always refers to the full circle or ellipse, respectively, even for ARCs or elliptical arcs, i.e., if *'min_num_points'* is set to *50* and a DXF entity of the type ARC is read that represents a semi-circle, this semi-circle is approximated by at least 25 sampling points. The parameter *'max_approx_error'* defines the maximum deviation of the XLD contour from the ideal circle or ellipse, respectively (unit: pixel). For the determination of the accuracy of the approximation both criteria are evaluated. Then, the criterion that leads to the more accurate approximation is used.

Internally, the following default values are used for the generic parameters:

- *'read_attributes' = 'true'*
- *'min_num_points' = 20*
- *'max_approx_error' = 0.25*

To achieve a more accurate approximation, either the value for *'min_num_points'* must be increased or the value for *'max_approx_error'* must be decreased.

Parameters

▷ **`Contours`** (output_object) ..... xld_cont(-array) ⇝ *object*
Read XLD contours.

▷ **`FileName`** (input_control) ..... filename.read ⇝ *string*
Name of the DXF file.
File extension: `.dxf`

▷ **`GenParamName`** (input_control) ..... attribute.name(-array) ⇝ *string*
Names of the generic parameters that can be adjusted for the DXF input.
Default: []
List of values: `GenParamName` ∈ {'read_attributes', 'min_num_points', 'max_approx_error'}

▷ **`GenParamValue`** (input_control) ..... attribute.value(-array) ⇝ *real* / integer / string
Values of the generic parameters that can be adjusted for the DXF input.
Default: []
Suggested values: `GenParamValue` ∈ {'true', 'false', 0.1, 0.25, 0.5, 1, 2, 5, 10, 20}

▷ **`DxfStatus`** (output_control) ..... string(-array) ⇝ *string*
Status information.

Result

If the parameters are correct and the file could be read the operator `read_contour_xld_dxf` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

write_contour_xld_dxf

See also

write_contour_xld_dxf, read_polygon_xld_dxf, query_contour_attribs_xld, query_contour_global_attribs_xld, get_contour_attrib_xld, get_contour_global_attrib_xld

Module

Foundation

read_polygon_xld_arc_info (: Polygons : FileName :)

Read XLD polygons from a file in ARC/INFO generate format.

read_polygon_xld_arc_info reads the lines stored in ARC/INFO generate format in the file FileName, and returns them as XLD polygons in Polygons. To find the file FileName, all directories contained in the HALCON system variable *'image_dir'* (usually this is the content of the environment variable HALCONIMAGES) are searched (see read_image). The returned polygons are in world coordinates. They can be transformed to the image coordinate system with the operator affine_trans_polygon_xld. The necessary transformation matrix can be generated by using read_world_file to read the transformation matrix from image to world coordinates, and inverting this matrix by calling hom_mat2d_invert.

Parameters

▷ **Polygons** (output_object) ..... xld_poly(-array) ⇝ *object*
Read XLD polygons.

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the ARC/INFO file.

Example

```
* Read the transformation and invert it
read_world_file ('image.tfw', WorldTransformation)
hom_mat2d_invert (WorldTransformation, ImageTransformation)
* Read the image
read_image (Image, 'image.tif')
* Read the line data
read_polygon_xld_arc_info (LinesWorld, 'lines.gen')
* Transform the line data to image coordinates
affine_trans_polygon_xld (LinesWorld, Lines, ImageTransformation)
```

Result

If the parameters are correct and the file could be read, the operator read_polygon_xld_arc_info returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

hom_mat2d_invert, affine_trans_polygon_xld

See also

read_world_file, write_polygon_xld_arc_info, read_contour_xld_arc_info

Module

Foundation

read_polygon_xld_dxf (: Polygons : FileName, GenParamName, GenParamValue : DxfStatus)

Read XLD polygons from a DXF file.

read_polygon_xld_dxf reads the contents of the DXF file FileName (DXF version AC1009, AutoCAD Release 12) and converts them to the XLD polygons Polygons. If no absolute path is given in FileName the DXF file is searched in the current directory of the HALCON process.

The output parameter DxfStatus contains information about the number of polygons that were read and, if necessary, warnings that parts of the DXF file could not be interpreted.

The operator read_polygon_xld_dxf supports the following DXF entities:

- POLYLINE
  - 2D curves made up of line segments
  - Closed 2D curves made up of line segments
- LWPOLYLINE
- LINE
- POINT
- CIRCLE
- ARC
- ELLIPSE
- SPLINE
- BLOCK
- INSERT

The x and y coordinates of the DXF entities are stored in the column and row coordinates, respectively, of the XLD polygons Polygons. The z coordinates of the DXF entities are ignored.

DXF entities of the type CIRCLE, ARC, ELLIPSE, and SPLINE are approximated by XLD polygons. The accuracy of this approximation can be controlled with the two generic parameters *'min_num_points'* and *'max_approx_error'* (for SPLINE only *'max_approx_error'*). Generic parameters are set by specifying the parameter name(s) in GenParamName and the corresponding value(s) in GenParamValue. The parameter *'min_num_points'* defines the minimum number of sampling points that are used for the approximation. Note that the parameter *'min_num_points'* always refers to the full circle or ellipse, respectively, even for ARCs or elliptical arcs, i.e., if *'min_num_points'* is set to *50* and a DXF entity of the type ARC is read that represents a semi-circle, this semi-circle is approximated by at least 25 sampling points. The parameter *'max_approx_error'* defines the maximum deviation of the XLD polygon from the ideal circle or ellipse, respectively (unit: pixel). For the determination of the accuracy of the approximation both criteria are evaluated. Then, the criterion that leads to the more accurate approximation is used.

Internally, the following default values are used for the generic parameters:

- *'min_num_points'* = 20
- *'max_approx_error'* = 0.25

To achieve a more accurate approximation, either the value for *'min_num_points'* must be increased or the value for *'max_approx_error'* must be decreased.

Note that reading a DXF file with read_polygon_xld_dxf results in exactly the same geometric information as reading the file with read_contour_xld_dxf. However, the resulting data structure is different.

Parameters

▷ **Polygons** (output_object) ..... xld_poly(-array) ⇝ *object*
Read XLD polygons.

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the DXF file.
File extension: .dxf

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that can be adjusted for the DXF input.
Default: []
List of values: GenParamName ∈ {'min_num_points', 'max_approx_error'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
Values of the generic parameters that can be adjusted for the DXF input.
Default: []
Suggested values: GenParamValue ∈ {0.1, 0.25, 0.5, 1, 2, 5, 10, 20}

▷ **DxfStatus** (output_control) . . . . . string(-array) ⇝ *string*
Status information.

Result

If the parameters are correct and the file could be read the operator read_polygon_xld_dxf returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

write_polygon_xld_dxf

See also

write_polygon_xld_dxf, read_contour_xld_dxf

Module

Foundation

serialize_xld (XLD : : : SerializedItemHandle)

Serialize an XLD object.

serialize_xld serializes an XLD object (see fwrite_serialized_item for an introduction of the basic principle of serialization). The XLD object is defined by the parameter XLD. The serialized XLD object is returned by the handle SerializedItemHandle and can be deserialized by deserialize_xld.

Parameters

▷ **XLD** (input_object) . . . . . xld(-array) ⇝ *object*
XLD object.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_xld returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contour_polygon_xld, gen_parallels_xld, gen_polygons_xld

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_xld

See also

deserialize_xld

Module

Foundation

write_contour_xld_arc_info (Contours : : FileName :)

Write XLD contours to a file in ARC/INFO generate format.

write_contour_xld_arc_info writes the XLD contours Contours to an ARC/INFO generate format file with name FileName. If no absolute path is given in FileName, the output file is created in the current directory of the HALCON process. The contours must have been transformed to the world coordinate system with affine_trans_contour_xld beforehand. The necessary transformation can be read from an ARC/INFO world file with read_world_file.

Parameters

- ▷ **Contours** (input_object) ..... xld_cont(-array) ⇝ *object*
  XLD contours to be written.
- ▷ **FileName** (input_control) ..... filename.write ⇝ *string*
  Name of the ARC/INFO file.

Example

```
* Read transformation and image
read_world_file ('image.tfw', WorldTransformation)
read_image (Image, 'image.tif')
* Segment image
* ...
* Write result
affine_trans_contour_xld (Contours, ContoursWorld, WorldTransformation)
write_contour_xld_arc_info (ContoursWorld, 'result.gen')
```

Result

If the parameters are correct and the file could be written, the operator write_contour_xld_arc_info returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

affine_trans_contour_xld

See also

read_world_file, read_contour_xld_arc_info, write_polygon_xld_arc_info

Module

Foundation

write_contour_xld_dxf (Contours : : FileName :)

Write XLD contours to a file in DXF format.

write_contour_xld_dxf writes the XLD contours Contours to the file FileName in DXF format. If no absolute path is given in FileName the output file is created in the current directory of the HALCON process.

Besides the geometry of the Contours, all attributes and global attributes that are defined for the Contours are written to the file.

write_contour_xld_dxf writes the file according to the DXF version AC1009 (AutoCAD Release 12). Each contour is stored as a POLYLINE. The attribute values are stored as extended data of each VERTEX of the POLYLINE. The global attribute values are stored as extended data of the POLYLINE. All attribute names are also stored as extended data of the POLYLINE.

The operator read_contour_xld_dxf can be used to read the XLD contours together with their attributes.

Other applications that are able to read DXF files only import the contour geometry, but they ignore the attribute information.

Description of the format of the extended data

Each block of extended data starts with the following DXF group:

1001 HALCON

The attributes are written in the following format as extended data of each VERTEX:

DXF	Explanation
1000	Meaning
contour attributes	
1002	Beginning of the value list
{	
1070	Number of attributes (here: 3)
3	
1040	Value of the first attribute
5.00434303	
1040	Value of the second attribute
126.8638916	
1040	Value of the third attribute
4.99164152	
1002	End of the value list
}	

The global attributes are written in the following format as extended data of each POLYLINE:

DXF	Explanation
1000	Meaning
global contour attributes	
1002	Beginning of the value list
{	
1070	Number of global attributes (here: 5)
5	
1040	Value of the first global attribute
0.77951831	
1040	Value of the second global attribute
0.62637949	
1040	Value of the third global attribute
103.94314575	
1040	Value of the fourth global attribute
0.21434096	
1040	Value of the fifth global attribute
0.21921949	
1002	End of the value list
}	

The names of the attributes are written in the following format as extended data of each POLYLINE:

DXF	Explanation
1000	Meaning
names of contour attributes	
1002	Beginning of the value list
{	
1070	Number of attribute names (here: 3)
3	
1000	Name of the first attribute
angle	
1000	Name of the second attribute
response	
1000	Name of the third attribute
edge_direction	
1002	End of the value list
}	

The names of the global attributes are written in the following format as extended data of each POLYLINE:

DXF	Explanation
1000	Meaning
names of global contour attributes	
1002	Beginning of the value list
{	
1070	Number of global attribute names (here: 5)
5	
1000	Name of the first global attribute
regr_norm_row	
1000	Name of the second global attribute
regr_norm_col	
1000	Name of the third global attribute
regr_dist	
1000	Name of the fourth global attribute
regr_mean_dist	
1000	Name of the fifth global attribute
regr_dev_dist	
1002	End of the value list
}	

Parameters

▷ **Contours** (input_object) . . . . . xld_cont(-array) ⇝ *object*
XLD contours to be written.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the DXF file.
File extension: `.dxf`

Result

If the parameters are correct and the file could be written the operator `write_contour_xld_dxf` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

edges_sub_pix

See also

read_contour_xld_dxf, write_polygon_xld_dxf, query_contour_attribs_xld, query_contour_global_attribs_xld, get_contour_attrib_xld, get_contour_global_attrib_xld

Module

Foundation

write_polygon_xld_arc_info (Polygons : : FileName :)

Write XLD polygons to a file in ARC/INFO generate format.

write_polygon_xld_arc_info writes the XLD polygons Polygons to an ARC/INFO generate format file with name FileName. If no absolute path is given in FileName, the output file is created in the current directory of the HALCON process. The polygons must have been transformed to the world coordinate system with affine_trans_polygon_xld beforehand. The necessary transformation can be read from an ARC/INFO world file with read_world_file.

Attention

The XLD contours that are possibly referenced by Polygons are not stored in the ARC/INFO file, since this is not possible with the ARC/INFO generate file format. Therefore, when the polygons are read again using read_polygon_xld_arc_info, this information is lost, and no references to contours are generated for the polygons. Hence, operators that access the contours associated with a polygon, e.g., split_contours_xld will not work correctly.

Parameters

▷ **Polygons** (input_object) ..... xld_poly(-array) ⇝ *object*
XLD polygons to be written.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Name of the ARC/INFO file.

Example

```
* Read transformation and image
read_world_file ('image.tfw', WorldTransformation)
read_image (Image, 'image.tif')
* Segment image
* ...
* Write result
affine_trans_polygon_xld (Polygons, PolygonsWorld, WorldTransformation)
write_polygon_xld_arc_info (PolygonsWorld, 'result.gen')
```

Result

If the parameters are correct and the file could be written, the operator write_polygon_xld_arc_info returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

affine_trans_polygon_xld

See also

read_world_file, read_polygon_xld_arc_info, write_contour_xld_arc_info

Module

Foundation

write_polygon_xld_dxf (Polygons : : FileName :)

Write XLD polygons to a file in DXF format.

write_polygon_xld_dxf writes the XLD polygons Polygons to the file FileName in DXF format. If no absolute path is given in FileName the output file is created in the current directory of the HALCON process.

write_polygon_xld_dxf writes the file according to the DXF version AC1009 (AutoCAD Release 12). Each polygon is stored as a POLYLINE.

The operator read_polygon_xld_dxf can be used to read the XLD polygon.

Attention

The XLD contours that are possibly referenced by Polygons are not stored in the DXF file. Therefore, when the polygons are read again using read_polygon_xld_dxf, this information is lost, and no references to contours are generated for the polygons. Hence, operators that access the contours associated with a polygon, e.g., split_contours_xld will not work correctly.

Parameters

▷ **Polygons** (input_object) ..... xld_poly(-array) ⇝ *object*
XLD polygons to be written.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Name of the DXF file.
File extension: .dxf

Result

If the parameters are correct and the file could be written the operator write_polygon_xld_dxf returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_polygons_xld

See also

read_polygon_xld_dxf, write_contour_xld_dxf

Module

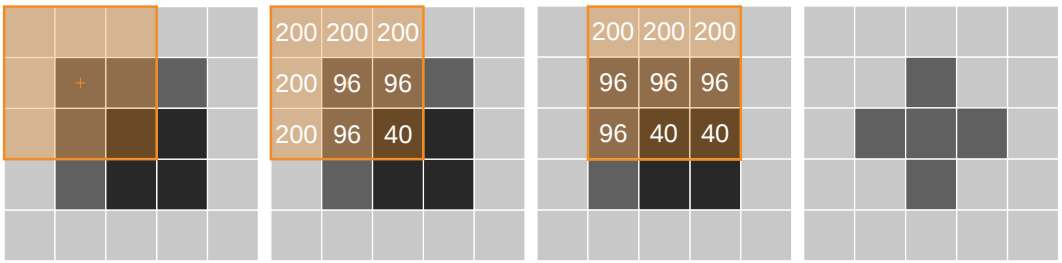
Foundation

Chapter 12

Filters

Filters are an important part of nearly every machine vision application. They can, for instance, be used to smooth images (e.g., `mean_image`), extract sub-pixel precise edges (e.g., `edges_sub_pix`) or process the frequency domain of images (e.g., `fft_image`).

Usually, filter operations are applied using filter masks, which are moved across the input image pixelwise. For each pixel a new value is calculated based on its neighborhood. The neighborhood is determined by shape and size of the filter mask (see the scheme below).



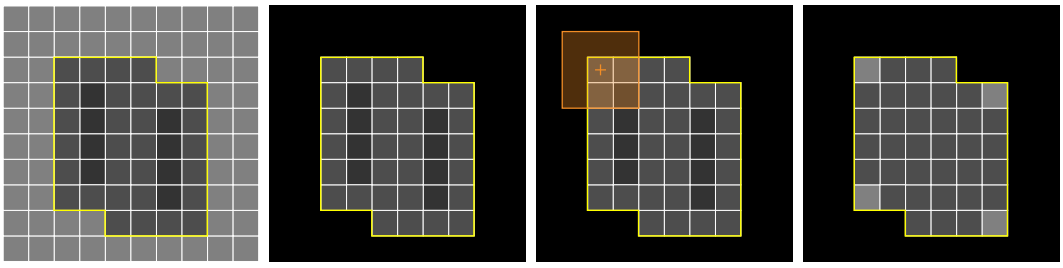
(1) (2) (3) (4)

(1) Image with a filter mask of size 3x3 pixels (orange). (2) For the computation of the new pixel value of the pixel located at the center point of the mask all the pixel values within the mask are taken into account. (3) The mask is moved across the image pixel by pixel. (4) In this case a median filter (see `median_rect`) is used for the computation of the resulting image.

You can tell from the working principle of filter masks, the treatment of pixels along the image or domain border requires special attention, as part of the filter mask exceeds the domain. In the following sections some consequential issues will be discussed.

Applying Filter Masks on Reduced Image Domains

If a filter that is using a mask is applied on an image with a reduced domain, the result along the domain boundary might be surprising because gray values lying outside the boundary are used as input for the filter process (see scheme below).



(1) (2) (3) (4)

(1) Image with a region (yellow) to which the image domain shall be reduced to. (2) The image domain is reduced to the chosen region using reduce_domain. (3) For the computation of a new value for the pixel in the center of the mask all the pixel values within the mask are taken into account. For pixels along the domain border this also includes pixels from outside the domain. (4) Result image after the application of a median filter (see median_rect).

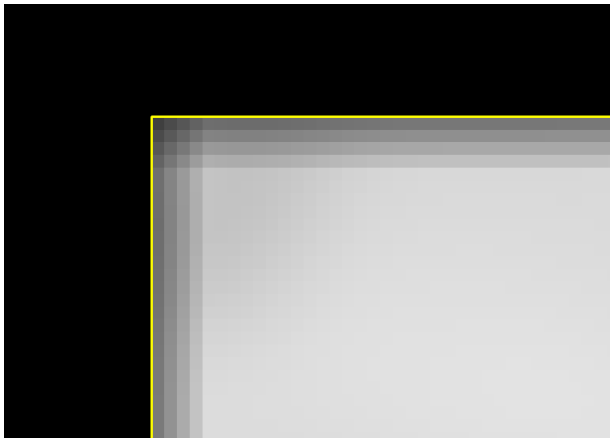
To understand this, the definition of domains in this context must be considered: For a filter, a domain defines for which input pixels output pixels must be calculated. But pixels outside the domain (which lie within the image matrix) might be used for processing nevertheless.

Problems Caused by Gray Values Outside of the Input Image Domain

Another point to notice is the handling of pixels outside the input image domain. The parameter *'init_new_image'* of set_system determines how those pixels are treated. If the pixels are initialized with 0, consecutive runs will yield identical results. Otherwise, those pixels remain undefined. While this enhances the program runtime, undefined pixels can differ from system to system, for example if parallelization is activated or not. It is merely guaranteed that the values are consistent if the program is executed repeatedly on systems with the same configuration. In certain cases, these 'undefined' pixels might lead to problems. Expanding the resulting image to the full domain with full_domain will lead to artifacts appearing outside of the former image domain.

Problems Caused when Applying Filters Consecutively

When two or more filters are applied consecutively on the same domain, the undefined or unexpected values (as described in the paragraphs above) have a higher impact on the result image. This is because with every following filter the error increases, starting from the border to the middle. In the following, four strategies for solving those problems are presented.



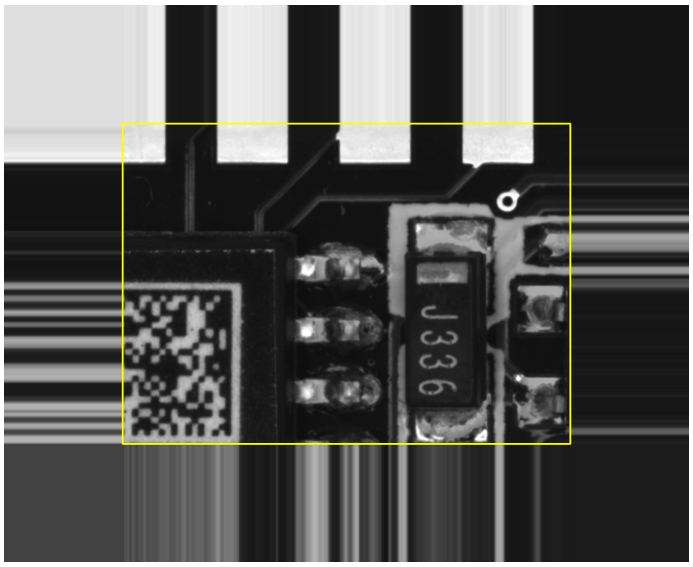
Artifacts at the domain border after applying two consecutive filters.

1. Errors caused by undefined pixels can easily be prevented by, e.g., choosing a dilated domain (see Morphology / Region) according to the filter mask. If multiple filters are applied consecutively, the image domain can be dilated in advance, considering the filter sizes to be used. For instance, when using a cascade of rectangular filters of arbitrary dimensions, the width and length dimension dim_d of the dilation mask can be calculated considering the individual filter mask dimensions dim_i and the number of filter operations n:

   $dim_d = (\sum_{i=1}^{n} dim_i) - (n-1)$.

   After the filters were applied, the image domain can be reduced to its original size (e.g., with reduce_domain).

2. Another option is to set the domain exactly to the size of the interesting part within the image and then calling the operator expand_domain_gray before applying a filter. This operator copies the pixels inside of the border to the outside of the border and therefore avoids errors caused by pixels that are undefined outside of the domain. Subsequently, the domain can again be reduced to its original size. This process should be repeated for every following filter operation. Note, however, that this option increases the runtime significantly.



The result of expand_domain_gray looks as if the boundary is copied multiple times and added at the outside.

3. If runtime is not an issue, the operator full_domain can be called before applying the first filter to the image. That way, the whole image is defined as domain and undefined pixels are avoided completely.

4. Another possibility of getting an image without undefined pixels is by calling the operator crop_domain before applying a filter. The operator crop_domain crops the image to the size of the domain, which means that the domain then covers a complete smaller image. Note, however, that for the cropped image the coordinate system has changed in relation to the original image, which will influence all following applications depending on the image coordinate system (e.g., calculating the center of gravity).

12.1 Arithmetic

abs_diff_image (Image1, Image2 : ImageAbsDiff : Mult :)

Calculate the absolute difference of two images.

abs_diff_image calculates the absolute difference between two images. The gray values g' of the output image ImageAbsDiff are calculated from the gray values $(g1, g2)$ of the input images (Image1 and Image2) as follows:

$$g' = |(g1 - g2)| * \texttt{Mult}$$

If an overflow or an underflow occurs the resulting values are clipped.

Several images can be processed in one call. In this case, both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Please note that the runtime of the operator depends on the value of Mult. For Mult = 1, a special optimization is used. Since values of Mult different from 1 cannot yield additional information, Mult = 1 should be used in applications. All other values of Mult should only be used for visualization purposes. Additionally, for byte, int1, int2, and uint2 images special optimizations are implemented for Mult = 1 that use SIMD technology. The actual application of these special optimizations is controlled by the system parameter *'mmx_enable'* (see set_system). If *'mmx_enable'* is set to *'true'* (and the SIMD instruction set is available), the internal calculations are performed using SIMD technology.

abs_diff_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, and real images. However, since for OpenCL 1.0 only single precision floating point is supported for all devices, and not all rounding modes are supported, the OpenCL implementation can produce slightly different results from the scalar or SIMD implementations.

Attention

Note that the acceleration gained by SIMD technology is highest on large, compact input regions. However, in rare cases, the execution of `abs_diff_image` might take significantly longer with SIMD technology than without, depending on the input region and the capabilities of the hardware. In these cases, the use of SIMD technology can be avoided by `set_system(::'mmx_enable','false':)`.

Parameters

▷ **`Image1`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real

Input image 1.

▷ **`Image2`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real

Input image 2.

▷ **`ImageAbsDiff`** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real

Absolute value of the difference of the input images.

▷ **`Mult`** (input_control) ..... number ⇝ *real* / integer

Scale factor.
Default: 1.0
Suggested values: `Mult` ∈ {1.0, 2.0, 3.0, 4.0}
Restriction: Mult > 0

Result

The operator `abs_diff_image` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)`. If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

`threshold`

Alternatives

`sub_image`

See also

`add_image`, `scale_image`, `dyn_threshold`

Module

Foundation

`abs_image` (Image : ImageAbs : :)

Calculate the absolute value (modulus) of an image.

The operator `abs_image` calculates the absolute gray values of images of any type and stores the result in `ImageAbs`. The power spectrum of complex images is calculated as a `real` image. For a `Image` of type 'int1', `ImageAbs` is of type `byte`. For `Image` of other signed types the type of the input image is used for `ImageAbs`. The behavior in case of overflows or underflows is undefined. The operator `abs_image` generates a logical copy of unsigned images.

`abs_image` can be executed on an OpenCL device for `int1`, `int2`, `int4`, `real`, and `complex` images. However, since for OpenCL 1.0 only single precision floating point is supported for all devices, and not all rounding

modes are supported, the OpenCL implementation can produce slightly different results from the scalar or SIMD implementations.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* int1 / int2 / int4 / int8 / real / complex
  Image(s) for which the absolute gray values are to be calculated.
- ▷ **ImageAbs** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / int4 / int8 / real
  Result image(s).

Example

```
convert_image_type (Image, ImageInt2, 'int2')
texture_laws (ImageInt2, ImageTexture, 'el', 2, 5)
abs_image (ImageTexture, ImageTexture)
```

Result

The operator `abs_image` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)`

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

`convert_image_type`, `power_byte`

Module

Foundation

acos_image (Image : ArccosImage : :)

Calculate the arccosine of an image.

`acos_image` calculates the arccosine of an input image `Image` and stores the result in the image `ArccosImage`. The angles in `ArccosImage` are represented in radians. If `Image` contains gray values outside the valid domain $[-1, 1]$ of the arccosine function, the corresponding gray values in `ArccosImage` are set to 0.

Attention

`acos_image` can be executed on OpenCL devices.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
  Input image.
- ▷ **ArccosImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
  Output image.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

asin_image, atan_image, atan2_image, tan_image, sin_image, cos_image

Module

Foundation

add_image (Image1, Image2 : ImageResult : Mult, Add :)

Add two images.

The operator add_image adds two images. The gray values $(g1, g2)$ of the input images (Image1 and Image2) are transformed as follows:

$$g' := (g1 + g2) * \texttt{Mult} + \texttt{Add}$$

If an overflow or an underflow occurs the values are clipped. Note that this is not the case with cyclic and direction images. The resulting image is stored in ImageResult.

It is possible to add byte images with int2, uint2 or int4 images and to add int4 to int2 or uint2 images. In this case the result will be of type int2 or int4 respectively.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Please note that the runtime of the operator varies with different control parameters. For frequently used combinations special optimizations are used. Additionally, for byte, int2, uint2, and int4 images special optimizations are implemented that use SIMD technology. The actual application of these special optimizations is controlled by the system parameter *'mmx_enable'* (see set_system). If *'mmx_enable'* is set to *'true'* (and the SIMD instruction set is available), the internal calculations are performed using SIMD technology.

add_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, real, direction, cyclic, and complex images. However, since for OpenCL 1.0 only single precision floating point is supported for all devices, and not all rounding modes are supported, the OpenCL implementation can produce slightly different results from the scalar or SIMD implementations.

Attention

Note that the acceleration gained by SIMD technology is highest on large, compact input regions. However, in rare cases, the execution of add_image might take significantly longer with SIMD technology than without, depending on the input region and the capabilities of the hardware. In these cases, the use of SIMD technology can be avoided by set_system(::'mmx_enable','false':).

Parameters

▷ **Image1** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Image(s) 1.

▷ **Image2** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Image(s) 2.

▷ **ImageResult** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Result image(s) by the addition.

▷ **Mult** (input_control) ..... number ⇝ *real* / integer

Factor for gray value adaption.
Default: 0.5
Suggested values: Mult ∈ {0.2, 0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 5.0}
Value range: -255.0 ≤ Mult ≤ 255.0
Minimum increment: 0.001
Recommended increment: 0.1

▷ **Add** (input_control) . . . . . number ⇝ *real* / integer
Value for gray value range adaption.
Default: 0
Suggested values: Add ∈ {0, 64, 128, 255, 512}
Value range: -512.0 ≤ Add ≤ 512.0
Minimum increment: 0.01
Recommended increment: 1.0

Example

```
read_image(Image1,'fabrik')
dev_display (Image1)
read_image(Image2,'monkey')
dev_display (Image2)
add_image(Image1,Image2,Result,0.5,10.0)
dev_display (Result)
```

Result

The operator add_image returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:) If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

sub_image, mult_image

See also

sub_image, mult_image

Module

Foundation

asin_image (Image : ArcsinImage : :)

Calculate the arcsine of an image.

asin_image calculates the arcsine of an input image Image and stores the result in the image ArcsinImage. The angles in ArcsinImage are represented in radians. If Image contains gray values outside the valid domain $[-1, 1]$ of the arcsine function, the corresponding gray values in ArcsinImage are set to 0.

Attention

asin_image can be executed on OpenCL devices.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
Input image.

▷ **ArcsinImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
Output image.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

acos_image, atan_image, atan2_image, tan_image, sin_image, cos_image

Module

Foundation

atan2_image (ImageY, ImageX : ArctanImage : :)

Calculate the arctangent of two images.

atan2_image calculates the arctangent ImageY/ImageX of the input images ImageY and ImageX, using the signs of the gray values of the two images to determine the quadrant of the result, and stores the result in the image ArctanImage. The angles in ArctanImage are represented in radians.

Attention

atan2_image can be executed on an OpenCL device for int1, int2, int4, and real images. Note that the results of the OpenCL code may vary from the results produced by the CPU.

Parameters

▷ **ImageY** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* int1 / int2 / int4 / int8 / real
Input image 1.

▷ **ImageX** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* int1 / int2 / int4 / int8 / real
Input image 2.

▷ **ArctanImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
Output image.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

atan_image

See also

acos_image, asin_image, tan_image, sin_image, cos_image

Module

Foundation

atan_image (Image : ArctanImage : :)

Calculate the arctangent of an image.

atan_image calculates the arctangent of an input image Image and stores the result in the image ArctanImage. The angles in ArctanImage are represented in radians.

Attention

`atan_image` can be executed on OpenCL devices.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
Input image.

▷ **ArctanImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
Output image.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

`atan2_image`

See also

`acos_image`, `asin_image`, `tan_image`, `sin_image`, `cos_image`

Module

Foundation

cos_image (Image : CosImage : :)

Calculate the cosine of an image.

`cos_image` calculates the cosine of an input image `Image` and stores the result in the image `CosImage` of type real. If `Image` is of type direction or cyclic, the angles in `Image` are converted to radians internally. If `Image` is of type real, the angles in `Image` are assumed to be represented in radians. For direction images, the value of 255 will result in a value of 0 in the output image.

Attention

`cos_image` can be executed on OpenCL devices.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* direction / cyclic / real
Input image.

▷ **CosImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
Output image.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

`sin_image`, `tan_image`, `asin_image`, `acos_image`, `atan_image`, `atan2_image`

Module

Foundation

div_image (Image1, Image2 : ImageResult : Mult, Add :)

Divide two images.

The operator div_image divides two images. The gray values $(g1, g2)$ of the input images (Image1) are transformed as follows:

$$g' := g1/g2 * \texttt{Mult} + \texttt{Add}$$

If an overflow or an underflow occurs the values are clipped. For a division by zero the result is set to zero.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

div_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, real and complex images. However, since for OpenCL 1.0 only single precision floating point is supported for all devices, and not all rounding modes are supported, the OpenCL implementation can produce slightly different results from the scalar implementation.

Parameters

▷ **Image1** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / complex

Image(s) 1.

▷ **Image2** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / complex

Image(s) 2.

▷ **ImageResult** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / complex

Result image(s) by the division.

▷ **Mult** (input_control) ..... number ⇝ *real* / integer

Factor for gray range adaption.
Default: 255
Suggested values: Mult ∈ {0.1, 0.2, 0.5, 1.0, 2.0, 3.0, 10, 100, 500, 1000}
Value range: -1000 ≤ Mult ≤ 1000
Minimum increment: 0.001
Recommended increment: 1

▷ **Add** (input_control) ..... number ⇝ *real* / integer

Value for gray range adaption.
Default: 0
Suggested values: Add ∈ {0.0, 128.0, 256.0, 1025}
Value range: -1000 ≤ Add ≤ 1000
Minimum increment: 0.01
Recommended increment: 1.0

Example

```
read_image(Image1,'fabrik')
dev_display (Image1)
read_image(Image2,'monkey')
dev_display (Image2)
div_image (Image1, Image2, Result, 200, 10)
dev_display (Result)
```

Result

The operator div_image returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:) If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

add_image, sub_image, mult_image

See also

add_image, sub_image, mult_image

Module

Foundation

exp_image (Image : ExpImage : Base :)

Calculate the exponentiation of an image.

exp_image calculates the exponentiation to the base Base of an input image Image and stores the result in the image ExpImage. If Image contains gray values that would overflow the range of ExpImage, e.g., > 88.722839 for Base $= 'e'$, the corresponding gray values in ExpImage are set to the largest value representable in ExpImage (i.e., 3.4028235×10^{38}).

Attention

exp_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, and real images. Note that the results of the OpenCL code may vary from the results produced by the CPU.

Parameters

▷ **Image** (input_object) . . . . (multichannel-)image(-array) ⇝ *object :* byte / int1 / uint2 / int2 / int4 / int8 / real
Input image.

▷ **ExpImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
Output image.

▷ **Base** (input_control) . . . . . number ⇝ *string* / integer / real
Base of the exponentiation.
Default: 'e'
Suggested values: Base ∈ {'e', 2, 10}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

pow_image, log_image

Module

Foundation

gamma_image (Image : GammaImage : Gamma, Offset, Threshold, MaxGray, Encode :)

Perform a gamma encoding or decoding of an image.

gamma_image performs a general gamma encoding or decoding of the input image Image and returns the resulting image in GammaImage. A generalized gamma encoding can be described as follows:

$$G' = \begin{cases} m\left((1+o)(G/m)^{\gamma} - o\right), & t < G/m \leq 1 \\ sG, & 0 \leq G/m \leq t \end{cases}$$

Here, G' is the gamma-encoded gray value, G is the linear gray value, $\gamma =$ Gamma, $o =$ Offset, $m =$ MaxGray, $t =$ Threshold, and s is a factor that is computed from Gamma, Offset, Threshold, and MaxGray in such a way that the linear and exponential parts of the transformation are continuous.

Analogously, a generalized gamma decoding can be described as follows:

$$G = \begin{cases} m\left(\frac{G'/m+o}{1+o}\right)^{1/\gamma}, & t' < G'/m \leq 1 \\ G'/s, & 0 \leq G'/m \leq t' \end{cases}$$

where the variables have identical meanings as for the gamma encoding and $t' = st$.

For example, the gamma encoding or decoding required by the sRGB standard can be obtained by setting Gamma $= 1.0/2.4$, Offset $= 0.055$, and Threshold $= 0.0031308$. Similarly, the gamma encoding or decoding required by the HDTV video standard can be obtained by setting Gamma $= 0.45$, Offset $= 0.099$, and Threshold $= 0.018$. In any case, MaxGray should be set as appropriate for the image type of Image (e.g., 255.0 for byte images).

Attention

gamma_image can be executed on an OpenCL device for byte, uint2 and real images. As the operation is performed in single precision floating point instead of double precision as on the CPU, the result of the OpenCL implementation can vary slightly from that of the CPU implementation.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **GammaImage** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Output image.

▷ **Gamma** (input_control) ..... real ⇝ *real*
Gamma coefficient of the exponential part of the transformation.
Default: 0.416666666667
Suggested values: Gamma ∈ {0.416666666667, 0.45}

▷ **Offset** (input_control) ..... real ⇝ *real*
Offset of the exponential part of the transformation.
Default: 0.055
Suggested values: Offset ∈ {0.055, 0.099}

▷ **Threshold** (input_control) ..... real ⇝ *real*
Gray value for which the transformation switches from linear to exponential.
Default: 0.0031308
Suggested values: Threshold ∈ {0.0031308, 0.018}

▷ **MaxGray** (input_control) ..... number ⇝ *real* / integer
Maximum gray value of the input image type.
Default: 255.0
Suggested values: MaxGray ∈ {1.0, 255.0, 511.0, 1023.0, 4095.0, 16383.0, 65535.0}

▷ **Encode** (input_control) ..... string ⇝ *string*
If *'true'*, perform a gamma encoding, otherwise a gamma decoding.
Default: 'true'
List of values: Encode ∈ {'true', 'false'}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

pow_image

See also

sqrt_image, exp_image, log_image

References

Erik Reinhard, Erum Arif Khan, Ahmet Oguz Akyüz, Garret M. Johnson: "Color Imaging: Fundamentals and Applications." A K Peters, Ltd., 2008.

CEI/IEC 61966-2-1:1999: "Colour management – Default RGB colour space – sRGB."

ITU-R BT.709-5: "Parameter values for the HDTV standards for production and international programme exchange."

Module

Foundation

invert_image (Image : ImageInvert : :)

Invert an image.

The operator invert_image inverts the gray values of an image. For images of the byte and cyclic type the result is calculated as:

$$g' = 255 - g$$

Images of the 'direction' type are transformed by

$$g' = (g + 90) \text{ modulo } 180$$

In the case of signed types the values are negated. The resulting image has the same pixel type as the input image.

Several images can be processed in one call. An output image is generated for every input image.

invert_image can be executed on an OpenCL device for byte, direction, cyclic, int1, int2, uint2, int4, and real images.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Input image(s).

▷ **ImageInvert** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Image(s) with inverted gray values.

Example

```
read_image(Orig,'fabrik')
invert_image(Orig,Invert)
dev_display(Invert)
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

watersheds

Alternatives

scale_image

See also

scale_image, add_image, sub_image

Module

Foundation

log_image (Image : LogImage : Base :)

Calculate the logarithm of an image.

log_image calculates the logarithm to the base Base of an input image Image and stores the result in the image LogImage. If Image contains gray values outside the valid domain of the logarithm function, i.e., ≤ 0, the corresponding gray values in LogImage are set to 0.

Attention

log_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, and real images. Note that the results of the OpenCL code may vary from the results produced by the CPU.

Parameters

▷ **Image** (input_object) . . . . (multichannel-)image(-array) ⇝ *object :* byte / int1 / uint2 / int2 / int4 / int8 / real
Input image.

▷ **LogImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
Output image.

▷ **Base** (input_control) . . . . . number ⇝ *string* / integer / real
Base of the logarithm.
Default: 'e'
Suggested values: Base ∈ {'e', 2, 10}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

pow_image, exp_image

Module

Foundation

max_image (Image1, Image2 : ImageMax : :)

Calculate the maximum of two images pixel by pixel.

max_image calculates the maximum of the images Image1 and Image2 (pixel by pixel). The result is stored in the image ImageMax. The resulting image has the same pixel type as the input image. If several (pairs of) images

are processed in one call, every i-th image from Image1 is compared to the i-th image from Image2. Thus the number of images in both input parameters must be the same. An output image is generated for every input pair.

max_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, real, direction, and cyclic images.

Attention

The two input images must be of the same type and size.

Parameters

▷ **Image1** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic

Image(s) 1.

▷ **Image2** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic

Image(s) 2.

▷ **ImageMax** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic

Result image(s) by the maximization.

Example

```
read_image(Image1,'monkey')
read_image(Image2,'fabrik')
max_image(Image1,Image2,Max)
dev_display (Max)
```

Result

If the parameter values are correct the operator max_image returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:). If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

min_image

Module

Foundation

min_image (Image1, Image2 : ImageMin : :)

Calculate the minimum of two images pixel by pixel.

The operator min_image determines the minimum (pixel by pixel) of the images Image1 and Image2. The result is stored in the image ImageMin. The resulting image has the same pixel type as the input image. If several (pairs of) images are processed in one call, every i-th image from Image1 is compared to the i-th image from Image2. Thus the number of images in both input parameters must be the same. An output image is generated for every input pair.

min_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, real, direction, and cyclic images.

Parameters

▷ **Image1** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic

Image(s) 1.

▷ **Image2** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic

Image(s) 2.

▷ **ImageMin** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic

Result image(s) by the minimization.

Result

If the parameter values are correct the operator `min_image` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)` If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

`gray_erosion`

See also

`max_image`

Module

Foundation

mult_image (Image1, Image2 : ImageResult : Mult, Add :)

Multiply two images.

`mult_image` multiplies two images. The gray values $(g1, g2)$ of the input images (`Image1`) are transformed as follows:

$$g' := g1 * g2 * \texttt{Mult} + \texttt{Add}$$

If an overflow or an underflow occurs the values are clipped. Note that this is not the case with cyclic and direction images.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

`mult_image` can be executed on an OpenCL device for byte, int1, int2, uint2, int4, real, direction, cyclic, and complex images. However, since for OpenCL 1.0 only single precision floating point is supported for all devices, and not all rounding modes are supported, the OpenCL implementation can produce slightly different results from the scalar implementation.

Parameters

▷ **Image1** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Image(s) 1.

▷ **Image2** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Image(s) 2.

▷ **ImageResult** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Result image(s) by the product.

▷ **Mult** (input_control) ..... number ⇝ *real* / integer

Factor for gray range adaption.
Default: 0.005
Suggested values: `Mult` ∈ {0.001, 0.01, 0.5, 1.0, 2.0, 3.0, 5.0, 10.0}
Value range: -255.0 ≤ `Mult` ≤ 255.0
Minimum increment: 0.001
Recommended increment: 0.1

▷ **Add** (input_control) ..... number ⇝ *real* / integer

Value for gray range adaption.
Default: 0
Suggested values: `Add` ∈ {0.0, 128.0, 256.0}
Value range: -512.0 ≤ `Add` ≤ 512.0
Minimum increment: 0.01
Recommended increment: 1.0

Example

```
read_image(Image1,'fabrik')
dev_display (Image1)
read_image(Image2,'monkey')
dev_display (Image2)
mult_image (Image1, Image2, Result, 0.01, 10)
dev_display (Result)
```

Result

The operator `mult_image` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)` If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

`add_image`, `sub_image`, `div_image`

See also

`add_image`, `sub_image`, `div_image`

Module

Foundation

pow_image (Image : PowImage : Exponent :)

Raise an image to a power.

pow_image raises the gray values of the input image Image to the power Exponent and stores the result in the image PowImage. If Image contains gray values that would overflow the range of PowImage, e.g., > 7131.55017 for Exponent $= 10$, the corresponding gray values in PowImage are set to the largest value representable in PowImage (i.e., 3.4028235×10^{38}). If Image contains gray values that cannot be raised to the power Exponent, i.e., if the gray values are negative and Exponent is not an integer, the corresponding gray values in PowImage are set to 0.

Attention

pow_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, and real images. Note that the results of the OpenCL code may vary from the results produced by the CPU.

Parameters

- ▷ **Image** (input_object) .... (multichannel-)image(-array) ⇝ *object :* byte / int1 / uint2 / int2 / int4 / int8 / real
  Input image.
- ▷ **PowImage** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* real
  Output image.
- ▷ **Exponent** (input_control) ..... number ⇝ *real* / integer
  Power to which the gray values are raised.
  Default: 2
  Suggested values: Exponent ∈ {0.25, 0.5, 2, 3, 4}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

gamma_image

See also

sqrt_image, exp_image, log_image

Module

Foundation

scale_image (Image : ImageScaled : Mult, Add :)

Scale the gray values of an image.

The operator scale_image scales the input images (Image) by the following transformation:

$$g' := g * \texttt{Mult} + \texttt{Add}$$

If an overflow or an underflow occurs the values are clipped. Note that this is not the case with cyclic and direction images.

This operator can be applied, e.g., to map the gray values of an image, i.e., the interval [GMin,GMax], to the maximum range [0:255]. For this, the parameters are chosen as follows:

$$\texttt{Mult} = \frac{255}{\text{GMax} - \text{GMin}} \qquad \texttt{Add} = -\texttt{Mult} * \text{GMin}$$

The values for GMin and GMax can be determined, e.g., with the operator `min_max_gray`.

Please note that the runtime of the operator varies with different control parameters. For frequently used combinations special optimizations are used. Additionally, special optimizations are implemented that use fixed point arithmetic (for int2 and uint2 images), and further optimizations that use SIMD technology (for byte, int2, and uint2 images). The actual application of these special optimizations is controlled by the system parameters *'int_zooming'* and *'mmx_enable'* (see `set_system`). If *'int_zooming'* is set to *'true'*, the internal calculation is performed using fixed point arithmetic, leading to much shorter execution times. However, the accuracy of the transformed gray values is slightly lower in this mode. The difference to the more accurate calculation (using *'int_zooming'* = *'false'*) is typically less than two gray levels. If *'mmx_enable'* is set to *'true'*(and the SIMD instruction set is available), the internal calculations are performed using fixed point arithmetic and SIMD technology. In this case the setting of *'int_zooming'* is ignored.

`scale_image` can be executed on an OpenCL device for byte, int1, int2, uint2, int4, real, direction, cyclic, and complex images. However, since for OpenCL 1.0 only single precision floating point is supported for all devices, and not all rounding modes are supported, the OpenCL implementation can produce slightly different results from the scalar or SIMD implementations.

Attention

Note that the acceleration gained by SIMD technology is highest on large, compact input regions. However, in rare cases, the execution of `scale_image` might take significantly longer with SIMD technology than without, depending on the input region and the capabilities of the hardware. In these cases, the use of SIMD technology can be avoided by `set_system(::'mmx_enable','false':)`.

Parameters

▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Image(s) whose gray values are to be scaled.

▷ **`ImageScaled`** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Result image(s) by the scale.

▷ **`Mult`** (input_control) ..... number ⇝ *real* / integer

Scale factor.
Default: 0.01
Suggested values: `Mult` ∈ {0.001, 0.003, 0.005, 0.008, 0.01, 0.02, 0.03, 0.05, 0.08, 0.1, 0.5, 1.0}
Minimum increment: 0.001
Recommended increment: 0.1

▷ **`Add`** (input_control) ..... number ⇝ *real* / integer

Offset.
Default: 0
Suggested values: `Add` ∈ {0, 10, 50, 100, 200, 500}
Minimum increment: 0.01
Recommended increment: 1.0

Example

```
* Complement of the gray values:
scale_image(Image,Invert,-1.0,255.0)
```

Result

The operator `scale_image` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)` Otherwise an exception treatment is carried out.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

min_max_gray

Alternatives

mult_image, add_image, sub_image

See also

min_max_gray

Module

Foundation

sin_image (Image : SinImage : :)

Calculate the sine of an image.

sin_image calculates the sine of an input image Image and stores the result in the image SinImage of type real. If Image is of type direction or cyclic, the angles in Image are converted to radians internally. If Image is of type real, the angles in Image are assumed to be represented in radians. For direction images, the value of 255 will result in a value of 0 in the output image.

Attention

sin_image can be executed on OpenCL devices.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* direction / cyclic / real
Input image.

▷ **SinImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
Output image.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

cos_image, tan_image, asin_image, acos_image, atan_image, atan2_image

Module

Foundation

sqrt_image (Image : SqrtImage : :)

Calculate the square root of an image.

sqrt_image calculates the square root of an input image Image and stores the result in the image SqrtImage of the same pixel type. In case the picture Image is of a signed pixel type, negative pixel values will be mapped to zero in SqrtImage.

sqrt_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, and real images.

Parameters

▷ **Image** (input_object) . . . . (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real
Input image

▷ **SqrtImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real
Output image

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

pow_image

Module

Foundation

sub_image (ImageMinuend, ImageSubtrahend : ImageSub : Mult, Add :)

Subtract two images.

The operator sub_image subtracts two images. The gray values $(g1, g2)$ of the input images (ImageMinuend and ImageSubtrahend) are transformed as follows:

$$g' := (g1 - g2) * \texttt{Mult} + \texttt{Add}$$

If an overflow or an underflow occurs the values are clipped. Note that this is not the case with cyclic and direction images.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Please note that the runtime of the operator varies with different control parameters. For frequently used combinations special optimizations are used. Additionally, for byte, int2, and uint2 images special optimizations are implemented that use SIMD technology. The actual application of these special optimizations is controlled by the system parameter *'mmx_enable'* (see set_system). If *'mmx_enable'* is set to *'true'* (and the SIMD instruction set is available), the internal calculations are performed using SIMD technology.

sub_image can be executed on an OpenCL device for byte, int1, int2, uint2, int4, real, direction, cyclic, and complex images. However, since for OpenCL 1.0 only single precision floating point is supported for all devices, and not all rounding modes are supported, the OpenCL implementation can produce slightly different results from the scalar or SIMD implementations.

Attention

Note that the acceleration gained by SIMD technology is highest on large, compact input regions. However, in rare cases, the execution of sub_image might take significantly longer with SIMD technology than without, depending on the input region and the capabilities of the hardware. In these cases, the use of SIMD technology can be avoided by set_system(::'mmx_enable','false':).

Parameters

▷ **ImageMinuend** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Minuend(s).

▷ **ImageSubtrahend** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Subtrahend(s).

▷ **ImageSub** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / int8 / real / direction / cyclic / complex

Result image(s) by the subtraction.

▷ **Mult** (input_control) . . . . . number ⇝ *real* / integer

Correction factor.
Default: 1.0
Suggested values: Mult ∈ {0.0, 1.0, 2.0, 3.0, 4.0}
Value range: -255.0 ≤ Mult ≤ 255.0
Minimum increment: 0.001
Recommended increment: 0.1

▷ **Add** (input_control) . . . . . number ⇝ *real* / integer

Correction value.
Default: 128.0
Suggested values: Add ∈ {0.0, 128.0, 256.0}
Value range: -512.0 ≤ Add ≤ 512.0
Minimum increment: 0.01
Recommended increment: 1.0

Example

```
read_image(Image1,'fabrik')
dev_display (Image1)
read_image(Image2,'monkey')
dev_display (Image2)
sub_image (Image1, Image2, Result, 1, 100)
dev_display (Result)
```

Result

The operator sub_image returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:) If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

dual_threshold

Alternatives

mult_image, add_image

See also

add_image, mult_image, dyn_threshold, check_difference

Module

Foundation

tan_image (Image : TanImage : :)

Calculate the tangent of an image.

tan_image calculates the tangent of an input image Image and stores the result in the image TanImage of type real. If Image is of type direction or cyclic, the angles in Image are converted to radians internally. If Image is of type real, the angles in Image are assumed to be represented in radians. For direction images, the value of 255 will result in a value of 0 in the output image.

Attention

tan_image can be executed on OpenCL devices.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* direction / cyclic / real
  Input image.
- ▷ **TanImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* real
  Output image.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

sin_image, cos_image, asin_image, acos_image, atan_image, atan2_image

Module

Foundation

12.2 Bit

bit_and (Image1, Image2 : ImageAnd : :)

Bit-by-bit AND of all pixels of the input images.

The operator bit_and calculates the "and" of all pixels of the input images bit by bit. The semantics of the "and" operation corresponds to that of C for the respective types (signed char, unsigned char, short, unsigned short, int/long). The images must have the same size and pixel type. The pixels within the definition range of the image in the first parameter are processed.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Parameters

- ▷ **Image1** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4
  Input image(s) 1.
- ▷ **Image2** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4
  Input image(s) 2.
- ▷ **ImageAnd** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4
  Result image(s) by AND-operation.

Example

```
read_image(Image1,'fabrik')
dev_display (Image1)
read_image(Image2,'monkey')
dev_display (Image2)
bit_and(Image1,Image2,ImageBitAnd)
dev_display (ImageBitAnd)
```

Result

If the images are correct (type and number) the operator `bit_and` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)` If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

`bit_mask`, `add_image`, `max_image`

See also

`bit_mask`, `add_image`, `max_image`

Module

Foundation

`bit_lshift` (Image : ImageLShift : Shift :)

Left shift of all pixels of the image.

The operator `bit_lshift` calculates a "left shift" of all pixels of the input image bit by bit. The semantics of the "left shift" operation corresponds to that of C ("«") for the respective types (signed char, unsigned char, short, unsigned short, int/long). If an overflow occurs the result is limited to the maximum value of the respective pixel type. Only the pixels within the definition range of the image are processed.

Several images can be processed in one call. An output image is generated for every input image.

Parameters

▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4
Input image(s).

▷ **`ImageLShift`** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4
Result image(s) by shift operation.

▷ **`Shift`** (input_control) ..... integer ⇝ *integer*
Shift value.
Default: 3
Suggested values: `Shift` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, 30, 31}
Minimum increment: 1
Recommended increment: 1

Example

```
read_image(&ByteImage,"fabrik");
convert_image_type(ByteImage,&Int2Image,"int2");
bit_lshift(Int2Image,&FullInt2Image,8);
```

Result

If the images are correct (type) and if `Shift` has a valid value the operator `bit_lshift` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)` If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

`scale_image`

See also

`bit_rshift`

Module

Foundation

bit_mask (Image : ImageMask : BitMask :)

Logical "AND" of each pixel using a bit mask.

The operator `bit_mask` carries out an "and" operation of each pixel with a fixed mask. The semantics of the "and" operation corresponds to that of C for the respective types (signed char, unsigned char, unsigned short, short, int/long). Only the pixels within the definition range of the image are processed.

Several images can be processed in one call. An output image is generated for every input image.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Input image(s).

▷ **ImageMask** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Result image(s) by combination with mask.

▷ **BitMask** (input_control) ..... integer ⇝ *integer*

Bit field
Default: 128
Suggested values: `BitMask` ∈ {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096}

Result

If the images are correct (type) the operator `bit_mask` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)` If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

threshold, bit_or

Alternatives

bit_slice

See also

bit_and, bit_lshift

Module

Foundation

bit_not (Image : ImageNot : :)

Complement all bits of the pixels.

The operator bit_not calculates the "complement" of all pixels of the input image bit by bit. The semantics of the "complement" operation corresponds to that of C ("~") for the respective types (signed char, unsigned char, short, unsigned short, int/long). Only the pixels within the definition range of the image are processed.

Several images can be processed in one call. An output image is generated for every input image.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Input image(s).

▷ **ImageNot** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Result image(s) by complement operation.

Example

```
read_image(Image,'monkey')
dev_display(Image)
bit_not(Image,ImageBitNot)
dev_display(ImageBitNot)
```

Result

If the images are correct (type) the operator bit_not returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:) If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

bit_or, bit_and, add_image

See also

bit_slice, bit_mask

Module

Foundation

bit_or (Image1, Image2 : ImageOr : :)

Bit-by-bit OR of all pixels of the input images.

The operator bit_or calculates the "or" of all pixels of the input images bit by bit. The semantics of the "or"operation corresponds to that of C for the respective types (signed char, unsigned char, short, unsigned short, int/long). The images must have the same size and pixel type. The pixels within the definition range of the image in the first parameter are processed.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Parameters

▷ **Image1** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Input image(s) 1.

▷ **Image2** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Input image(s) 2.

▷ **ImageOr** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Result image(s) by OR-operation.

Example

```
read_image(Image1,'fabrik')
dev_display (Image1)
read_image(Image2,'monkey')
dev_display (Image2)
bit_or(Image1,Image2,ImageBitOr)
dev_display (ImageBitOr)
```

Result

If the images are correct (type and number) the operator bit_or returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:) If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

bit_and, add_image

See also

bit_xor, bit_and

Module

Foundation

bit_rshift (Image : ImageRShift : Shift :)

Right shift of all pixels of the image.

The operator bit_rshift calculates a “right shift” of all pixels of the input image bit by bit. The semantics of the “right shift” operation corresponds to that of C (“»”) for the respective types (signed char, unsigned char, short, unsigned short, int/long). Only the pixels within the definition range of the image are processed.

Several images can be processed in one call. An output image is generated for every input image.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Input image(s).

▷ **ImageRShift** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Result image(s) by shift operation.

▷ **Shift** (input_control) . . . . . integer ⇝ *integer*

shift value
Default: 3
Suggested values: Shift ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, 30, 31}
Minimum increment: 1
Recommended increment: 1

Example

```
bit_rshift(Int2Image,&ReducedInt2Image,8);
convert_image_type(ReducedInt2Image,&ByteImage,"byte");
```

Result

If the images are correct (type) and Shift has a valid value the operator bit_rshift returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:) If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

scale_image

See also

bit_lshift

Module

Foundation

bit_slice (Image : ImageSlice : Bit :)

Extract a bit from the pixels.

The operator bit_slice extracts a bit level from the input image. The semantics of the “and” operation corresponds to that of C for the respective types (signed char, unsigned char, short, unsigned short, int/long). Only the pixels within the definition range of the image are processed.

Several images can be processed in one call. An output image is generated for every input image.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4
Input image(s).

▷ **ImageSlice** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4
Result image(s) by extraction.

▷ **Bit** (input_control) ..... integer ⇝ *integer*
Bit to be selected.
Default: 8
Suggested values: `Bit` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, 30, 32}
Minimum increment: 1
Recommended increment: 1

Example

```
read_image(Image,'fabrik')
for I:= 1 to 8 by 1
    bit_slice (Image, ImageSlice, I)
    threshold (ImageSlice, Region, 1, 255)
    dev_display (Region)
endfor
```

Result

If the images are correct (type) and `Bit` has a valid value, the operator `bit_slice` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)` If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

`threshold`, `bit_or`

Alternatives

`bit_mask`

See also

`bit_and`, `bit_lshift`

Module

Foundation

bit_xor (Image1, Image2 : ImageXor : :)

Bit-by-bit XOR of all pixels of the input images.

The operator `bit_xor` calculates the "xor" of all pixels of the input images bit by bit. The semantics of the "xor" operation corresponds to that of C for the respective types (signed char, unsigned char, short, unsigned short, int/long). The images must have the same size and pixel type. The pixels within the definition range of the image in the first parameter are processed.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Parameters

▷ **Image1** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Input image(s) 1.

▷ **Image2** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Input image(s) 2.

▷ **ImageXor** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4

Result image(s) by XOR-operation.

Example

```
read_image(Image1,'fabrik')
dev_display (Image1)
read_image(Image2,'monkey')
dev_display (Image2)
bit_xor(Image1,Image2,ImageBitXor)
dev_display (ImageBitXor)
```

Result

If the parameter values are correct the operator `bit_xor` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) can be determined by the operator `set_system(::'no_object_result',<Result>:)` If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

`bit_or`, `bit_and`, `add_image`

See also

`bit_or`, `bit_and`

Module

Foundation

12.3 Color

apply_color_trans_lut (Image1, Image2, Image3 : ImageResult1, ImageResult2, ImageResult3 : ColorTransLUTHandle :)

Color space transformation using pre-generated look-up-table.

`apply_color_trans_lut` transforms an 3-channel image from the RGB color space to further color spaces and vice versa using a pre-generated look-up-table. The three channels of the image are passed as three separate images on input and output.

For further information about the color space transformations see operators `trans_from_rgb` and `trans_to_rgb`

Parameters

- ▷ **Image1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Input image (channel 1).
- ▷ **Image2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Input image (channel 2).
- ▷ **Image3** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Input image (channel 3).
- ▷ **ImageResult1** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Color-transformed output image (channel 1).
- ▷ **ImageResult2** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Color-transformed output image (channel 2).
- ▷ **ImageResult3** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Color-transformed output image (channel 3).
- ▷ **ColorTransLUTHandle** (input_control) ..... color_trans_lut ⇝ *handle*
  Handle of the look-up-table for the color space transformation.

Result

The operator `apply_color_trans_lut` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct and the input images could be successfully transformed. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

`create_color_trans_lut`

Possible Successors

`clear_color_trans_lut`

Alternatives

`trans_from_rgb`, `trans_to_rgb`

See also

`create_color_trans_lut`, `clear_color_trans_lut`

Module

Foundation

cfa_to_rgb (CFAImage : RGBImage : CFAType, Interpolation :)

Convert a single-channel color filter array image into an RGB image.

`cfa_to_rgb` converts a single-channel color filter array image `CFAImage` into an RGB image `RGBImage`. Color filter array images are typically generated by single-chip CCD cameras. The conversion from color filter array image to RGB image is typically done on the camera itself or is performed by the device driver of the frame grabber that is used to grab the image. In some cases, however, the device driver simply passes the color filter array image through unchanged. In this case, the corresponding HALCON frame grabber interface typically converts the image into an RGB image. Hence, the operator `cfa_to_rgb` is normally used if the images are not being grabbed using the HALCON frame grabber interface (`grab_image` or `grab_image_async`), but are grabbed using function calls from the frame grabber SDK, and are passed to HALCON using `gen_image1` or `gen_image1_extern`.

In single-chip CCD cameras, a color filter array in front of the sensor provides (subsampled) color information. The most frequently used filter is the so-called Bayer filter. The color filter array has the following layout in this case:

G	B	G	B	G	B	⋯
R	G	R	G	R	G	⋯
G	B	G	B	G	B	⋯
R	G	R	G	R	G	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋱

Each gray value of the input image `CFAImage` corresponds to the brightness of the pixel behind the corresponding color filter. Hence, in the above layout, the pixel (0,0) corresponds to a green color value, while the pixel (0,1) corresponds to a blue color value. The layout of the Bayer filter is completely determined by the first two elements of the first row of the image, and can be chosen with the parameter `CFAType`. In particular, this enables the correct conversion of color filter array images that have been cropped out of a larger image (e.g., using `crop_part` or `crop_rectangle1`).

The algorithm that is used to interpolate the RGB values is determined by the parameter `Interpolation`. For `Interpolation` = *'bilinear'*, a bilinear interpolation is performed. While this algorithm is very fast, it typically leads to "zipper-like" artifacts and color artifacts at strong edges. For `Interpolation` = *'bilinear_dir'*, a modified version of the bilinear interpolation is performed that may lead to fewer zipper-like artifacts, especially at horizontal or vertical edges in the image. The results may still exhibit color artifacts at strong edges, however. The runtime of this algorithm is only slightly longer than that of bilinear interpolation. For `Interpolation` = *'bilinear_enhanced'*, an enhanced version of the bilinear interpolation is performed that produces fewer zipper-like artifacts and color artifacts than the other two bilinear algorithms in most cases. The runtime of this algorithm is significantly longer than that of the other two algorithms.

If *'mmx_enable'* is set to *'true'* (and the SIMD instruction set is available), the internal calculations for `byte` images are performed using SIMD technology for `Interpolation` = *'bilinear'* and `Interpolation` = *'bilinear_dir'*.

For `Interpolation` = *'bilinear'* and `Interpolation` = *'bilinear_dir'*, `cfa_to_rgb` can be executed on OpenCL devices. The width of the input image should be a multiple of four for `byte` images, or two for `uint2` images, as the operation will be much slower otherwise.

Parameters

▷ **`CFAImage`** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2
Input image.

▷ **`RGBImage`** (output_object) ..... multichannel-image(-array) ⇝ *object :* byte / uint2
Output image.

▷ **`CFAType`** (input_control) ..... string ⇝ *string*
Color filter array type.
Default: 'bayer_gb'
List of values: `CFAType` ∈ {'bayer_gb', 'bayer_gr', 'bayer_bg', 'bayer_rg'}

▷ **`Interpolation`** (input_control) ..... string ⇝ *string*
Interpolation type.
Default: 'bilinear'
List of values: `Interpolation` ∈ {'bilinear', 'bilinear_dir', 'bilinear_enhanced'}

Result

`cfa_to_rgb` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system(::'no_object_result', <Result>:)`. If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

gen_image1_extern, gen_image1, grab_image

Possible Successors

decompose3

See also

trans_from_rgb

Module

Foundation

clear_color_trans_lut (: : ColorTransLUTHandle :)

Release the look-up-table needed for color space transformation.

The operator clear_color_trans_lut frees the memory of the look-up-table created by create_color_trans_lut. After calling clear_color_trans_lut the handle ColorTransLUTHandle becomes invalid.

Parameters

▷ **ColorTransLUTHandle** (input_control) ..... color_trans_lut ⇝ *handle*
Handle of the look-up-table handle for the color space transformation.

Result

The operator clear_all_color_trans_luts returns the value 2 (H_MSG_TRUE) if the look-up-table was successfully released. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_color_trans_lut, apply_color_trans_lut

See also

create_color_trans_lut, apply_color_trans_lut

Module

Foundation

create_color_trans_lut (: : ColorSpace, TransDirection, NumBits : ColorTransLUTHandle)

Creates the look-up-table for transformation of an image from the RGB color space to an arbitrary color space.

The operator create_color_trans_lut returns a look-up table (LUT) for color space transformation of a 8 bit RGB image to a different color space and vice versa. That means, this operator is only valid for three channel images, where each channel has pixels of type byte. If the input images have different domains, only the intersection of the three input images is used for the transformation.

The parameter ColorSpace identifies the initial or the target color space transformation and the direction of the transformation is set by TransDirection. ColorTransLUTHandle contains the handle of the created look-up-table.

Using a LUT significantly speeds up the process of transformation a RGB image to another color space and vice versa with exception of the *'yiq'*, *'yuv'*, *'argyb'*, *'ciexyz'*, and *'i1i2i3'* color spaces. Be aware that the LUT already needs 48MB of memory space.

For further information about the possible color space transformations see the description of the operators trans_from_rgb and trans_to_rgb.

Parameters

▷ **ColorSpace** (input_control) . . . . . string ⇝ *string*
Color space of the output image.
Default: 'hsv'
List of values: ColorSpace ∈ {'yiq', 'yuv', 'argyb', 'ciexyz', 'ciexyz4', 'cielab', 'cielchab', 'cieluv', 'cielchuv', 'hls', 'hsi', 'hsv', 'ciexyz3', 'ciexyz4', 'hls', 'ihs', 'i1i2i3', 'lms'}

▷ **TransDirection** (input_control) . . . . . string ⇝ *string*
Direction of color space transformation.
Default: 'from_rgb'
List of values: TransDirection ∈ {'from_rgb', 'to_rgb'}

▷ **NumBits** (input_control) . . . . . integer ⇝ *integer*
Number of bits of the input image.
Default: 8
List of values: NumBits ∈ {8}

▷ **ColorTransLUTHandle** (output_control) . . . . . color_trans_lut ⇝ *handle*
Handle of the look-up-table for color space transformation.

Result

The operator create_color_trans_lut returns the value 2 (H_MSG_TRUE) if the given parameters are correct and the look-up-table is generated. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

apply_color_trans_lut

See also

apply_color_trans_lut, clear_color_trans_lut

Module

Foundation

gen_principal_comp_trans (MultichannelImage : : : Trans, TransInv, Mean, Cov, InfoPerComp)

Compute the transformation matrix of the principal component analysis of multichannel images.

gen_principal_comp_trans computes the transformation matrix of a principal components analysis of multichannel images. This is useful for images obtained, e.g., with the thematic mapper of the Landsat satellite. Because the spectral bands are highly correlated, it is desirable to transform them to uncorrelated images. This can be used to save storage, since the bands containing little information can be discarded, and with respect to a later classification step.

The operator gen_principal_comp_trans takes one or more multichannel images MultichannelImage and computes the transformation matrix Trans for the principal components analysis, as well as its inverse TransInv. All input images must have the same number of channels. The principal components analysis is performed based on the collection of data of all images. Hence, gen_principal_comp_trans facilitates using the statistics of multiple images.

If n is the number of channels, Trans and TransInv are matrices of dimension $n \times (n+1)$, which describe an affine transformation of the multichannel gray values. They can be used to transform a multichannel image with linear_trans_color. For information purposes, the mean gray value of the channels and the $n \times n$ covariance matrix of the channels are returned in Mean and Cov, respectively. The parameter InfoPerComp contains the relative information content of each output channel.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **MultichannelImage** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Multichannel input image.

▷ **Trans** (output_control) ..... real-array ⇝ *real*
Transformation matrix for the computation of the PCA.

▷ **TransInv** (output_control) ..... real-array ⇝ *real*
Transformation matrix for the computation of the inverse PCA.

▷ **Mean** (output_control) ..... real-array ⇝ *real*
Mean gray value of the channels.

▷ **Cov** (output_control) ..... real-array ⇝ *real*
Covariance matrix of the channels.

▷ **InfoPerComp** (output_control) ..... real-array ⇝ *real*
Information content of the transformed channels.

Result

The operator `gen_principal_comp_trans` returns the value 2 (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`linear_trans_color`

Alternatives

`principal_comp`

Module

Foundation

linear_trans_color (Image : ImageTrans : TransMat :)

Compute an affine transformation of the color values of a multichannel image.

`linear_trans_color` performs an affine transformation of the color values of the multichannel image `Image` and returns the result in `ImageTrans`. The affine transformation of the color values is described by the transformation matrix `TransMat`. If n is the number of channels in `Image` and m is the number of channels in `ImageTrans`, `TransMat` is a homogeneous $m \times (n+1)$ matrix that is stored row by row. Homogeneous means that the left $m \times n$ submatrix of `TransMat` describes a linear transformation of the color values, while the last column of `TransMat` describes a constant offset of the color values. The transformation matrix is typically computed with `gen_principal_comp_trans`. In particular, it is possible to project an image onto the first m principal components by selecting the first $m \times (n+1)$ values of the transformation returned by `gen_principal_comp_trans`. The transformation can, however, also be specified directly. For example, a transformation from RGB to YIQ, which is described by the following transformation

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ 0.599 & -0.276 & -0.324 \\ 0.214 & -0.522 & 0.309 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 127.5 \\ 127.5 \end{pmatrix}$$

can be achieved by setting `TransMat` to

$$[0.299, 0.587, 0.114, 0.0, 0.599, -0.276, -0.324, 127.5, 0.214, -0.522, 0.309, 127.5]$$

Here, it should be noted that the above transformation is unnormalized, i.e., the resulting color values can lie outside the range $[0, 255]$. The transformation *'yiq'* in `trans_from_rgb` additionally scales the rows of the matrix (except for the constant offset) appropriately.

To avoid a loss of information, `linear_trans_color` returns an image of type real. If a different image type is desired, the image can be transformed with `convert_image_type`.

Attention

`linear_trans_color` can be executed on OpenCL devices if the image `Image` consists of nine channels or less and is transformed to an image of three channels or less. Since the calculations are done in single precision floating point, the results may differ from those calculated by the CPU.

Parameters

▷ **Image** (input_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Multichannel input image.

▷ **ImageTrans** (output_object) ..... multichannel-image(-array) ⇝ *object :* real
Multichannel output image.

▷ **TransMat** (input_control) ..... real-array ⇝ *real*
Transformation matrix for the color values.

Result

The operator `linear_trans_color` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

`gen_principal_comp_trans`

Possible Successors

`convert_image_type`

Alternatives

`principal_comp`, `trans_from_rgb`, `trans_to_rgb`

Module

Foundation

principal_comp (MultichannelImage : PCAImage : : InfoPerComp)

Compute the principal components of multichannel images.

`principal_comp` does a principal components analysis of multichannel images. This is useful for images obtained, e.g., with the thematic mapper of the Landsat satellite. Because the spectral bands are highly correlated, it is desirable to transform them to uncorrelated images. This can be used to save storage, since the bands containing little information can be discarded, and with respect to a later classification step.

The operator `principal_comp` takes a (multichannel) image `MultichannelImage` and transforms it to the output image `PCAImage`, which contains the same number of channels, using the principal components analysis. The parameter `InfoPerComp` contains the relative information content of each output channel.

Attention

`principal_comp` can be executed on OpenCL devices if image consists of eight channels or less. Since the calculations are done in single precision floating point, the results may differ from those calculated by the CPU.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **`MultichannelImage`** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Multichannel input image.

▷ **`PCAImage`** (output_object) ..... multichannel-image ⇝ *object :* real
Multichannel output image.

▷ **`InfoPerComp`** (output_control) ..... real-array ⇝ *real*
Information content of each output channel.

Result

The operator `principal_comp` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`gen_principal_comp_trans`

See also

`linear_trans_color`

Module

Foundation

`rgb1_to_gray` (RGBImage : GrayImage : :)

Transform an RGB image into a gray scale image.

`rgb1_to_gray` transforms an RGB image into a gray scale image. The three channels of the RGB image are passed as the first three channels of the input image. The image is transformed according to the following formula:

$k = 0.299r + 0.587g + 0.114b$.

If one of the input images in `RGBImage` is a single-channel image its reference will be simply copied to the output `GrayImage`.

Parameters

▷ **`RGBImage`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Three-channel RBG image.

▷ **`GrayImage`** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2 / real
Gray scale image.

Example

```
* Transformation from rgb to gray
read_image(Image,'patras')
dev_display(Image)
rgb1_to_gray(Image,GrayImage)
dev_display(GrayImage)
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

compose3

Alternatives

trans_from_rgb, rgb3_to_gray

Module

Foundation

rgb3_to_gray (ImageRed, ImageGreen, ImageBlue : ImageGray : :)

Transform an RGB image to a gray scale image.

rgb3_to_gray transforms an RGB image into a gray scale image. The three channels of the RGB image are passed as three separate images. The image is transformed according to the following formula:

$k = 0.299r + 0.587g + 0.114b$.

Parameters

- ▷ **ImageRed** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2 / real
  Input image (red channel).
- ▷ **ImageGreen** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2 / real
  Input image (green channel).
- ▷ **ImageBlue** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2 / real
  Input image (blue channel).
- ▷ **ImageGray** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2 / real
  Gray scale image.

Example

```
* Transformation from rgb to gray
read_image(Image,'patras')
dev_display(Image)
decompose3(Image, ImageR, ImageG, ImageB)
rgb3_to_gray(ImageR,ImageG,ImageB,GrayImage)
dev_display(GrayImage)
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

decompose3

Alternatives

rgb1_to_gray, trans_from_rgb

Module

Foundation

trans_from_rgb (ImageRed, ImageGreen, ImageBlue : ImageResult1, ImageResult2, ImageResult3 : ColorSpace :)

Transform an image from the RGB color space to an arbitrary color space.

trans_from_rgb transforms an image from the RGB color space to an arbitrary color space (ColorSpace). The three channels of the image are passed as three separate images on input and output.

The operator trans_from_rgb supports the image types byte, uint2, int4, and real. In the case of real images, all values should lay within 0 and 1. If not, the results of the transformation may not be reasonable.

Certain scalings are performed accordingly to the image type:

- Considering byte and uint2 images, the domain of color space values is generally mapped to the full domain of [0..255] or [0..65535], respectively. Because of this, the origin of signed values (e.g., CIELab) may not be at the center of the domain.
- Hue values are represented by angles of $[0..2\pi[$ and are coded for the particular image types differently:
  - byte-images map the angle domain to $[0..255]$.
  - uint2/int4-images are coded in minutes of arc $[0..21600[$, except for the transformations *'cielchab'* and *'cielchuv'* for int4-images, where they are coded in seconds of arc $[0..1296000[$.
  - real-images are coded in radians $[0..2\pi[$, except for the transformations *'cielchab'* and *'cielchuv'*, where the standards ISO 11664-4:2008 and ISO 11664-5:2009 require the hue to be specified in degrees.
- Saturation values are represented by percentages of $[0..100]$ and are coded differently for the particular image type:
  - byte-images map the saturation values to $[0..255]$.
  - uint2/int4 map the saturation values to $[0..10000]$.
  - real-images map the saturation values to $[0..1]$.

Supported are the transformations listed below. Note, all domains are based on RGB values scaled to [0; 1]. To obtain the domain of a certain image type, they must be scaled accordingly with the value range. Due to different precision the values obtained using the given equations may slightly differ from the values returned by the operator.

'yiq'

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ I &= -0.27(B-Y) + 0.74(B-Y) \\ Q &= 0.41(B-Y) + 0.48(R-Y) \end{aligned}$$

Value range:

$Y \in [0;1]$, $I \in [-0.59947; 0.59947]$, $Q \in [-0.52243; 0.52243]$

'yuv'

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ U &= 0.493(B-Y) \\ V &= 0.877(R-Y) \end{aligned}$$

Note, this implies that Y, U, and V are not independent of each other.

Value range:

$Y \in [0.0; 1.0]$, $U \in [-0.436798; 0.436798]$, $V \in [-0.614777; 0.614777]$

'argyb'

$$\begin{pmatrix} A \\ Rg \\ Yb \end{pmatrix} = \begin{pmatrix} 0.30 & 0.59 & 0.11 \\ 0.50 & -0.50 & 0.00 \\ 0.25 & 0.25 & -0.50 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Value range:

$A \in [0;1]$, $Rg \in [-0.5; 0.5]$, $Yb \in [-0.5; 0.5]$

'ciexyz'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

The primary colors used correspond to sRGB respectively CIE Rec. 709. D65 is used as white point.

Used primary colors (x, y, z)**:**

$$\text{red:=} \begin{pmatrix} 0.6400 \\ 0.3300 \\ 0.0300 \end{pmatrix}, \quad \text{green:=} \begin{pmatrix} 0.3000 \\ 0.6000 \\ 0.1000 \end{pmatrix}, \quad \text{blue:=} \begin{pmatrix} 0.1500 \\ 0.0600 \\ 0.7900 \end{pmatrix}, \quad \text{white}_{65}\text{:=} \begin{pmatrix} 0.3127 \\ 0.3290 \\ 0.3583 \end{pmatrix}$$

Value range:

$X \in [0; 0.950456]$, $Y \in [0; 1]$, $Z \in [0; 1.088754]$

'hls'

```
Min := min([R, G, B])
Max := max([R, G, B])
L := (Min + Max) / 2
if (Max == Min)
   H := 0
   S := 0
else
   if (L > 0.5)
      S := (Max - Min) / (2 - Max - Min)
   else
      S := (Max - Min) / (Max + Min)
   endif
   if (R == Max)
      H := ((G - B) / (Max - Min)) * rad(60)
   elseif (G == Max)
      H := (2 + (B - R) / (Max - Min)) * rad(60)
   elseif (B == Max)
      H := (4 + (R - G) / (Max - Min)) * rad(60)
   endif
endif
```

Value range:

$H \in [0; 2\pi[$, $L \in [0; 1]$, $S \in [0; 1]$

'hsi'

$$\begin{pmatrix} M1 \\ M2 \\ I1 \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{6}} & \frac{-1}{\sqrt{6}} & \frac{-1}{\sqrt{6}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{pmatrix} H \\ S \\ I \end{pmatrix} = \begin{pmatrix} \operatorname{atan2}(M2, M1) \\ \sqrt{M1^2 + M2^2} \\ \frac{I1}{\sqrt{3}} \end{pmatrix}$$

Value range:

$H \in [0; 2\pi[$, $\in [0; \sqrt{\frac{2}{3}}]$, $I \in [0; 1]$

'hsv'

```
Min := min([R, G, B])
Max := max([R, G, B])
V := Max
if (Max == Min)
   S := 0
```

```
   H := 0
else
   S := (Max - Min) / Max
   if (R == Max)
      H := ((G - B) / (Max - Min)) * rad(60)
   elseif (G == Max)
      H := (2 + (B - R) / (Max - Min)) * rad(60)
   elseif (B == Max)
      H := (4 + (R - G) / (Max - Min)) * rad(60)
   endif
endif
```

Value range:
$H \in [0; 2\pi[$, $S \in [0; 1]$, $V \in [0; 1]$

'ihs'

```
Min := min([R, G, B])
Max := max([R, G, B])
I := (R + G + B) / 3
if (I == 0)
   H := 0
   S := 1
else
   S := 1 - Min / I
   if (S == 0)
        H := 0
   else
      X := (R + R - G - B) / 2
      Y := (R - G) * (R - G) + (R - B) * (G - B)
      Z := sqrt(Y)
      if (Z == 0)
         H := 0
      else
         H := acos(X / Z)
      endif
      if (B > G)
         H := rad(360) - H
      endif
   endif
endif
```

Value range:
$I \in [0; 1]$, $H \in [0; 2\pi[$, $S \in [0; 1]$

'cielab'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{aligned} L &= 116 f(\tfrac{Y}{Y_w}) - 16 \\ a &= 500(f(\tfrac{X}{X_w}) - f(\tfrac{Y}{Y_w})) \\ b &= 200(f(\tfrac{Y}{Y_w}) - f(\tfrac{Z}{Z_w})) \end{aligned}$$

where

$$f(t) = \begin{cases} t^{\frac{1}{3}}, & \text{if } t > (\frac{6}{29})^3 \\ \frac{841}{108} * t + \frac{4}{29}, & \text{otherwise} \end{cases}$$

Black point B: (R_B, G_B, B_B) = (0, 0, 0)

White point W (according to image type):

- `byte`: (R_W, G_W, B_W) = (255, 255, 255)
- `uint2`: (R_W, G_W, B_W) = ($2^{16}-1$, $2^{16}-1$, $2^{16}-1$)
- `int4`: (R_W, G_W, B_W) = ($2^{31}-1$, $2^{31}-1$, $2^{31}-1$)
- `real`: (R_W, G_W, B_W) = (1.0, 1.0, 1.0)

Value range:
$L \in [0;100]$, $a \in [-86.1813;98.2352]$, $b \in [-107.8617;94.4758]$ (`byte` and `uint2`: scaled to the maximum gray value. `int4`: L is scaled to the maximum gray value, a and b are scaled to the minimum gray value, such that the origin stays at 0.)

'cielchab'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{aligned} L &= 116 f(\tfrac{Y}{Y_w}) - 16 \\ a &= 500(f(\tfrac{X}{X_w}) - f(\tfrac{Y}{Y_w})) \\ b &= 200(f(\tfrac{Y}{Y_w}) - f(\tfrac{Z}{Z_w})) \\ C_{ab} &= \sqrt{a^2+b^2} \\ h_{ab} &= \operatorname{atan2}(b,a) \end{aligned}$$

where

$$f(t) = \begin{cases} t^{\frac{1}{3}}, & \text{if } t > (\frac{6}{29})^3 \\ \frac{841}{108} * t + \frac{4}{29}, & \text{otherwise} \end{cases}$$

h_{ab} lies

- between 0° and 90° if a and b are both positive,
- between 90° and 180° if a is negative and b is positive,
- between 180° and 270° if a and b are both negative, and
- between 270° and 360° if a is positive and b is negative.

Black point B: (R_B, G_B, B_B) = (0, 0, 0)
White point W (according to image type):

- `byte`: (R_W, G_W, B_W) = (255, 255, 255)
- `uint2`: (R_W, G_W, B_W) = ($2^{16}-1$, $2^{16}-1$, $2^{16}-1$)
- `int4`: (R_W, G_W, B_W) = ($2^{31}-1$, $2^{31}-1$, $2^{31}-1$)
- `real`: (R_W, G_W, B_W) = (1.0, 1.0, 1.0)

Value range:
$L \in [0;100]$, $C \in [0;133.8086]$, $h \in [0;360[$ (`byte`: scaled to the maximum gray value. `uint2`: L and C are scaled to the maximum gray value, h is given in minutes of arc. `int4`: L and C are scaled to the maximum gray value, h is given in seconds of arc.)

'cieluv'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{aligned} u' &= 4X/(X+15Y+3Z) \\ v' &= 9Y/(X+15Y+3Z) \\ u'_w &= 4X_w/(X_w+15Y_w+3Z_w) \\ v'_w &= 9Y_w/(X_w+15Y_w+3Z_w) \\ L &= 116 f(\tfrac{Y}{Y_w}) - 16 \\ u &= 13L(u'-u'_w) \\ v &= 13L(v'-v'_w) \end{aligned}$$

where

$$f(t) = \begin{cases} t^{\frac{1}{3}}, & \text{if } t > (\frac{6}{29})^3 \\ \frac{841}{108} * t + \frac{4}{29} & \text{otherwise} \end{cases}$$

Black point B: (R_B, G_B, B_B) = (0, 0, 0)

White point W (according to image type):

- `byte`: (R_W, G_W, B_W) = (255, 255, 255)
- `uint2`: (R_W, G_W, B_W) = ($2^{16}-1$, $2^{16}-1$, $2^{16}-1$)
- `int4`: (R_W, G_W, B_W) = ($2^{31}-1$, $2^{31}-1$, $2^{31}-1$)
- `real`: (R_W, G_W, B_W) = (1.0, 1.0, 1.0)

Value range:

$L \in [0;100]$, $u \in [-83.0774;175.0148]$, $v \in [-134.1008;107.3923]$ (`byte` and `uint2`: scaled to the maximum gray value. `int4`: L is scaled to the maximum gray value, u and v are scaled to the minimum gray value, such that the origin stays at 0.)

'cielchuv'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{aligned} u' &= 4X/(X+15Y+3Z) \\ v' &= 9Y/(X+15Y+3Z) \\ u'_w &= 4X_w/(X_w+15Y_w+3Z_w) \\ v'_w &= 9Y_w/(X_w+15Y_w+3Z_w) \\ L &= 116f(\tfrac{Y}{Y_w}) - 16 \\ u &= 13L(u'-u'_w) \\ v &= 13L(v'-v'_w) \\ C_{uv} &= \sqrt{u^2+v^2} \\ h_{uv} &= \operatorname{atan2}(v,u) \end{aligned}$$

where

$$f(t) = \begin{cases} t^{\frac{1}{3}}, & \text{if } t > (\frac{6}{29})^3 \\ \frac{841}{108} * t + \frac{4}{29}, & \text{otherwise} \end{cases}$$

Black point B: (R_B, G_B, B_B) = (0, 0, 0)

White point W (according to image type):

- `byte`: (R_W, G_W, B_W) = (255, 255, 255)
- `uint2`: (R_W, G_W, B_W) = ($2^{16}-1$, $2^{16}-1$, $2^{16}-1$)
- `int4`: (R_W, G_W, B_W) = ($2^{31}-1$, $2^{31}-1$, $2^{31}-1$)
- `real`: (R_W, G_W, B_W) = (1.0, 1.0, 1.0)

Value range:

$L \in [0;100]$, $C \in [0;179.0402]$, $h \in [0;360[$ (`byte`: scaled to the maximum gray value. `uint2`: L and C are scaled to the maximum gray value, h is given in minutes of arc. `int4`: L and C are scaled to the maximum gray value, h is given in seconds of arc.)

'i1i2i3'

$$\begin{pmatrix} I1 \\ I2 \\ I3 \end{pmatrix} = \begin{pmatrix} 0.333 & 0.333 & 0.333 \\ 1.0 & 0.0 & -1.0 \\ -0.5 & 1.0 & -0.5 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Value range:

$I1 \in [0;1]$, $I2 \in [-1;1]$, $I3 \in [-1;1]$

'ciexyz2'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.639175 & 0.175258 & 0.185567 \\ 0.306931 & 0.584158 & 0.108911 \\ 0.000000 & 0.060773 & 0.939227 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Value range:
$X \in [0;1.0]$, $Y \in [0;1.0]$, $Z \in [0;1.0]$

'ciexyz3'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.618 & 0.177 & 0.205 \\ 0.299 & 0.587 & 0.114 \\ 0.000 & 0.056 & 0.944 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Value range:
$X \in [0;1]$, $Y \in [0;1]$, $Z \in [0;1]$

'ciexyz4'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.476 & 0.299 & 0.175 \\ 0.262 & 0.656 & 0.082 \\ 0.020 & 0.161 & 0.909 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Used primary colors (x, y, z)**:**

$$\text{red:=} \begin{pmatrix} 0.628 \\ 0.346 \\ 0.026 \end{pmatrix}, \quad \text{green:=} \begin{pmatrix} 0.268 \\ 0.588 \\ 0.144 \end{pmatrix}, \text{blue:=} \begin{pmatrix} 0.150 \\ 0.070 \\ 0.780 \end{pmatrix}, \quad \text{white}_{65}\text{:=} \begin{pmatrix} 0.313 \\ 0.329 \\ 0.358 \end{pmatrix}$$

Value range:
$X \in [0;0.951]$, $Y \in [0;1]$, $Z \in [0;1.088]$

'lms'

$$\begin{pmatrix} L \\ M \\ S \end{pmatrix} = \begin{pmatrix} 0.390469 & 0.549910 & 0.008901 \\ 0.070925 & 0.963118 & 0.001358 \\ 0.023143 & 0.128013 & 0.935976 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

This conceptually is a transformation from RGB to CIE XYZ (see *'ciexyz'* above) followed by a transformation from CIE XYZ to LMS.

Value range:
$L \in [0;0.949280]$, $M \in [0;1.035401]$, $S \in [0;1.087132]$

Attention

As the calculations are made with a different numerical precision, the OpenCL implementation of the cielab transformation for images of type `int4` is slightly less accurate than the pure C version.

Parameters

- ▷ **ImageRed** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Input image (red channel).
- ▷ **ImageGreen** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Input image (green channel).
- ▷ **ImageBlue** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Input image (blue channel).
- ▷ **ImageResult1** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Color-transformed output image (channel 1).
- ▷ **ImageResult2** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Color-transformed output image (channel 1).
- ▷ **ImageResult3** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Color-transformed output image (channel 1).
- ▷ **ColorSpace** (input_control) . . . . . string ⇝ *string*
  Color space of the output image.
  Default: 'hsv'
  List of values: `ColorSpace` ∈ {'cielab', 'cielchab', 'cieluv', 'cielchuv', 'hsv', 'hsi', 'yiq', 'yuv', 'argyb', 'ciexyz', 'ciexyz2', 'ciexyz3', 'ciexyz4', 'hls', 'ihs', 'i1i2i3', 'lms'}

Example

```
* Transformation from rgb to hsv and conversely
read_image(Image,'patras')
dev_display(Image)
decompose3(Image, Image1, Image2, Image3)
trans_from_rgb(Image1,Image2,Image3,ImageH,ImageS,ImageV,'hsv')
trans_to_rgb(ImageH,ImageS,ImageV,ImageR,ImageG,ImageB,'hsv')
compose3(ImageR,ImageG,ImageB,Multichannel)
dev_display(Multichannel)
```

Result

`trans_from_rgb` returns 2 (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

`decompose3`

Possible Successors

`compose3`

Alternatives

`linear_trans_color`, `rgb1_to_gray`, `rgb3_to_gray`

See also

`trans_to_rgb`

References

ITU-R BT.470-6: “Conventional Television Systems”, 1998.

ISO 11664-4:2008: “Colorimetry — Part 4: CIE 1976 L*a*b* Colour Space”, 2008.

ISO 11664-5:2009: “Colorimetry — Part 5: CIE 1976 L*u*v* Colour space and u’,v’ uniform chromaticity scale diagram”, 2009.

Module

Foundation

trans_to_rgb (ImageInput1, ImageInput2, ImageInput3 : ImageRed, ImageGreen, ImageBlue : ColorSpace :)

Transform an image from an arbitrary color space to the RGB color space.

`trans_to_rgb` transforms an image from an arbitrary color space (`ColorSpace`) to the RGB color space. The three channels of the image are passed as three separate images on input and output.

The operator `trans_to_rgb` supports the image types `byte`, `uint2`, `int4`, and `real`. The domain of the input images must match the domain provided by a corresponding transformation with `trans_from_rgb`. If not, the results of the transformation may not be reasonable.

This includes some scalings in the case of certain image types and transformations:

- Considering `byte` and `uint2` images, the domain of color space values is expected to be spread to the full domain of [0..255] or [0..65535], respectively. This includes a shift in the case of signed values, such that the origin of signed values (e.g., CIELab) may not be at the center of the domain.

- Hue values are represented by angles of $[0..2\pi[$ and are coded for the particular image types differently:
  - `byte`-images map the angle domain on $[0..255]$.
  - `uint2`/`int4`-images are coded in minutes of arc $[0..21600[$, except for the transformations *'cielchab'* and *'cielchuv'* for `int4`-images, where they are coded in seconds of arc $[0..1296000[$.
  - `real`-images are coded in radians $[0..2\pi[$, except for the transformations *'cielchab'* and *'cielchuv'*, where the standards ISO 11664-4:2008 and ISO 11664-5:2009 require the hue to be specified in degrees.
- Saturation values are represented by percentages of $[0..100]$ and are coded for the particular image type differently:
  - `byte`-images map the saturation values to $[0..255]$.
  - `uint2`/`int4`-images map the saturation values to $[0..10000]$.
  - `real`-images map the saturation values to $[0..1]$.

Supported are the transformations listed below. Note, all domains are based on RGB values scaled to [0; 1]. To obtain the domain of a certain image type, they must be scaled accordingly with the value range. Due to different precision the values obtained using the given equations may slightly differ from the values returned by the operator.

'yiq'

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0.946882217 & 0.623556582 \\ 1.0 & -0.267024956 & -0.649523352 \\ 1.0 & -1.108545035 & 1.709006928 \end{pmatrix} \begin{pmatrix} Y \\ I \\ Q \end{pmatrix}$$

Value range:
$Y \in [0;1]$, $I \in [-0.59947; 0.59947]$, $Q \in [-0.52243; 0.52243]$

'yuv'

$$\begin{aligned} R &= Y + \tfrac{1.0}{0.877} V \\ G &= Y - \tfrac{0.114}{0.493*0.587} U - \tfrac{0.299}{0.877*0.587} V \\ B &= Y + \tfrac{1.0}{0.493} U \end{aligned}$$

Note, this implies that Y, U, and V are not independent of each other.

Value range:
$Y \in [0.0; 1.0]$ $U \in [-0.436798; 0.436798]$ $V \in [-0.614777; 0.614777]$

'argyb'

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.00 & 1.29 & 0.22 \\ 1.00 & -0.71 & 0.22 \\ 1.00 & 0.29 & -1.78 \end{pmatrix} \begin{pmatrix} A \\ Rg \\ Yb \end{pmatrix}$$

Value range:
$A \in [0;1]$, $Rg \in [-0.5; 0.5]$, $Yb \in [-0.5; 0.5]$

'ciexyz'

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

Value range:
$X \in [0; 0.950456]$ $Y \in [0;1]$ $Z \in [0; 1.088754]$

'cielab'

$$
\begin{aligned}
f_y &= (L+16)/116 \\
f_x &= a/500 + f_y \\
f_z &= b/200 - f_y
\end{aligned}
$$

$$
X = \begin{cases} X_w * f_x^3, & \text{if } f_x > \frac{6}{29} \\ (f_x - \frac{4}{29}) * X_w * \frac{108}{841}, & \text{otherwise} \end{cases}
$$

$$
Y = \begin{cases} Y_w * f_y^3, & \text{if } f_y > \frac{6}{29} \\ (f_y - \frac{4}{29}) * Y_w * \frac{108}{841}, & \text{otherwise} \end{cases}
$$

$$
Z = \begin{cases} Z_w * f_z^3, & \text{if } f_z > \frac{6}{29} \\ (f_z - \frac{4}{29}) * Z_w * \frac{108}{841}, & \text{otherwise} \end{cases}
$$

$$
\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}
$$

Black point B: (R_B, G_B, B_B) = (0, 0, 0)

White point W (according to image type):

- `byte`: (R_W, G_W, B_W) = (255, 255, 255)
- `uint2`: (R_W, G_W, B_W) = $(2^{16}-1, 2^{16}-1, 2^{16}-1)$
- `int4`: (R_W, G_W, B_W) = $(2^{31}-1, 2^{31}-1, 2^{31}-1)$
- `real`: (R_W, G_W, B_W) = (1.0, 1.0, 1.0)

Value range:

$L \in [0;100]$, $a \in [-86.1813;98.2352]$, $b \in [-107.8617;94.4758]$ (`byte` and `uint2`: scaled to the maximum gray value. `int4`: L is scaled to the maximum gray value, a and b is scaled to the minimum gray value, such that the origin stays at 0.)

'cielchab'

$$
\begin{aligned}
a &= C_{ab} \cos h_{ab} \\
b &= C_{ab} \sin h_{ab} \\
f_y &= (L+16)/116 \\
f_x &= a/500 + f_y \\
f_z &= b/200 - f_y
\end{aligned}
$$

$$
X = \begin{cases} X_w * f_x^3, & \text{if } f_x > \frac{6}{29} \\ (f_x - \frac{4}{29}) * X_w * \frac{108}{841}, & \text{otherwise} \end{cases}
$$

$$
Y = \begin{cases} Y_w * f_y^3, & \text{if } f_y > \frac{6}{29} \\ (f_y - \frac{4}{29}) * Y_w * \frac{108}{841}, & \text{otherwise} \end{cases}
$$

$$
Z = \begin{cases} Z_w * f_z^3, & \text{if } f_z > \frac{6}{29} \\ (f_z - \frac{4}{29}) * Z_w * \frac{108}{841}, & \text{otherwise} \end{cases}
$$

$$
\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}
$$

Black point B: (R_B, G_B, B_B) = (0, 0, 0)

White point W (according to image type):

- `byte`: (R_W, G_W, B_W) = (255, 255, 255)
- `uint2`: (R_W, G_W, B_W) = $(2^{16}-1, 2^{16}-1, 2^{16}-1)$
- `int4`: (R_W, G_W, B_W) = $(2^{31}-1, 2^{31}-1, 2^{31}-1)$
- `real`: (R_W, G_W, B_W) = (1.0, 1.0, 1.0)

Value range:

$L \in [0;100]$, $C \in [0;133.8086]$, $h \in [0;360[$ (`byte`: scaled to the maximum gray value. `uint2`: L and C are scaled to the maximum gray value, h is given in minutes of arc. `int4`: L and C are scaled to the maximum gray value, h is given in seconds of arc.)

'cieluv'

$$
\begin{array}{lcl}
f_y & = & (L+16)/116 \\
Y & = & \begin{cases} Y_w * f_y^3, & \text{if } f_y > \frac{6}{29} \\ (f_y - \frac{4}{29}) * Y_w * \frac{108}{841}, & \text{otherwise} \end{cases} \\
u' & = & \frac{u}{13L} + u'_w \\
v' & = & \frac{v}{13L} + v'_w \\
x & = & \frac{9u'}{6u'-16v'+12} \\
y & = & \frac{4v'}{6u'-16v'+12} \\
X & = & \frac{xY}{y} \\
Z & = & (1-x-y)\frac{Y}{y}
\end{array}
$$

$$
\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}
$$

Black point B: (R_B, G_B, B_B) = (0, 0, 0)

White point W (according to image type):

- `byte`: (R_W, G_W, B_W) = (255, 255, 255)
- `uint2`: (R_W, G_W, B_W) = ($2^{16}-1$, $2^{16}-1$, $2^{16}-1$)
- `int4`: (R_W, G_W, B_W) = ($2^{31}-1$, $2^{31}-1$, $2^{31}-1$)
- `real`: (R_W, G_W, B_W) = (1.0, 1.0, 1.0)

Value range:

$L \in [0;100]$, $u \in [-83.0774;175.0148]$, $v \in [-134.1008;107.3923]$ (`byte` and `uint2`: scaled to the maximum gray value. `int4`: L are scaled to the maximum gray value, u and v are scaled to the minimum gray value, such that the origin stays at 0.)

'cielchuv'

$$
\begin{array}{lcl}
u & = & C_{uv} \cos h_{uv} \\
v & = & C_{uv} \sin h_{uv} \\
f_y & = & (L+16)/116 \\
Y & = & \begin{cases} Y_w * f_y^3, & \text{if } f_y > \frac{6}{29} \\ (f_y - \frac{4}{29}) * Y_w * \frac{108}{841}, & \text{otherwise} \end{cases} \\
u' & = & \frac{u}{13L} + u'_w \\
v' & = & \frac{u}{13L} + v'_w \\
x & = & \frac{9u'}{6u'-16v'+12} \\
y & = & \frac{4v'}{6u'-16v'+12} \\
X & = & \frac{xY}{y} \\
Z & = & (1-x-y)\frac{Y}{y}
\end{array}
$$

$$
\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}
$$

Black point B: (R_B, G_B, B_B) = (0, 0, 0)

White point W (according to image type):

- `byte`: (R_W, G_W, B_W) = (255, 255, 255)
- `uint2`: (R_W, G_W, B_W) = ($2^{16}-1$, $2^{16}-1$, $2^{16}-1$)
- `int4`: (R_W, G_W, B_W) = ($2^{31}-1$, $2^{31}-1$, $2^{31}-1$)
- `real`: (R_W, G_W, B_W) = (1.0, 1.0, 1.0)

Value range:

$L \in [0;100][$, $C \in [0;179.0402]$, $h \in [0;360[$ (`byte`: scaled to the maximum gray value. `uint2`: L and C are scaled to the maximum gray value, h is given in minutes of arc. `int4`: L and C are scaled to the maximum gray value, h is given in seconds of arc.)

'hls'

```
Hi := floor(H/rad(60))
Hf := H/rad(60) - Hi
if (L <= 0.5)
   Max := L * (S + 1)
else
   Max := L + S - (L * S)
endif
Min := 2 * L - Max
if (S == 0)
   R := L
   G := L
   B := L
else
  if (Hi == 0)
     R := Max
     G := Min + Hf * (Max - Min)
     B := Min
  elseif (Hi == 1)
     R := Min + (1 - Hf) * (Max - Min)
     G := Max
     B := Min
  elseif (Hi == 2)
     R := Min
     G := Max
     B := Min + Hf * (Max - Min)
  elseif (Hi == 3)
     R := Min
     G := Min + (1 - Hf) * (Max - Min)
     B := Max
  elseif (Hi == 4)
     R := Min + Hf * (Max - Min)
     G := Min
     B := Max
  elseif (Hi == 5)
     R := Max
     G := Min
     B := Min + (1 - Hf) * (Max - Min)
  endif
endif
```

Value range:
$H \in [0; 2\pi[$, $L \in [0; 1]$, $S \in [0; 1]$

'hsi'

$$\begin{aligned} M1 &= S * \cos H \\ M2 &= S * \sin H \\ I1 &= I * \sqrt{3} \end{aligned}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{6}} & 0 & \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{6}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} M1 \\ M2 \\ I1 \end{pmatrix}$$

Value range:
$H \in [0; 2\pi[$, $S \in [0; \sqrt{\frac{2}{3}}]$, $I \in [0; 1]$

'hsv'

```
if (S == 0)
   R := V
   G := V
   B := V
else
   Hi := floor(H/rad(60))
   Hf := H/rad(60) - Hi
   if (Hi == 0)
      R := V
      G := V * (1 - (S * (1 - Hf)))
      B := V * (1 - S)
   elseif (Hi == 1)
      R := V * (1 - (S * Hf))
      G := V
      B := V * (1 - S)
   elseif (Hi == 2)
      R := V * (1 - S)
      G := V
      B := V * (1 - (S * (1 - Hf)))
   elseif (Hi == 3)
      R := V * (1 - S)
      G := V * (1 - (S * Hf))
      B := V
   elseif (Hi == 4)
      R := V * (1 - (S * (1 - Hf)))
      G := V * (1 - S)
      B := V
   elseif (Hi == 5)
      R := V
      G := V * (1 - S)
      B := V * (1 - (S * Hf))
   endif
endif
```

Value range:
$H \in [0; 2\pi[$, $S \in [0; 1]$, $V \in [0; 1]$

'ciexyz4'

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 2.750 & -1.149 & -0.426 \\ -1.118 & 2.026 & 0.033 \\ 0.138 & -0.333 & 1.104 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

Value range:
$X \in [0; 0.951]$, $Y \in [0; 1]$, $Z \in [0; 1.088]$

'lms'

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 2.858334 & -1.628721 & -0.024820 \\ -0.210432 & 1.158402 & 0.000321 \\ -0.041894 & -0.118163 & 1.068973 \end{pmatrix} \begin{pmatrix} L \\ M \\ S \end{pmatrix}$$

Value range:
$L \in [0; 0.949280]$, $M \in [0; 1.035401]$, $S \in [0; 1.087132]$

Attention

As the calculations are made with a different numerical precision, the OpenCL implementation of the cielab transformation for images of type `int4` is slightly less accurate than the pure C version.

Parameters

- ▷ **ImageInput1** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Input image (channel 1).
- ▷ **ImageInput2** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Input image (channel 2).
- ▷ **ImageInput3** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Input image (channel 3).
- ▷ **ImageRed** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Red channel.
- ▷ **ImageGreen** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Green channel.
- ▷ **ImageBlue** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
  Blue channel.
- ▷ **ColorSpace** (input_control) . . . . . string ⇝ *string*
  Color space of the input image.
  Default: 'hsv'
  List of values: `ColorSpace` ∈ {'hsi', 'yiq', 'yuv', 'argyb', 'ciexyz', 'ciexyz4', 'cielab', 'cielchab', 'cieluv', 'cielchuv', 'hls', 'hsv', 'lms'}

Example

```
* Transformation from rgb to hsv and conversely
read_image(Image,'patras')
dev_display(Image)
decompose3(Image, Image1, Image2, Image3)
trans_from_rgb(Image1,Image2,Image3,ImageH,ImageS,ImageV,'hsv')
trans_to_rgb(ImageH,ImageS,ImageV,ImageR,ImageG,ImageB,'hsv')
compose3(ImageR,ImageG,ImageB,Multichannel)
dev_display(Multichannel)
```

Result

`trans_to_rgb` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system(::'no_object_result', <Result>:)`. If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

`decompose3`

Possible Successors

`compose3`, `disp_color`

Alternatives

`linear_trans_color`

See also

`decompose3`, `trans_from_rgb`

References

ITU-R BT.470-6: "Conventional Television Systems", 1998.

ISO 11664-4:2008: "Colorimetry — Part 4: CIE 1976 L*a*b* Colour space", 2008.

ISO 11664-5:2009: "Colorimetry — Part 5: CIE 1976 L*u*v* Colour space and u',v' uniform chromaticity scale diagram", 2009.

Module

Foundation

12.4 Edges

close_edges (Edges, EdgeImage : RegionResult : MinAmplitude :)

Close edge gaps using the edge amplitude image.

close_edges closes gaps in the output of an edge detector, and thus tries to produce complete object contours. This is done by examining the neighbors of each edge point to determine the point with maximum amplitude (i.e., maximum gradient), and adding the point to the edge if its amplitude is larger than the minimum amplitude passed in MinAmplitude. This operator expects as input the edges (Edges) and amplitude image (EdgeImage) returned by typical edge operators, such as edges_image or sobel_amp. close_edges does not take into account the edge directions that may be returned by an edge operator. Thus, in areas where the gradient is almost constant the edges may become rather "wiggly."

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Edges** (input_object) . . . . . region(-array) ⇝ *object*
  Region containing one pixel thick edges.
- ▷ **EdgeImage** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2 / int4
  Edge amplitude (gradient) image.
- ▷ **RegionResult** (output_object) . . . . . region(-array) ⇝ *object*
  Region containing closed edges.
- ▷ **MinAmplitude** (input_control) . . . . . integer ⇝ *integer*
  Minimum edge amplitude.
  Default: 16
  Suggested values: MinAmplitude ∈ {5, 8, 10, 12, 16, 20, 25, 30, 40, 50}
  Value range: 1 ≤ MinAmplitude
  Minimum increment: 1
  Recommended increment: 1

Example

```
sobel_amp(Image,&EdgeAmp,"sum_abs",5);
threshold(EdgeAmp,&EdgeRegion,40.0,255.0);
skeleton(EdgeRegion,&ThinEdge);
close_edges(ThinEdge,EdgeAmp,&CloseEdges,15);
skeleton(CloseEdges,&ThinCloseEdges);
```

Result

close_edges returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

edges_image, sobel_amp, threshold, skeleton

Possible Successors

skeleton

Alternatives

close_edges_length, dilation1, closing

See also

gray_skeleton

Module

Foundation

close_edges_length (Edges, Gradient : ClosedEdges : MinAmplitude, MaxGapLength :)

Close edge gaps using the edge amplitude image.

close_edges_length closes gaps in the output of an edge detector, and thus tries to produce complete object contours. This operator expects as input the edges (Edges) and amplitude image (Gradient) returned by typical edge operators, such as edges_image or sobel_amp.

Contours are closed in two steps: First, one pixel wide gaps in the input contours are closed, and isolated points are eliminated. After this, open contours are extended by up to MaxGapLength points by adding edge points until either the contour is closed or no more significant edge points can be found. A gradient is regarded as significant if it is larger than MinAmplitude. The neighboring points examined as possible new edge points are the point in the direction of the contour and its two adjacent points in an 8-neighborhood. For each of these points, the sum of its gradient and the maximum gradient of that points three possible neighbors is calculated (*look ahead* of length 1). The point with the maximum sum is then chosen as the new edge point.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Edges** (input_object) ..... region(-array) ⇝ *object*
Region containing one pixel thick edges.

▷ **Gradient** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
Edge amplitude (gradient) image.

▷ **ClosedEdges** (output_object) ..... region(-array) ⇝ *object*
Region containing closed edges.

▷ **MinAmplitude** (input_control) ..... integer ⇝ *integer*
Minimum edge amplitude.
Default: 16
Suggested values: MinAmplitude ∈ {5, 8, 10, 12, 16, 20, 25, 30, 40, 50}
Value range: 0 ≤ MinAmplitude ≤ 255
Minimum increment: 1
Recommended increment: 1

▷ **MaxGapLength** (input_control) ..... integer ⇝ *integer*
Maximal number of points by which edges are extended.
Default: 3
Suggested values: MaxGapLength ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20, 30, 40, 50, 70, 100}
Value range: 1 ≤ MaxGapLength ≤ 127
Minimum increment: 1
Recommended increment: 1

Example

```
sobel_amp(Image,&EdgeAmp,"sum_abs",5);
threshold(EdgeAmp,&EdgeRegion,40.0,255.0);
skeleton(EdgeRegion,&ThinEdge);
close_edges_length(ThinEdge,EdgeAmp,&CloseEdges,15,3);
```

Result

close_edges_length returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

edges_image, sobel_amp, threshold, skeleton

Alternatives

close_edges, dilation1, closing

References

M. Üsbeck: "Untersuchungen zur echtzeitfähigen Segmentierung"; Studienarbeit, Bayerisches Forschungszentrum für Wissensbasierte Systeme (FORWISS), Erlangen, 1993.

Module

Foundation

derivate_gauss (Image : DerivGauss : Sigma, Component :)

Convolve an image with derivatives of the Gaussian.

derivate_gauss convolves an image with the derivatives of a Gaussian and calculates various features derived therefrom. Sigma is the parameter of the Gaussian (i.e., the amount of smoothing). If one value is passed in Sigma the amount of smoothing in the column and row direction is identical. If two values are passed in Sigma the first value specifies the amount of smoothing in the column direction, while the second value specifies the amount of smoothing in the row direction. The possible values for Component are:

'none': Smoothing only.

'x': First derivative along x.

$$g'(x,y) = \frac{\partial g(x,y)}{\partial x}$$

'y': First derivative along y.

$$g'(x,y) = \frac{\partial g(x,y)}{\partial y}$$

'gradient': Absolute value of the gradient.

$$g'(x,y) = \sqrt{\frac{\partial g(x,y)^2}{\partial x}\frac{\partial g(x,y)^2}{\partial y}}$$

'gradient_dir': Gradient direction in radians.

$$\phi = \text{atan2}(\frac{\partial g(x,y)}{\partial y}, \frac{\partial g(x,y)}{\partial x})$$

'xx': Second derivative along x.

$$g'(x,y) = \frac{\partial^2 g(x,y)}{\partial x^2}$$

'yy': Second derivative along y.

$$g'(x,y) = \frac{\partial^2 g(x,y)}{\partial y^2}$$

'xy': Second derivative along x and y.

$$g'(x,y) = \frac{\partial^2 g(x,y)}{\partial x \partial y}$$

'xxx': Third derivative along x.

$$g'(x,y) = \frac{\partial^3 g(x,y)}{\partial x^3}$$

'yyy': Third derivative along y.

$$g'(x,y) = \frac{\partial^3 g(x,y)}{\partial y^3}$$

'xxy': Third derivative along x, x and y.

$$g'(x,y) = \frac{\partial^3 g(x,y)}{\partial x^2 \partial y}$$

'xyy': Third derivative along x, y and y.

$$g'(x,y) = \frac{\partial^3 g(x,y)}{\partial x \partial y^2}$$

'det': Determinant of the Hessian matrix:

$$DET = \frac{\partial^2 g(x,y)}{\partial x^2}\frac{\partial^2 g(x,y)}{\partial y^2} - \left(\frac{\partial^2 g(x,y)}{\partial y \partial x}\right)^2$$

'laplace': Laplace operator (trace of the Hessian matrix):

$$TR = \frac{\partial^2 g(x,y)}{\partial x^2} + \frac{\partial^2 g(x,y)}{\partial y^2}$$

'mean_curvature': Mean curvature H

$$\begin{aligned} a &= (1 + \frac{\partial g(x,y)^2}{\partial x})\frac{\partial^2 g(x,y)}{\partial y^2} \\ b &= 2\frac{\partial g(x,y)}{\partial x}\frac{\partial g(x,y)}{\partial y}\frac{\partial^2 g(x,y)}{\partial y \partial x} \\ c &= (1 + \frac{\partial g(x,y)^2}{\partial y})\frac{\partial^2 g(x,y)}{\partial x^2} \\ d &= (1 + \frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y})^{\frac{3}{2}} \\ H &= \frac{a - b + c}{d} \\ H &= \frac{1}{2}(\kappa_{min} + \kappa_{max}) \end{aligned}$$

'gauss_curvature': Gaussian curvature K

$$K = \frac{DET}{(1 + \frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y})^2}$$

'area': Differential Area A

$$\begin{aligned} A &= EG - F^2 \\ E &= 1 + \frac{\partial g(x,y)}{\partial x}^2 \\ F &= \frac{\partial g(x,y)}{\partial x}\frac{\partial g(x,y)}{\partial y} \\ G &= 1 + \frac{\partial g(x,y)}{\partial y}^2 \end{aligned}$$

'eigenvalue1': First eigenvalue

$$\begin{aligned} a &= \frac{\frac{\partial^2 g(x,y)}{\partial x^2} + \frac{\partial^2 g(x,y)}{\partial y^2}}{2} \\ \lambda_1 &= a + \sqrt{a^2 - (\frac{\partial^2 g(x,y)}{\partial x^2}\frac{\partial^2 g(x,y)}{\partial y^2} - \frac{\partial^2 g(x,y)^2}{\partial y \partial x})} \end{aligned}$$

'eigenvalue2': Second eigenvalue

$$\begin{aligned} a &= \frac{\frac{\partial^2 g(x,y)}{\partial x^2} + \frac{\partial^2 g(x,y)}{\partial y^2}}{2} \\ \lambda_2 &= a - \sqrt{a^2 - (\frac{\partial^2 g(x,y)}{\partial x^2}\frac{\partial^2 g(x,y)}{\partial y^2} - \frac{\partial^2 g(x,y)^2}{\partial y \partial x})} \end{aligned}$$

'eigenvec_dir': Direction of the eigenvector corresponding to the first eigenvalue in radians

'main1_curvature' First main curvature

$$\kappa_{max} = H + \sqrt{H^2 - K}$$

'main2_curvature' Second main curvature

$$\kappa_{min} = H - \sqrt{H^2 - K}$$

'kitchen_rosenfeld': Second derivative perpendicular to the gradient

$$k = \frac{\frac{\partial^2 g(x,y)}{\partial x^2}\frac{\partial g(x,y)^2}{\partial y} + \frac{\partial^2 g(x,y)}{\partial y^2}\frac{\partial g(x,y)^2}{\partial x} - 2\frac{\partial^2 g(x,y)}{\partial y \partial x}\frac{\partial g(x,y)^2}{\partial x}\frac{\partial g(x,y)^2}{\partial y}}{\frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y}}$$

'zuniga_haralick': Normalized second derivative perpendicular to the gradient

$$k = \frac{\frac{\partial^2 g(x,y)}{\partial x^2}\frac{\partial g(x,y)^2}{\partial y} + \frac{\partial^2 g(x,y)}{\partial y^2}\frac{\partial g(x,y)^2}{\partial x} - 2\frac{\partial^2 g(x,y)}{\partial y \partial x}\frac{\partial g(x,y)^2}{\partial x}\frac{\partial g(x,y)^2}{\partial y}}{\left(\frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y}\right)^{\frac{3}{2}}}$$

'2nd_ddg': Second derivative along the gradient

$$k = \frac{\frac{\partial^2 g(x,y)}{\partial x^2}\frac{\partial g(x,y)^2}{\partial x} + 2\frac{\partial g(x,y)}{\partial x}\frac{\partial g(x,y)}{\partial y}\frac{\partial^2 g(x,y)}{\partial y \partial x} + \frac{\partial^2 g(x,y)}{\partial y^2}\frac{\partial g(x,y)^2}{\partial y}}{\frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y}}$$

'de_saint_venant': Second derivative along and perpendicular to the gradient

$$k = \frac{\frac{\partial g(x,y)}{\partial x}\frac{\partial g(x,y)}{\partial y}\left(\frac{\partial^2 g(x,y)}{\partial x^2} - \frac{\partial^2 g(x,y)}{\partial y^2}\right) - \left(\frac{\partial g(x,y)^2}{\partial x} - \frac{\partial g(x,y)^2}{\partial y}\right)\frac{\partial^2 g(x,y)}{\partial x \partial y}}{\frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y}}$$

Attention

Besides the pure C version there are specific implementations of `derivate_gauss` for speed up. Such an optimization is applied in case it is supported by the system and the respective system parameter `*_enable` is set to *'true'*, see `set_system`. The following optimizations are supported (listed according to their priority):

- using *AVX512f* instructions (*'avx512f_enable'*)
- using *AVX* instructions (*'avx_enable'*)
- using *SSE2* instructions (*'sse2_enable'*)

These implementations are slightly inaccurate compared to the pure C version due to numerical issues. For example, using SSE2 instructions the inaccuracy is in order of magnitude of $1.0e-5$ for 'byte' images and `Component` set to *'none'*, *'x'*, or *'y'*.

In case accuracy is preferred over performance, set all corresponding system parameter to *'false'* (using `set_system`) before calling `derivate_gauss`. This way `derivate_gauss` does not use the accelerations. Do not forget to set the parameter back to *'true'* afterwards.

`derivate_gauss` is only executed on an OpenCL device if `Sigma` induces a filter width respectively height of up to 129 pixels. This corresponds to a `Sigma` of less than 20.7 for `Component` = *'none'*. The OpenCL implementation is slightly inaccurate compared to the pure C version due to numerical issues.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real

Input images.

▷ **`DerivGauss`** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* real

Filtered result images.

▷ **Sigma** (input_control) . . . . . real(-array) ⇝ *real*
Sigma of the Gaussian.
Default: 1.0
Suggested values: Sigma ∈ {0.7, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0}
Value range: 0.01 ≤ Sigma ≤ 50.0
Minimum increment: 0.01
Recommended increment: 0.1

▷ **Component** (input_control) . . . . . string ⇝ *string*
Derivative or feature to be calculated.
Default: 'x'
List of values: Component ∈ {'none', 'x', 'y', 'gradient', 'xx', 'yy', 'xy', 'xxx', 'yyy', 'xxy', 'xyy', 'det', 'mean_curvature', 'gauss_curvature', 'eigenvalue1', 'eigenvalue2', 'main1_curvature', 'main2_curvature', 'kitchen_rosenfeld', 'zuniga_haralick', '2nd_ddg', 'de_saint_venant', 'area', 'laplace', 'gradient_dir', 'eigenvec_dir'}

Example

```
read_image(&Image,"mreut");
derivate_gauss(Image,&Gauss,3.0,"x");
zero_crossing(Gauss,&ZeroCrossings);
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

zero_crossing, dual_threshold

Alternatives

laplace, laplace_of_gauss, binomial_filter, gauss_filter, smooth_image, isotropic_diffusion

See also

zero_crossing, dual_threshold

Module

Foundation

diff_of_gauss (Image : DiffOfGauss : Sigma, SigFactor :)

Approximate the LoG operator (Laplace of Gaussian).

diff_of_gauss approximates the Laplace-of-Gauss operator by a difference of Gaussians. The standard deviations of these Gaussians can be calculated, according to Marr, from the Parameter Sigma of the LoG and the ratio of the two standard deviations (SigFactor) as:

$$
\begin{aligned}
sigma1 &= \frac{\mathtt{Sigma}}{\sqrt{-2\frac{\log\left(\frac{1}{\mathtt{SigFactor}}\right)}{\mathtt{SigFactor}^2-1}}} \\
sigma2 &= \frac{sigma1}{\mathtt{SigFactor}} \\
\mathtt{DiffOfGauss} &= \mathtt{Image} * gauss(sigma1) - \mathtt{Image} * gauss(sigma2)
\end{aligned}
$$

For a SigFactor = 1.6, according to Marr, an approximation to the Mexican-Hat-Operator results. The resulting image is stored in DiffOfGauss.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2
  Input image
- ▷ **DiffOfGauss** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* int2
  LoG image.
- ▷ **Sigma** (input_control) ..... real ⇝ *real*
  Smoothing parameter of the Laplace operator to approximate.
  Default: 3.0
  Suggested values: Sigma ∈ {2.0, 3.0, 4.0, 5.0}
  Minimum increment: 0.01
  Recommended increment: 0.1
  Restriction: Sigma > 0.0
- ▷ **SigFactor** (input_control) ..... real ⇝ *real*
  Ratio of the standard deviations used (Marr recommends 1.6).
  Default: 1.6
  Minimum increment: 0.01
  Recommended increment: 0.1
  Restriction: SigFactor > 0.0

Example

```
read_image(Image,'fabrik')
diff_of_gauss(Image,Laplace,2.0,1.6)
zero_crossing(Laplace,ZeroCrossings)
```

Complexity

The execution time depends linearly on the number of pixels and the size of sigma.

Result

diff_of_gauss returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

zero_crossing, dual_threshold

Alternatives

laplace, derivate_gauss

References

D. Marr: "Vision (A computational investigation into human representation and processing of visual information)"; New York, W.H. Freeman and Company; 1982.

Module

Foundation

edges_color (Image : ImaAmp, ImaDir : Filter, Alpha, NMS, Low, High :)

Extract color edges using Canny, Deriche, or Shen filters.

edges_color extracts color edges from the input image Image. To define color edges, the multi-channel image Image is regarded as a mapping $f : R^2 \mapsto R^n$, where n is the number of channels in Image. For such functions, there is a natural extension of the gradient: the metric tensor G, which can be used to calculate for every direction, given by the direction vector v, the rate of change of f in the direction v. For notational convenience, G will be regarded as a two-dimensional matrix. Thus, the rate of change of the function f in the direction v is given by $v^T G v$, where

$$G = \begin{pmatrix} f_x^T f_x & f_x^T f_y \\ f_x^T f_y & f_y^T f_y \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} \frac{\partial f_i}{\partial x}\frac{\partial f_i}{\partial x} & \sum_{i=1}^{n} \frac{\partial f_i}{\partial x}\frac{\partial f_i}{\partial y} \\ \sum_{i=1}^{n} \frac{\partial f_i}{\partial x}\frac{\partial f_i}{\partial y} & \sum_{i=1}^{n} \frac{\partial f_i}{\partial y}\frac{\partial f_i}{\partial y} \end{pmatrix} .$$

The partial derivatives of the images, which are necessary to calculate the metric tensor, are calculated with the corresponding edge filters, analogously to edges_image. For Filter = *'canny'*, the partial derivatives of the Gaussian smoothing masks are used (see derivate_gauss), for *'deriche1'* and Filter = *'deriche2'* the corresponding Deriche filters, for Filter = *'shen'* the corresponding Shen filters, and for Filter = *'sobel_fast'* the Sobel filter. Analogously to single-channel images, the gradient direction is defined by the vector v in which the rate of change f is maximum. The vector v is given by the eigenvector corresponding to the largest eigenvalue of G. The square root of the eigenvalue is the equivalent of the gradient magnitude (the amplitude) for single-channel images, and is returned in ImaAmp. For single-channel images, both definitions are equivalent. Since the gradient magnitude may be larger than what can be represented in the input image data type (byte or uint2), it is stored in the next larger data type (uint2 or int4) in ImaAmp. The eigenvector also is used to define the edge direction. In contrast to single-channel images, the edge direction can only be defined modulo 180 degrees. Like in the output of edges_image, the edge directions are stored in 2-degree steps, and are returned in ImaDir. Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction). For speed reasons, the edge directions are not computed explicitly for Filter = *'sobel_fast'*. Therefore, ImaDir is an empty object in this case.

The "filter width" (i.e., the amount of smoothing) can be chosen arbitrarily for all filters except *'sobel_fast'* (where the filter width is 3×3 and Alpha is ignored), and can be estimated by calling info_edges for concrete values of the parameter Alpha. It decreases for increasing Alpha for the Deriche and Shen filters and increases for the Canny filter, where it is the standard deviation of the Gaussian on which the Canny operator is based. "Wide" filters exhibit a larger invariance to noise, but also a decreased ability to detect small details. Non-recursive filters, such as the Canny filter, are realized using filter masks, and thus the execution time increases for increasing filter width. In contrast, the execution time for recursive filters does not depend on the filter width. Thus, arbitrary filter widths are possible using the Deriche and Shen filters without increasing the run time of the operator. The resulting advantage in speed compared to the Canny operator naturally increases for larger filter widths. As border treatment, the recursive operators assume that the images are zero outside of the image, while the Canny operator mirrors the gray value at the image border. The signal-noise-ratio of the filters is comparable for the following choices of Alpha:

$$\begin{aligned} \texttt{Alpha}('deriche2') &= \texttt{Alpha}('deriche1')/2 \\ \texttt{Alpha}('shen') &= \texttt{Alpha}('deriche1')/2 \\ \texttt{Alpha}('canny') &= 1.77/\texttt{Alpha}('deriche1') \end{aligned}$$

edges_color optionally offers to apply a non-maximum-suppression (NMS = *'nms'*/*'inms'*/*'hvnms'*; *'none'* if not desired) and hysteresis threshold operation (Low,High; at least one negative if not desired) to the resulting edge image. Conceptually, this corresponds to the following calls:

```
nonmax_suppression_dir(...,NMS,...)
hysteresis_threshold(...,Low,High,1000,...).
```

Note that the hysteresis threshold operation is not applied if NMS is set to *'none'*.

For *'sobel_fast'*, the same non-maximum-suppression is performed for all values of NMS except *'none'*. Additionally, for *'sobel_fast'* the resulting edges are thinned to a width of one pixel.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Input image.

▷ **ImaAmp** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* uint2 / int4
Edge amplitude (gradient magnitude) image.

▷ **ImaDir** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* direction
Edge direction image.

▷ **Filter** (input_control) . . . . . string ⇝ *string*
Edge operator to be applied.
Default: 'canny'
List of values: `Filter` ∈ {'canny', 'deriche1', 'deriche2', 'shen', 'sobel_fast'}

▷ **Alpha** (input_control) . . . . . real ⇝ *real*
Filter parameter: small values result in strong smoothing, and thus less detail (opposite for 'canny').
Default: 1.0
Suggested values: `Alpha` ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9, 1.0, 1.1, 1.2, 1.5, 2.0, 2.5, 3.0}
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Alpha > 0.0

▷ **NMS** (input_control) . . . . . string ⇝ *string*
Non-maximum suppression ('none', if not desired).
Default: 'nms'
List of values: `NMS` ∈ {'nms', 'inms', 'hvnms', 'none'}

▷ **Low** (input_control) . . . . . integer ⇝ *integer*
Lower threshold for the hysteresis threshold operation (negative if no thresholding is desired).
Default: 20
Suggested values: `Low` ∈ {5, 10, 15, 20, 25, 30, 40}
Minimum increment: 1
Recommended increment: 5
Restriction: Low >= 1 || Low < 0

▷ **High** (input_control) . . . . . integer ⇝ *integer*
Upper threshold for the hysteresis threshold operation (negative if no thresholding is desired).
Default: 40
Suggested values: `High` ∈ {10, 15, 20, 25, 30, 40, 50, 60, 70}
Minimum increment: 1
Recommended increment: 5
Restriction: High >= 1 || High < 0 && High >= Low

Result

`edges_color` returns 2 (`H_MSG_TRUE`) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Successors

`threshold`

Alternatives

`edges_color_sub_pix`

See also

edges_image, edges_sub_pix, info_edges, nonmax_suppression_amp, hysteresis_threshold

References

C. Steger: "Subpixel-Precise Extraction of Lines and Edges"; International Archives of Photogrammetry and Remote Sensing, vol. XXXIII, part B3; pp. 141-156; 2000.

C. Steger: "Unbiased Extraction of Curvilinear Structures from 2D and 3D Images"; Herbert Utz Verlag, München; 1998.

S. Di Zenzo: "A Note on the Gradient of a Multi-Image"; Computer Vision, Graphics, and Image Processing, vol. 33; pp. 116-125; 1986.

Aldo Cumani: "Edge Detection in Multispectral Images"; Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing, vol. 53, no. 1; pp. 40-51; 1991.

J.Canny: "Finding Edges and Lines in Images"; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge; 1983.

J.Canny: "A Computational Approach to Edge Detection"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-8, vol. 6; pp. 679-698; 1986.

R.Deriche: "Using Canny's Criteria to Derive a Recursively Implemented Optimal Edge Detector"; International Journal of Computer Vision; vol. 1, no. 2; pp. 167-187; 1987.

R.Deriche: "Fast Algorithms for Low-Level Vision"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-12, no. 1; pp. 78-87; 1990.

J. Shen, S. Castan: "An Optimal Linear Operator for Step Edge Detection"; Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing, vol. 54, no. 2; pp. 112-133; 1992.

Module

Foundation

edges_color_sub_pix (Image : Edges : Filter, Alpha, Low, High :)

Extract subpixel precise color edges using Deriche, Shen, or Canny filters.

edges_color_sub_pix extracts subpixel precise color edges from the input image Image. The definition of color edges is given in the description of edges_color. The same edge filters as in edges_color can be selected: *'canny'*, *'deriche1'*, *'deriche2'*, and *'shen'*. In addition, a fast Sobel filter can be selected with *'sobel_fast'*. The filters are specified by the parameter Filter.

The "filter width" (i.e., the amount of smoothing) can be chosen arbitrarily. For a detailed description of this parameter see edges_color. This parameter is ignored for Filter = *'sobel_fast'*.

The extracted edges are returned as subpixel precise XLD contours in Edges. For all edge operators except for *'sobel_fast'*, the following attributes are defined for each edge point (see get_contour_attrib_xld for further information):

- *'edge_direction'*: Gives the direction of the edge (not of the XLD contour), calculated from the image gradients in horizontal and vertical direction. The angles [rad] are given with respect to the column axis of the image.
- *'angle'*: Direction of the normal vectors to the contour in radians (oriented such that the normal vectors point to the right side of the contour as the contour is traversed from start to end point; the angles are given with respect to the row axis of the image).
- *'response'*: Edge amplitude (gradient magnitude).

edges_color_sub_pix links the edge points into edges by using an algorithm similar to a hysteresis threshold operation, which is also used in edges_sub_pix and lines_gauss. Points with an amplitude larger than High are immediately accepted as belonging to an edge, while points with an amplitude smaller than Low are rejected. All other points are accepted as edges if they are connected to accepted edge points (see also lines_gauss and hysteresis_threshold).

Because edge extractors are often unable to extract certain junctions, a mode that tries to extract these missing junctions by different means can be selected by appending *'_junctions'* to the values of `Filter` that are described above. This mode is analogous to the mode for completing junctions that is available in `edges_sub_pix` and `lines_gauss`.

The edge operator *'sobel_fast'* has the same semantics as all the other edge operators. Internally, however, it is based on significantly simplified variants of the individual processing steps (hysteresis thresholding, edge point linking, and extraction of the subpixel edge positions). Therefore, *'sobel_fast'* in some cases may return slightly less accurate edge positions and may select different edge parts.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **`Image`** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image.

▷ **`Edges`** (output_object) . . . . . xld_cont-array ⇝ *object*
Extracted edges.

▷ **`Filter`** (input_control) . . . . . string ⇝ *string*
Edge operator to be applied.
Default: 'canny'
List of values: `Filter` ∈ {'canny', 'deriche1', 'deriche2', 'shen', 'sobel_fast', 'canny_junctions', 'deriche1_junctions', 'deriche2_junctions', 'shen_junctions'}

▷ **`Alpha`** (input_control) . . . . . real ⇝ *real*
Filter parameter: small values result in strong smoothing, and thus less detail (opposite for 'canny').
Default: 1.0
Suggested values: `Alpha` ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9, 1.0, 1.1, 1.2, 1.5, 2.0, 2.5, 3.0}
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Alpha > 0.0

▷ **`Low`** (input_control) . . . . . number ⇝ *real* / integer
Lower threshold for the hysteresis threshold operation.
Default: 20
Suggested values: `Low` ∈ {5, 10, 15, 20, 25, 30, 40}
Value range: 0 ≤ Low
Minimum increment: 1
Recommended increment: 5
Restriction: Low > 0

▷ **`High`** (input_control) . . . . . number ⇝ *real* / integer
Upper threshold for the hysteresis threshold operation.
Default: 40
Suggested values: `High` ∈ {10, 15, 20, 25, 30, 40, 50, 60, 70}
Value range: 0 ≤ High
Minimum increment: 1
Recommended increment: 5
Restriction: High > 0 && High >= Low

Complexity

The amount of temporary memory required is dependent on the height H of the domain of `Image`.

Result

`edges_color_sub_pix` returns `2` (`H_MSG_TRUE`) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

edges_color

See also

edges_image, edges_sub_pix, info_edges, hysteresis_threshold, lines_gauss, lines_facet

References

C. Steger: "Subpixel-Precise Extraction of Lines and Edges"; International Archives of Photogrammetry and Remote Sensing, vol. XXXIII, part B3; pp. 141-156; 2000.

C. Steger: "Unbiased Extraction of Curvilinear Structures from 2D and 3D Images"; Herbert Utz Verlag, München; 1998.

S. Di Zenzo: "A Note on the Gradient of a Multi-Image"; Computer Vision, Graphics, and Image Processing, vol. 33; pp. 116-125; 1986.

Aldo Cumani: "Edge Detection in Multispectral Images"; Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing, vol. 53, no. 1; pp. 40-51; 1991.

J.Canny: "Finding Edges and Lines in Images"; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge; 1983.

J.Canny: "A Computational Approach to Edge Detection"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-8, vol. 6; pp. 679-698; 1986.

R.Deriche: "Using Canny's Criteria to Derive a Recursively Implemented Optimal Edge Detector"; International Journal of Computer Vision; vol. 1, no. 2; pp. 167-187; 1987.

R.Deriche: "Fast Algorithms for Low-Level Vision"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-12, no. 1; pp. 78-87; 1990.

J. Shen, S. Castan: "An Optimal Linear Operator for Step Edge Detection"; Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing, vol. 54, no. 2; pp. 112-133; 1992.

Module

2D Metrology

edges_image (Image : ImaAmp, ImaDir : Filter, Alpha, NMS, Low, High :)

Extract edges using Deriche, Lanser, Shen, or Canny filters.

edges_image detects step edges using recursively implemented filters (according to Deriche, Lanser and Shen) or the conventionally implemented "derivative of Gaussian" filter (using filter masks) proposed by Canny. Furthermore, a very fast variant of the Sobel filter can be used. Thus, the following edge operators are available:

'deriche1', *'lanser1'*, *'deriche1_int4'*, *'deriche2'*, *'lanser2'*, *'deriche2_int4'*, *'shen'*, *'mshen'*, *'canny'*, and *'sobel_fast'*

(parameter Filter).

The edge amplitudes (gradient magnitude) are returned in ImaAmp.

For all filters except *'sobel_fast'*, the edge directions are returned in ImaDir. For *'sobel_fast'*, the edge direction is not computed to speed up the filter. Consequently, ImaDir is an empty image object. The edge operators *'deriche1'* respectively *'deriche2'* are also available for int4-images, and return the signed filter response instead of its absolute value. This behavior can be obtained for byte-images as well by selecting *'deriche1_int4'* respectively *'deriche2_int4'* as filter. This can be used to calculate the second derivative of an image by applying edges_image (with parameter value *'lanser2'*) to the signed first derivative. Edge directions are stored in 2-degree steps, i.e., an edge direction of x degrees in mathematically positive sense and with respect to the horizontal axis is stored as $x/2$ in the edge direction image. Furthermore, the direction of the change of intensity is taken into account. Let $[E_x, E_y]$ denote the image gradient. Then the following edge directions are returned as $r/2$:

intensity increase	E_x/E_y	
edge direction r		
from bottom to top	$0/+$	0
from lower right to upper left	$-/+$	$]0, 90[$
from right to left	$-/0$	90
from upper right to lower left	$-/-$	$]90, 180[$
from top to bottom	$0/-$	180
from upper left to lower right	$+/-$	$]180, 270[$
from left to right	$+/0$	270
from lower left to upper right	$+/+$	$]270, 360[$

Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction).

The "filter width" (i.e., the amount of smoothing) can be chosen arbitrarily for all filters except *'sobel_fast'* (where the filter width is 3×3 and `Alpha` is ignored), and can be estimated by calling `info_edges` for concrete values of the parameter `Alpha`. It decreases for increasing `Alpha` for the Deriche, Lanser and Shen filters and increases for the Canny filter, where it is the standard deviation of the Gaussian on which the Canny operator is based. "Wide" filters exhibit a larger invariance to noise, but also a decreased ability to detect small details. Non-recursive filters, such as the Canny filter, are realized using filter masks, and thus the execution time increases for increasing filter width. In contrast, the execution time for recursive filters does not depend on the filter width. Thus, arbitrary filter widths are possible using the Deriche, Lanser and Shen filters without increasing the run time of the operator. The resulting advantage in speed compared to the Canny operator naturally increases for larger filter widths. As border treatment, the recursive operators assume that the images to be zero outside of the image, while the Canny operator repeats the gray value at the image's border. The signal-noise-ratio of the filters is comparable for the following choices of `Alpha`:

$$\begin{aligned}
\texttt{Alpha}('lanser1') &= \texttt{Alpha}('deriche1') \\
\texttt{Alpha}('deriche2') &= \texttt{Alpha}('deriche1')/2 \\
\texttt{Alpha}('lanser2') &= \texttt{Alpha}('deriche2') \\
\texttt{Alpha}('shen') &= \texttt{Alpha}('deriche1')/2 \\
\texttt{Alpha}('mshen') &= \texttt{Alpha}('shen') \\
\texttt{Alpha}('canny') &= 1.77/\texttt{Alpha}('deriche1')
\end{aligned}$$

The originally proposed recursive filters (*'deriche1'*, *'deriche2'*, *'shen'*) return a biased estimate of the amplitude of diagonal edges. This bias is removed in the corresponding modified version of the operators (*'lanser1'*, *'lanser2'* and *'mshen'*), while maintaining the same execution speed.

For relatively small filter widths (11×11), i.e., for `Alpha`('lanser2') = *0.5*, all filters yield similar results. Only for "wider" filters differences begin to appear: the Shen filters begin to yield qualitatively inferior results. However, they are the fastest of the implemented operators — closely followed by the Deriche operators.

`edges_image` optionally offers to apply a non-maximum-suppression (`NMS` = *'nms'*/*'inms'*/*'hvnms'*; *'none'* if not desired) and hysteresis threshold operation (`Low`,`High`; at least one negative if not desired) to the resulting edge image. Conceptually, this corresponds to the following calls:

```
nonmax_suppression_dir(...,NMS,...)
hysteresis_threshold(...,Low,High,999,...).
```

Note that the hysteresis threshold operation is not applied if `NMS` is set to *'none'*.

For *'sobel_fast'*, the same non-maximum-suppression is performed for all values of `NMS` except *'none'*. Additionally, for *'sobel_fast'* the resulting edges are thinned to a width of one pixel.

`edges_image` can be executed on OpenCL devices for the filter types *'canny'* and *'sobel_fast'*.

Attention

The OpenCL implementation of `edges_image` will generally compute results that differ somewhat from the CPU implementation.

Since `edges_image` uses Gauss convolution internally for the *'canny'* filter, the same limitations for OpenCL apply as for `derivate_gauss`: `Alpha` must be chosen small enough that the required filter mask is less than 129 pixels in size.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
Input image.

▷ **ImaAmp** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2 / int4 / real
Edge amplitude (gradient magnitude) image.

▷ **ImaDir** (output_object) . . . . . image(-array) ⇝ *object :* direction
Edge direction image.

▷ **Filter** (input_control) . . . . . string ⇝ *string*
Edge operator to be applied.
Default: 'canny'
List of values: `Filter` ∈ {'deriche1', 'deriche1_int4', 'deriche2', 'deriche2_int4', 'lanser1', 'lanser2', 'shen', 'mshen', 'canny', 'sobel_fast'}

▷ **Alpha** (input_control) . . . . . real ⇝ *real*
Filter parameter: small values result in strong smoothing, and thus less detail (opposite for 'canny').
Default: 1.0
Suggested values: `Alpha` ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9, 1.1}
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Alpha > 0.0

▷ **NMS** (input_control) . . . . . string ⇝ *string*
Non-maximum suppression ('none', if not desired).
Default: 'nms'
List of values: `NMS` ∈ {'nms', 'inms', 'hvnms', 'none'}

▷ **Low** (input_control) . . . . . integer ⇝ *integer* / real
Lower threshold for the hysteresis threshold operation (negative, if no thresholding is desired).
Default: 20
Suggested values: `Low` ∈ {5, 10, 15, 20, 25, 30, 40}
Minimum increment: 1
Recommended increment: 5
Restriction: Low != 0

▷ **High** (input_control) . . . . . integer ⇝ *integer* / real
Upper threshold for the hysteresis threshold operation (negative, if no thresholding is desired).
Default: 40
Suggested values: `High` ∈ {10, 15, 20, 25, 30, 40, 50, 60, 70}
Minimum increment: 1
Recommended increment: 5
Restriction: High >= Low

Example

```
read_image(Image,'fabrik')
edges_image(Image,Amp,Dir,'lanser2',0.5,'none',-1,-1)
hysteresis_threshold(Amp,Margin,20,30,30)
```

Result

`edges_image` returns 2 (`H_MSG_TRUE`) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

- Automatically parallelized on internal data level.

Possible Predecessors

info_edges

Possible Successors

threshold, hysteresis_threshold, close_edges_length

Alternatives

sobel_dir, frei_dir, kirsch_dir, prewitt_dir, robinson_dir

See also

info_edges, nonmax_suppression_amp, hysteresis_threshold, bandpass_image

References

S.Lanser, W.Eckstein: "Eine Modifikation des Deriche-Verfahrens zur Kantendetektion"; 13. DAGM-Symposium, München; Informatik Fachberichte 290; Seite 151 - 158; Springer-Verlag; 1991.

S.Lanser: "Detektion von Stufenkanten mittels rekursiver Filter nach Deriche"; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

J.Canny: "Finding Edges and Lines in Images"; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge; 1983.

J.Canny: "A Computational Approach to Edge Detection"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-8, vol. 6; S. 679-698; 1986.

R.Deriche: "Using Canny's Criteria to Derive a Recursively Implemented Optimal Edge Detector"; International Journal of Computer Vision; vol. 1, no. 2; S. 167-187; 1987.

R.Deriche: "Optimal Edge Detection Using Recursive Filtering"; Proc. of the First International Conference on Computer Vision, London; S. 501-505; 1987.

R.Deriche: "Fast Algorithms for Low-Level Vision"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-12, no. 1; S. 78-87; 1990.

S.Castan, J.Zhao und J.Shen: "Optimal Filter for Edge Detection Methods and Results"; Proc. of the First European Conference on Computer Vision, Antibes; Lecture Notes on computer Science; no. 427; S. 12-17; Springer-Verlag; 1990.

Module

Foundation

edges_sub_pix (Image : Edges : Filter, Alpha, Low, High :)

Extract sub-pixel precise edges using Deriche, Lanser, Shen, or Canny filters.

edges_sub_pix detects step edges using recursively implemented filters (according to Deriche, Lanser and Shen) or the conventionally implemented "derivative of Gaussian" filter (using filter masks) proposed by Canny. Thus, the following edge operators are available for Filter:

'deriche1', *'lanser1'*, *'deriche2'*, *'lanser2'*, *'shen'*, *'mshen'*, *'canny'*, *'sobel'* und *'sobel_fast'*.

The extracted edges are returned as sub-pixel precise XLD contours in Edges. For all edge operators except *'sobel_fast'*, the following attributes are defined for each edge point (see get_contour_attrib_xld for further details):

'edge_direction': Gives the direction of the edge (not of the XLD contour), calculated from the image gradients in horizontal and vertical direction. The angles [rad] are given with respect to the column axis of the image.

'angle': Direction of the normal vectors to the contour in radians (oriented such that the normal vectors point to the right side of the contour as the contour is traversed from start to end point; the angles are given with respect to the row axis of the image).

'response': Edge amplitude (gradient magnitude).

The "filter width" (i.e., the amount of smoothing) can be chosen arbitrarily for all edge operators except *'sobel'* and *'sobel_fast'*, and can be estimated by calling info_edges for concrete values of the parameter Alpha. For all filters (Deriche, Lanser and Shen filters), the "filter width" decreases for increasing Alpha. The only exception

is the Canny filter, where an increasing Alpha also causes an increase of the "filter width". "Wide" filters exhibit a larger invariance to noise, but also a decreased ability to detect small details. Non-recursive filters, such as the Canny filter, are realized using filter masks, and thus the execution time increases for increasing filter width. In contrast, the execution time for recursive filters does not depend on the filter width. Thus, arbitrary filter widths are possible using the Deriche, Lanser and Shen filters without increasing the run time of the operator. The resulting advantage in speed compared to the Canny operator naturally increases for larger filter widths. As border treatment, the recursive operators assume that the images to be zero outside of the image, while the Canny operator repeats the gray value at the image's border. The signal-noise-ratio of the filters is comparable for the following choices of Alpha:

$$\begin{aligned}
\texttt{Alpha}('lanser1') &= \texttt{Alpha}('deriche1') \\
\texttt{Alpha}('deriche2') &= \texttt{Alpha}('deriche1')/2 \\
\texttt{Alpha}('lanser2') &= \texttt{Alpha}('deriche2') \\
\texttt{Alpha}('shen') &= \texttt{Alpha}('deriche1')/2 \\
\texttt{Alpha}('mshen') &= \texttt{Alpha}('shen') \\
\texttt{Alpha}('canny') &= 1.77/\texttt{Alpha}('deriche1')
\end{aligned}$$

The originally proposed recursive filters (*'deriche1'*, *'deriche2'*, *'shen'*) return a biased estimate of the amplitude of diagonal edges. This bias is removed in the corresponding modified version of the operators (*'lanser1'*, *'lanser2'* and *'mshen'*), while maintaining the same execution speed.

For relatively small filter widths (11×11), i.e., for Alpha (*'lanser2'* = *0.5*), all filters yield similar results. Only for "wider" filters differences begin to appear: the Shen filters begin to yield qualitatively inferior results. However, they are the fastest of the implemented operators that support arbitrary mask sizes, closely followed by the Deriche operators. The two Sobel filters, which use a fixed mask size of (3×3), are faster than the other filters. Of these two, the filter *'sobel_fast'* is significantly faster than *'sobel'*.

edges_sub_pix links the edge points into edges by using an algorithm similar to a hysteresis threshold operation, which is also used in lines_gauss. Points with an amplitude larger than High are immediately accepted as belonging to an edge, while points with an amplitude smaller than Low are rejected. All other points are accepted as edges if they are connected to accepted edge points (see also lines_gauss and hysteresis_threshold).

Because edge extractors are often unable to extract certain junctions, a mode that tries to extract these missing junctions by different means can be selected by appending *'_junctions'* to the values of Filter that are described above. This mode is analogous to the mode for completing junctions that is available in lines_gauss.

The edge operator *'sobel_fast'* has the same semantics as all the other edge operators. Internally, however, it is based on significantly simplified variants of the individual processing steps (hysteresis thresholding, edge point linking, and extraction of the subpixel edge positions). Therefore, *'sobel_fast'* in some cases may return slightly less accurate edge positions and may select different edge parts.

edges_sub_pix can be executed on OpenCL devices for the filter types *'canny'* and *'sobel_fast'*. This will require up to width*height*29 bytes of pinned memory. Since allocating memory is an expensive operation, it would make sense to set the pinned memory cache to at least this size (using set_compute_device_param for parameter *'pinned_mem_cache_capacity'*, or to disable pinned memory completely (using set_compute_device_param for parameter *'alloc_pinned'*), in which case the normal memory cache is used. Note that the results can vary from the CPU implementation.

Attention

Since edges_sub_pix uses Gauss convolution internally for the *'canny'* filter, the same limitations for OpenCL apply as for derivate_gauss: Alpha must be chosen small enough that the required filter mask is less than 129 pixels in size. Also, edges_sub_pix is not available on OpenCL devices for HALCON XL, as double precision floating point arithmetic would be required, but OpenCL devices are optimized for single precision arithmetic.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **Edges** (output_object) ..... xld_cont-array ⇝ *object*
  Extracted edges.

▷ **Filter** (input_control) . . . . . string ⇝ *string*
Edge operator to be applied.
Default: 'canny'
List of values: Filter ∈ {'deriche1', 'lanser1', 'deriche2', 'lanser2', 'shen', 'mshen', 'canny', 'sobel', 'sobel_fast', 'deriche1_junctions', 'lanser1_junctions', 'deriche2_junctions', 'lanser2_junctions', 'shen_junctions', 'mshen_junctions', 'canny_junctions', 'sobel_junctions'}

▷ **Alpha** (input_control) . . . . . real ⇝ *real*
Filter parameter: small values result in strong smoothing, and thus less detail (opposite for 'canny').
Default: 1.0
Suggested values: Alpha ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9, 1.1}
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Alpha > 0.0

▷ **Low** (input_control) . . . . . integer ⇝ *integer* / real
Lower threshold for the hysteresis threshold operation.
Default: 20
Suggested values: Low ∈ {5, 10, 15, 20, 25, 30, 40}
Minimum increment: 1
Recommended increment: 5
Restriction: Low > 0

▷ **High** (input_control) . . . . . integer ⇝ *integer* / real
Upper threshold for the hysteresis threshold operation.
Default: 40
Suggested values: High ∈ {10, 15, 20, 25, 30, 40, 50, 60, 70}
Minimum increment: 1
Recommended increment: 5
Restriction: High > 0 && High >= Low

Example

```
read_image(Image,'fabrik')
edges_sub_pix(Image,Edges,'lanser2',0.5,20,40)
```

Complexity

Let A be the number of pixels in the domain of Image. Then the runtime complexity is $O(A * \text{Alpha})$ for the Canny filter and $O(A)$ for the recursive Lanser, Deriche, and Shen filters.

The amount of temporary memory required is dependent on the height H of the domain of Image and the width W of Image. Let $S = W * H$, then edges_sub_pix requires at least $60 * S$ bytes of temporary memory during execution for all edge operators except *'sobel_fast'*. For *'sobel_fast'*, at least $9 * S$ bytes of temporary memory are required.

Result

edges_sub_pix returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

segment_contours_xld, gen_polygons_xld, select_shape_xld

Alternatives

sobel_dir, frei_dir, kirsch_dir, prewitt_dir, robinson_dir, edges_image

See also

info_edges, hysteresis_threshold, bandpass_image, lines_gauss, lines_facet

References

S.Lanser, W.Eckstein: “Eine Modifikation des Deriche-Verfahrens zur Kantendetektion”; 13. DAGM-Symposium, München; Informatik Fachberichte 290; Seite 151 - 158; Springer-Verlag; 1991.

S.Lanser: “Detektion von Stufenkanten mittels rekursiver Filter nach Deriche”; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

J.Canny: “Finding Edges and Lines in Images”; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge; 1983.

J.Canny: “A Computational Approach to Edge Detection”; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-8, vol. 6; S. 679-698; 1986.

R.Deriche: “Using Canny’s Criteria to Derive a Recursively Implemented Optimal Edge Detector”; International Journal of Computer Vision; vol. 1, no. 2; S. 167-187; 1987.

R.Deriche: “Optimal Edge Detection Using Recursive Filtering”; Proc. of the First International Conference on Computer Vision, London; S. 501-505; 1987.

R.Deriche: “Fast Algorithms for Low-Level Vision”; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-12, no. 1; S. 78-87; 1990.

S.Castan, J.Zhao und J.Shen: “Optimal Filter for Edge Detection Methods and Results”; Proc. of the First European Conference on Computer Vision, Antibes; Lecture Notes on computer Science; no. 427; S. 12-17; Springer-Verlag; 1990.

Module

2D Metrology

frei_amp (Image : ImageEdgeAmp : :)

Detect edges (amplitude) using the Frei-Chen operator.

frei_amp calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{matrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{matrix}$$

$$B = \begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix}$$

The result image contains the maximum response of the masks A and B.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Input image.
- ▷ **ImageEdgeAmp** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Edge amplitude (gradient magnitude) image.

Example

```
read_image(Image,'fabrik')
frei_amp(Image,Frei_amp)
threshold(Frei_amp,Edges,128,255)
```

Result

frei_amp always returns 2 (H_MSG_TRUE). If the input is empty the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

binomial_filter, gauss_filter, sigma_image, median_image, smooth_image

Alternatives

sobel_amp, kirsch_amp, prewitt_amp, robinson_amp, roberts

See also

bandpass_image, laplace_of_gauss

Module

Foundation

frei_dir (Image : ImageEdgeAmp, ImageEdgeDir : :)

Detect edges (amplitude and direction) using the Frei-Chen operator.

frei_dir calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{matrix} 1 & \sqrt{2} & 1 \\ 0 & 0 & 0 \\ -1 & -\sqrt{2} & -1 \end{matrix}$$

$$B = \begin{matrix} 1 & 0 & -1 \\ \sqrt{2} & 0 & -\sqrt{2} \\ 1 & 0 & -1 \end{matrix}$$

The result image contains the maximum response of the masks A and B. The edge directions are returned in ImageEdgeDir, and are stored in 2-degree steps, i.e., an edge direction of x degrees in mathematically positive sense and with respect to the horizontal axis is stored as $x/2$ in the edge direction image. Furthermore, the direction of the change of intensity is taken into account. Let $[E_x, E_y]$ denote the image gradient. Then the following edge directions are returned as $r/2$:

intensity increase	E_x/E_y	
edge direction r		
from bottom to top	$0/+$	0
from lower right to upper left	$-/+$	$]0, 90[$
from right to left	$-/0$	90
from upper right to lower left	$-/-$	$]90, 180[$
from top to bottom	$0/-$	180
from upper left to lower right	$+/-$	$]180, 270[$
from left to right	$+/0$	270
from lower left to upper right	$+/+$	$]270, 360[$

Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction).

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Input image.

▷ **ImageEdgeAmp** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Edge amplitude (gradient magnitude) image.

▷ **ImageEdgeDir** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* direction
Edge direction image.

Example

```
read_image(Image,'fabrik')
frei_dir(Image,Frei_dirA,Frei_dirD)
threshold(Frei_dirA,Res,128,255)
```

Result

`frei_dir` always returns `2` (`H_MSG_TRUE`). If the input is empty the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

`binomial_filter`, `gauss_filter`, `sigma_image`, `median_image`, `smooth_image`

Possible Successors

`hysteresis_threshold`, `threshold`, `gray_skeleton`, `nonmax_suppression_dir`, `close_edges`, `close_edges_length`

Alternatives

`edges_image`, `sobel_dir`, `robinson_dir`, `prewitt_dir`, `kirsch_dir`

See also

`bandpass_image`, `laplace_of_gauss`

Module

Foundation

highpass_image (Image : Highpass : Width, Height :)

Extract high frequency components from an image.

`highpass_image` extracts high frequency components in an image by applying a linear filter with the following matrix (in case of a 7×5 matrix):

$$\begin{matrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -35 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$$

This corresponds to applying a mean operator (`mean_image`), and then subtracting the original gray value. For byte images, a value of 128 is added to the result, i.e., zero crossings occur for 128. Correspondingly for uint2 images, 32767 is added.

This filter emphasizes high frequency components (edges and corners). The cutoff frequency is determined by the size (Height × Width) of the filter matrix: the larger the matrix, the smaller the cutoff frequency is.

At the image borders the pixels' gray values are mirrored. In case of over- or underflow the gray values are clipped (255 and 0, resp.).

Attention

If even values are passed for Height or Width, the operator uses the next larger odd value instead. Thus, the center of the filter mask is always uniquely determined.

highpass_image can be executed on OpenCL devices. The same limitations as for mean_image and sub_image apply.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Input image.

▷ **Highpass** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
High-pass-filtered result image.

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the filter mask.
Default: 9
Suggested values: Width ∈ {3, 5, 7, 9, 11, 13, 17, 21, 29, 41, 51, 73, 101}
Value range: 1 ≤ Width
Minimum increment: 2
Recommended increment: 2

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the filter mask.
Default: 9
Suggested values: Height ∈ {3, 5, 7, 9, 11, 13, 17, 21, 29, 41, 51, 73, 101}
Value range: 1 ≤ Height
Minimum increment: 2
Recommended increment: 2

Example

```
highpass_image(Image,&Highpass,7,5);
threshold(Highpass,&Region,60.0,255.0);
skeleton(Region,&Skeleton);
```

Result

highpass_image returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

threshold, skeleton

Alternatives

mean_image, sub_image, convol_image, bandpass_image

See also

dyn_threshold

Module

Foundation

info_edges (: : Filter, Mode, Alpha : Size, Coeffs)

Return the filter coefficients of a filter in edges_image.

info_edges returns the coefficients Coeffs and an estimate of the width Size of any of the filters used in edges_image. To do so, the corresponding continuous impulse responses of the filters are sampled until the first filter coefficient is smaller than five percent of the largest coefficient. Alpha is the filter parameter (see edges_image). Seven edge operators are supported (parameter Filter):

'deriche1', *'lanser1'*, *'deriche2'*, *'lanser2'*, *'shen'*, *'mshen'* and *'canny'*.

The parameter Mode (*'edge'*/*'smooth'*) is used to determine whether the corresponding edge or smoothing operator is to be sampled.

The Canny operator (which uses Gaussian smoothing) is implemented using conventional filter masks. Therefore, for the Canny filter, the coefficients Coeffs of the one-dimensional impulse responses are returned as $f(n)$ with $n \geq 0$. All other filters are implemented recursively; here, the Coeffs are coefficients of a corresponding non-recursive filter.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Filter** (input_control) ..... string ⇝ *string*
Name of the edge operator.
Default: 'lanser2'
List of values: Filter ∈ {'deriche1', 'lanser1', 'deriche2', 'lanser2', 'shen', 'mshen', 'canny'}

▷ **Mode** (input_control) ..... string ⇝ *string*
1D edge filter ('edge') or 1D smoothing filter ('smooth').
Default: 'edge'
List of values: Mode ∈ {'edge', 'smooth'}

▷ **Alpha** (input_control) ..... real ⇝ *real*
Filter parameter: small values result in strong smoothing, and thus less detail (opposite for 'canny').
Default: 0.5
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Alpha > 0.0

▷ **Size** (output_control) ..... integer ⇝ *integer*
Filter width in pixels.

▷ **Coeffs** (output_control) ..... integer-array ⇝ *integer*
For Canny filters: Coefficients of the "positive" half of the 1D impulse response. All others: Coefficients of a corresponding non-recursive filter.

Example

```
read_image(Image,'fabrik')
info_edges('lanser2','edge',0.5,Size,Coeffs)
edges_image(Image,Amp,Dir,'lanser2',0.5,'none',-1,-1)
hysteresis_threshold(Amp,Margin,20,30,30)
```

Result

info_edges returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

edges_image, threshold, skeleton

See also

edges_image

Module

Foundation

kirsch_amp (Image : ImageEdgeAmp : :)

Detect edges (amplitude) using the Kirsch operator.

kirsch_amp calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$\begin{matrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{matrix}$$

$$\begin{matrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{matrix}$$

$$\begin{matrix} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{matrix}$$

$$\begin{matrix} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{matrix}$$

$$\begin{matrix} 5 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & -3 & -3 \end{matrix}$$

$$\begin{matrix} -3 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & 5 & -3 \end{matrix}$$

$$\begin{matrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{matrix}$$

$$\begin{matrix} -3 & -3 & -3 \\ -3 & 0 & 5 \\ -3 & 5 & 5 \end{matrix}$$

The result image contains the maximum response of all masks.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Input image.

▷ **ImageEdgeAmp** (output_object) ..... image(-array) ⇝ *object :* byte / int2 / uint2
Edge amplitude (gradient magnitude) image.

Example

```
read_image(Image,'fabrik')
kirsch_amp(Image,Kirsch_amp)
threshold(Kirsch_amp,Edges,128,255)
```

Result

`kirsch_amp` always returns 2 (`H_MSG_TRUE`). If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

`binomial_filter`, `gauss_filter`, `sigma_image`, `median_image`, `smooth_image`

Alternatives

`sobel_amp`, `frei_amp`, `prewitt_amp`, `robinson_amp`, `roberts`

See also

`bandpass_image`, `laplace_of_gauss`

Module

Foundation

kirsch_dir (Image : ImageEdgeAmp, ImageEdgeDir : :)

Detect edges (amplitude and direction) using the Kirsch operator.

`kirsch_dir` calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$\begin{matrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{matrix}$$

$$\begin{matrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{matrix}$$

$$\begin{matrix} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{matrix}$$

$$\begin{matrix} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{matrix}$$

$$\begin{matrix} 5 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & -3 & -3 \end{matrix}$$

$$\begin{matrix} -3 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & 5 & -3 \end{matrix}$$

$$\begin{matrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{matrix}$$

$$\begin{matrix} -3 & -3 & -3 \\ -3 & 0 & 5 \\ -3 & 5 & 5 \end{matrix}$$

The result image contains the maximum response of all masks. The edge directions are returned in ImageEdgeDir, and are stored as $x/2$. They correspond to the direction of the mask yielding the maximum response.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Input image.
- ▷ **ImageEdgeAmp** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Edge amplitude (gradient magnitude) image.
- ▷ **ImageEdgeDir** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* direction
  Edge direction image.

Example

```
read_image(Image,'fabrik')
kirsch_dir(Image,Kirsch_dirA,Kirsch_dirD)
threshold(Kirsch_dirA,Res,128,255)
```

Result

kirsch_dir always returns 2 (H_MSG_TRUE). If the input is empty the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

binomial_filter, gauss_filter, sigma_image, median_image, smooth_image

Possible Successors

hysteresis_threshold, threshold, gray_skeleton, nonmax_suppression_dir, close_edges, close_edges_length

Alternatives

edges_image, sobel_dir, robinson_dir, prewitt_dir, frei_dir

See also

bandpass_image, laplace_of_gauss

Module

Foundation

laplace (Image : ImageLaplace : ResultType, MaskSize, FilterMask :)

Calculate the Laplace operator by using finite differences.

`laplace` filters the input images `Image` using a Laplace operator. Depending on the parameter `FilterMask` the following approximations of the Laplace operator are used:

'n_4'

$$\begin{matrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{matrix}$$

'n_8'

$$\begin{matrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{matrix}$$

'n_8_isotropic'

$$\begin{matrix} 10 & 22 & 10 \\ 22 & -128 & 22 \\ 10 & 22 & 10 \end{matrix}$$

For the three filter masks the following normalizations of the resulting gray values is applied, (i.e., by dividing the result by the given divisor): *'n_4'* normalization by 1, *'n_8'*, normalization by 2 and for *'n_8_isotropic'* normalization by 32.

For a Laplace operator with size 3×3, the corresponding filter is applied directly, while for larger filter sizes the input image is first smoothed using a Gaussian filter (see `gauss_image`) or a binomial filter (see `binomial_filter`) of size `MaskSize`-2. The Gaussian filter is selected for the above values of `ResultType`. Here, `MaskSize` = 5, 7, 9, 11, or 13 must be used. The binomial filter is selected by appending *'_binomial'* to the above values of `ResultType`. Here, `MaskSize` can be selected between 5 and 39. Furthermore, it is possible to select different amounts of smoothing for the column and row direction by passing two values in `MaskSize`. Here, the first value of `MaskSize` corresponds to the mask width (smoothing in the column direction), while the second value corresponds to the mask height (smoothing in the row direction) of the binomial filter. Therefore,

```
laplace(O:R:'absolute',MaskSize,N:)
```

for `MaskSize` $>$ *3* is equivalent to

```
gauss_image(O:G:MaskSize-2:)
laplace(G:R:'absolute',3,N:)
```

and

```
laplace(O:R:'absolute_binomial',MaskSize,N:)
```

is equivalent to

```
binomial_filter(O:B:MaskSize-2,MaskSize-2:)
laplace(B:R:'absolute',3,N:).
```

`laplace` either returns the absolute value of the Laplace filtered image (`ResultType` = *'absolute'*) in a byte or uint2 image or the signed result (`ResultType` = *'signed'* or *'signed_clipped'*). Here, the output image type has the same number of bytes per pixel as the input image (i.e., int1 or int2) for *'signed_clipped'*, while the output image has the next larger number of pixels (i.e., int2 or int4) for *'signed'*.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Input image.

▷ **ImageLaplace** (output_object) . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2 / int2 / int2 / int4
Laplace-filtered result image.

▷ **ResultType** (input_control) . . . . . string ⇝ *string*
Type of the result image, whereas for byte and uint2 the absolute value is used.
Default: 'absolute'
List of values: `ResultType` ∈ {'absolute', 'signed_clipped', 'signed', 'absolute_binomial', 'signed_clipped_binomial', 'signed_binomial'}

▷ **MaskSize** (input_control) . . . . . integer(-array) ⇝ *integer*
Size of filter mask.
Default: 3
List of values: `MaskSize` ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39}

▷ **FilterMask** (input_control) . . . . . string ⇝ *string*
Filter mask used in the Laplace operator
Default: 'n_4'
List of values: `FilterMask` ∈ {'n_4', 'n_8', 'n_8_isotropic'}

Example

```
read_image(&Image,"mreut");
laplace(Image,&Laplace,"signed",3,"n_8_isotropic");
zero_crossing(Laplace,&ZeroCrossings);
```

Result

`laplace` returns 2 (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

`zero_crossing`, `dual_threshold`, `threshold`

Alternatives

`diff_of_gauss`, `laplace_of_gauss`, `derivate_gauss`

See also

`highpass_image`, `edges_image`

Module

Foundation

laplace_of_gauss (Image : ImageLaplace : Sigma :)

LoG-Operator (Laplace of Gaussian).

`laplace_of_gauss` calculates the Laplace-of-Gaussian operator, i.e., the Laplace operator on a Gaussian smoothed image, for arbitrary smoothing parameters `Sigma`. The Laplace operator is given by:

$$\Delta g(x,y) = \frac{\partial^2 g(x,y)}{\partial x^2} + \frac{\partial^2 g(x,y)}{\partial y^2}$$

The derivatives in `laplace_of_gauss` are calculated by appropriate derivatives of the Gaussian, resulting in the following formula for the convolution mask:

$$\Delta G_\sigma(x,y) =$$
$$\frac{1}{2\pi\sigma^4}\left(\frac{x^2+y^2}{2\sigma^2}-1\right)\left[\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)\right]$$

Attention

Note that filter operators may return may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **`Image`** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
  Input image.
- ▷ **`ImageLaplace`** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* int2
  Laplace filtered image.
- ▷ **`Sigma`** (input_control) . . . . . number ⇝ *real* / integer
  Smoothing parameter of the Gaussian.
  Default: 2.0
  Suggested values: `Sigma` ∈ {0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0, 5.0, 7.0}
  Value range: 0.01 ≤ `Sigma` ≤ 25.0
  Minimum increment: 0.01
  Recommended increment: 0.1

Example

```
read_image(&Image,"mreut");
laplace_of_gauss(Image,&Laplace,2.0);
zero_crossing(Laplace,&ZeroCrossings);
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

`zero_crossing`, `dual_threshold`

Alternatives

`laplace`, `diff_of_gauss`, `derivate_gauss`

See also

`derivate_gauss`

Module

Foundation

`prewitt_amp` (Image : ImageEdgeAmp : :)

Detect edges (amplitude) using the Prewitt operator.

`prewitt_amp` calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{matrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{matrix}$$

$$B = \begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix}$$

The result image contains the maximum response of the masks A and B.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Input image.

▷ **ImageEdgeAmp** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Edge amplitude (gradient magnitude) image.

Example

```
read_image(Image,'fabrik')
prewitt_amp(Image,Prewitt)
threshold(Prewitt,Edges,128,255)
```

Result

`prewitt_amp` always returns 2 (`H_MSG_TRUE`). If the input is empty the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

`binomial_filter`, `gauss_filter`, `sigma_image`, `median_image`, `smooth_image`

Possible Successors

`threshold`, `gray_skeleton`, `nonmax_suppression_amp`, `close_edges`, `close_edges_length`

Alternatives

`sobel_amp`, `kirsch_amp`, `frei_amp`, `robinson_amp`, `roberts`

See also

`bandpass_image`, `laplace_of_gauss`

Module

Foundation

prewitt_dir (Image : ImageEdgeAmp, ImageEdgeDir : :)

Detect edges (amplitude and direction) using the Prewitt operator.

`prewitt_dir` calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{matrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{matrix}$$

$$B = \begin{matrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{matrix}$$

The result image contains the maximum response of the masks A and B. The edge directions are returned in `ImageEdgeDir`, and are stored in 2-degree steps, i.e., an edge direction of x degrees mathematically positive sense and with respect to the horizontal axis is stored as $x/2$ in the edge direction image. Furthermore, the direction of the change of intensity is taken into account. Let $[E_x, E_y]$ denote the image gradient. Then the following edge directions are returned as $r/2$:

intensity increase	E_x/E_y	
edge direction r		
from bottom to top	$0/+$	0
from lower right to upper left	$-/+$	$]0, 90[$
from right to left	$-/0$	90
from upper right to lower left	$-/-$	$]90, 180[$
from top to bottom	$0/-$	180
from upper left to lower right	$+/-$	$]180, 270[$
from left to right	$+/0$	270
from lower left to upper right	$+/+$	$]270, 360[$

Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction).

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Input image.
- ▷ **`ImageEdgeAmp`** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Edge amplitude (gradient magnitude) image.
- ▷ **`ImageEdgeDir`** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* direction
  Edge direction image.

Example

```
read_image(Image,'fabrik')
prewitt_dir(Image,PrewittA,PrewittD)
threshold(PrewittA,Edges,128,255)
```

Result

`prewitt_dir` always returns 2 (`H_MSG_TRUE`). If the input is empty the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

binomial_filter, gauss_filter, sigma_image, median_image, smooth_image

Possible Successors

hysteresis_threshold, threshold, gray_skeleton, nonmax_suppression_dir, close_edges, close_edges_length

Alternatives

edges_image, sobel_dir, robinson_dir, frei_dir, kirsch_dir

See also

bandpass_image, laplace_of_gauss

Module

Foundation

roberts (Image : ImageRoberts : FilterType :)

Detect edges using the Roberts filter.

roberts calculates the first derivative of an image and is used as an edge operator. If the following mask describes a part of the image,

$$\begin{array}{cc} A & B \\ C & D \end{array}$$

the different filter types are defined as follows:

'roberts_max'	$max(\|A-D\|, \|B-C\|)$
'gradient_max'	$max(\|A+B-(C+D)\|, \|A+C-(B+D)\|)$
'gradient_sum'	$\|A+B-(C+D)\| + \|A+C-(B+D)\|$

If an overflow occurs the result is clipped. The result of the operator is stored at the pixel with the coordinates of "D".

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Input image.
- ▷ **ImageRoberts** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Roberts-filtered result images.
- ▷ **FilterType** (input_control) ..... string ⇝ *string*
  Filter type.
  Default: 'gradient_sum'
  List of values: FilterType ∈ {'roberts_max', 'gradient_max', 'gradient_sum'}

Example

```
read_image(Image,'fabrik')
roberts(Image,Roberts,'roberts_max')
threshold(Roberts,Margin,128,255)
```

Result

roberts returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

binomial_filter, gauss_filter

Possible Successors

threshold, skeleton

Alternatives

edges_image, sobel_amp, frei_amp, kirsch_amp, prewitt_amp

See also

laplace, highpass_image, bandpass_image

Module

Foundation

robinson_amp (Image : ImageEdgeAmp : :)

Detect edges (amplitude) using the Robinson operator.

robinson_amp calculates an approximation of the first derivative of the image data and is used as an edge detector. In robinson_amp the following four of the originally proposed eight 3×3 filter masks are convolved with the image. The other four masks are obtained by a multiplication by -1. All masks contain only the values 0,1,-1,2,-2.

$$\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$$

$$\begin{matrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{matrix}$$

$$\begin{matrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{matrix}$$

$$\begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

The result image contains the maximum response of all masks.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Input image.

▷ **ImageEdgeAmp** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Edge amplitude (gradient magnitude) image.

Example

```
read_image(Image,'fabrik')
robinson_amp(Image,Robinson_amp)
threshold(Robinson_amp,Edges,128,255)
```

Result

`robinson_amp` always returns `2` (`H_MSG_TRUE`). If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

`binomial_filter`, `gauss_filter`, `sigma_image`, `median_image`, `smooth_image`

Alternatives

`sobel_amp`, `frei_amp`, `prewitt_amp`, `roberts`

See also

`bandpass_image`, `laplace_of_gauss`

Module

Foundation

robinson_dir (Image : ImageEdgeAmp, ImageEdgeDir : :)

Detect edges (amplitude and direction) using the Robinson operator.

`robinson_dir` calculates an approximation of the first derivative of the image data and is used as an edge detector. In `robinson_amp` the following four of the originally proposed eight 3×3 filter masks are convolved with the image. The other four masks are obtained by a multiplication by -1. All masks contain only the values 0,1,-1,2,-2.

$$\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}$$

$$\begin{matrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{matrix}$$

$$\begin{matrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{matrix}$$

$$\begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

The result image contains the maximum response of all masks. The edge directions are returned in `ImageEdgeDir`, and are stored as $x/2$. They correspond to the direction of the mask yielding the maximum response.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Input image.
- ▷ **ImageEdgeAmp** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Edge amplitude (gradient magnitude) image.
- ▷ **ImageEdgeDir** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* direction
  Edge direction image.

Example

```
read_image(Image,'fabrik')
robinson_dir(Image,Robinson_dirA,Robinson_dirD)
threshold(Robinson_dirA,Res,128,255)
```

Result

`robinson_dir` always returns `2` (`H_MSG_TRUE`). If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

`binomial_filter`, `gauss_filter`, `sigma_image`, `median_image`, `smooth_image`

Possible Successors

`hysteresis_threshold`, `threshold`, `gray_skeleton`, `nonmax_suppression_dir`, `close_edges`, `close_edges_length`

Alternatives

`edges_image`, `sobel_dir`, `kirsch_dir`, `prewitt_dir`, `frei_dir`

See also

`bandpass_image`, `laplace_of_gauss`

Module

Foundation

sobel_amp (Image : EdgeAmplitude : FilterType, Size :)

Detect edges (amplitude) using the Sobel operator.

`sobel_amp` calculates first derivative of an image and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

$$B = \begin{matrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{matrix}$$

These masks are used differently, according to the selected filter type. (In the following, a and b denote the results of convolving an image with A and B for one particular pixel.)

$'sum\_sqrt'$	$\sqrt{a^2+b^2}/4$
$'sum\_abs'$	$(\|a\|+\|b\|)/4$
$'thin\_sum\_abs'$	$(thin(\|a\|)+thin(\|b\|))/4$
$'thin\_max\_abs'$	$max(thin(\|a\|),thin(\|b\|))/4$
$'x'$	$b/4$ (b for `real` images)
$'y'$	$a/4$ (a for `real` images)

Here, $thin(x)$ is equal to x for a vertical maximum (mask A) and a horizontal maximum (mask B), respectively, and 0 otherwise. Thus, for *'thin_sum_abs'* and *'thin_max_abs'* the gradient image is thinned. For the filter types *'x'* and *'y'* if the input image is of type byte the output image is of type int1, of type int2 otherwise. For a Sobel operator with size 3×3, the corresponding filters A and B are applied directly, while for larger filter sizes the input image is first smoothed using a Gaussian filter (see `gauss_image`) or a binomial filter (see `binomial_filter`) of size `Size`-2. The Gaussian filter is selected for the above values of `FilterType`. Here, `Size` = 5, 7, 9, 11, or 13 must be used. The binomial filter is selected by appending *'_binomial'* to the above values of `FilterType`. Here, `Size` can be selected between 5 and 39. Furthermore, it is possible to select different amounts of smoothing the column and row direction by passing two values in `Size`. Here, the first value of `Size` corresponds to the mask width (smoothing in the column direction), while the second value corresponds to the mask height (smoothing in the row direction) of the binomial filter. The binomial filter can only be used for images of type byte, uint2 and real. Since smoothing reduces the edge amplitudes, in this case the edge amplitudes are multiplied by a factor of 2 to prevent information loss. Therefore,

```
sobel_amp(I,E,FilterType,S)
```

for `Size` > 3 is conceptually equivalent to

```
scale_image(I,F,2,0)
gauss_image(F,G,S-2)
sobel_amp(G,E,FilterType,3)
```

or to

```
scale_image(I,F,2,0)
binomial_filter(F,G,S[0]-2,S[1]-2)
sobel_amp(G,E,FilterType,3).
```

For `sobel_amp` special optimizations are implemented `FilterType` $= 'sum\_abs'$ that use SIMD technology. The actual application of these special optimizations is controlled by the system parameter *'sse2_enable'* and *'avx2_enable'*, respectively (see `set_system`). If *'sse2_enable'* or *'avx2_enable'*is set to *'true'* (and the SIMD instruction set is available), the internal calculations are performed using SIMD technology. Note that SIMD technology performs best on large, compact input regions. Depending on the input region and the capabilities of the hardware the execution of `sobel_amp` might even take significantly more time with SIMD technology than without.

`sobel_amp` can be executed on OpenCL devices for the filter types *'sum_abs'*, *'sum_sqrt'*, *'x'* and *'y'* (as well as their binomial variants). Note that when using gaussian filtering for `Size` > 3, the results can vary from the CPU implementation.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
  Input image.
- ▷ **`EdgeAmplitude`** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* int1 / int2 / uint2 / real
  Edge amplitude (gradient magnitude) image.

▷ **FilterType** (input_control) . . . . . string ⇝ *string*
Filter type.
Default: 'sum_abs'
List of values: FilterType ∈ {'sum_abs', 'thin_sum_abs', 'thin_max_abs', 'sum_sqrt', 'x', 'y', 'sum_abs_binomial', 'thin_sum_abs_binomial', 'thin_max_abs_binomial', 'sum_sqrt_binomial', 'x_binomial', 'y_binomial'}

▷ **Size** (input_control) . . . . . integer(-array) ⇝ *integer*
Size of filter mask.
Default: 3
List of values: Size ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39}

Example

```
read_image(Image,'fabrik')
sobel_amp(Image,Amp,'sum_abs',3)
threshold(Amp,Edg,128,255)
```

Result

sobel_amp returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

binomial_filter, gauss_filter, mean_image, anisotropic_diffusion, sigma_image

Possible Successors

threshold, nonmax_suppression_amp, gray_skeleton

Alternatives

frei_amp, roberts, kirsch_amp, prewitt_amp, robinson_amp

See also

laplace, highpass_image, bandpass_image

Module

Foundation

sobel_dir (Image : EdgeAmplitude, EdgeDirection : FilterType, Size :)

Detect edges (amplitude and direction) using the Sobel operator.

sobel_dir calculates first derivative of an image and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

$$B = \begin{matrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{matrix}$$

These masks are used differently, according to the selected filter type. (In the following, a and b denote the results of convolving an image with A and B for one particular pixel.)

- *'sum_sqrt'*: $\sqrt{a^2 + b^2}/4$
- *'sum_abs'*: $(|a| + |b|)/4$

For a Sobel operator with size 3×3, the corresponding filters A and B are applied directly, while for larger filter sizes the input image is first smoothed using a Gaussian filter (see `gauss_image`) or a binomial filter (see `binomial_filter`) of size `Size`-2. The Gaussian filter is selected for the above values of `FilterType`. Here, `Size` = 5, 7, 9, 11, or 13 must be used. The binomial filter is selected by appending *'_binomial'* to the above values of `FilterType`. Here, `Size` can be selected between 5 and 39. Furthermore, it is possible to select different amounts of smoothing the column and row direction by passing two values in `Size`. Here, the first value of `Size` corresponds to the mask width (smoothing in the column direction), while the second value corresponds to the mask height (smoothing in the row direction) of the binomial filter. The binomial filter can only be used for images of type byte, uint2 and real. Since smoothing reduces the edge amplitudes, in this case the edge amplitudes are multiplied by a factor of 2 to prevent information loss. Therefore,

```
sobel_dir(I,Amp,Dir,FilterType,S)
```

for `Size` $>$ *3* is conceptually equivalent to

```
scale_image(I,F,2,0)
gauss_image(F,G,S-2)
sobel_dir(G,Amp,Dir,FilterType,3)
```

or to

```
scale_image(I,F,2,0)
binomial_filter(F,G,S[0]-2,S[1]-2)
sobel_dir(G,Amp,Dir,FilterType,3).
```

The edge directions are returned in `EdgeDirection`, and are stored in 2-degree steps, i.e., an edge direction of x degrees in mathematically positive sense and with respect to the horizontal axis is stored as $x/2$ in the edge direction image. Furthermore, the direction of the change of intensity is taken into account. Let $[E_x, E_y]$ denote the image gradient. Then the following edge directions r are returned as $r/2$:

intensity increase	E_x/E_y	**edge direction** r **[deg]**
from bottom to top	0 / +	0
from lower right to upper left	- / +	]0,90[
from right to left	- / 0	90
from upper right to lower left	- / -	]90,180[
from top to bottom	0 / -	180
from upper left to lower right	+ / -	]180,270[
from left to right	+ / 0	270
from lower left to upper right	+ / +	]270,360[.

Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction).

`sobel_amp` can be executed on OpenCL devices. Note that when using gaussian filtering for `Size` $>$ 3, the results can vary from the CPU implementation.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **`Image`** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Input image.

▷ **`EdgeAmplitude`** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Edge amplitude (gradient magnitude) image.

▷ **EdgeDirection** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* direction
Edge direction image.

▷ **FilterType** (input_control) . . . . . string ⇝ *string*
Filter type.
Default: 'sum_abs'
List of values: FilterType ∈ {'sum_abs', 'sum_sqrt', 'sum_abs_binomial', 'sum_sqrt_binomial'}

▷ **Size** (input_control) . . . . . integer(-array) ⇝ *integer*
Size of filter mask.
Default: 3
List of values: Size ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39}

Example

```
read_image(Image,'fabrik')
sobel_dir(Image,Amp,Dir,'sum_abs',3)
threshold(Amp,Edg,128,255)
```

Result

sobel_dir returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

binomial_filter, gauss_filter, mean_image, anisotropic_diffusion, sigma_image

Possible Successors

nonmax_suppression_dir, hysteresis_threshold, threshold

Alternatives

edges_image, frei_dir, kirsch_dir, prewitt_dir, robinson_dir

See also

roberts, laplace, highpass_image, bandpass_image

Module

Foundation

12.5 Enhancement

coherence_enhancing_diff (Image : ImageCED : Sigma, Rho, Theta, Iterations :)

Perform a coherence enhancing diffusion of an image.

The operator coherence_enhancing_diff performs an anisotropic diffusion process on the input image Image to increase the coherence of the image structures contained in Image. In particular, noncontinuous image edges are connected by diffusion, without being smoothed perpendicular to their dominating direction. For this, coherence_enhancing_diff uses the anisotropic diffusion equation

$$u_t = div(G(u)\nabla u)$$

formulated by Weickert. With a 2×2 coefficient matrix G that depends on the gray values in Image, this is an enhancement of the mean curvature flow or intrinsic heat equation

$$u_t = div(\frac{\nabla u}{|\nabla u|})|\nabla u| = curv(u)|\nabla u|$$

on the gray value function u defined by the input image Image at a time $t_0 = 0$. The smoothing operator mean_curvature_flow is a direct application of the mean curvature flow equation. The discrete diffusion equation is solved in Iterations time steps of length Theta, so that the output image ImageCED contains the gray value function at the time Iterations · Theta.

To detect the edge direction more robustly, in particular on noisy input data, an additional isotropic smoothing step can precede the computation of the gray value gradients. The parameter Sigma determines the magnitude of the smoothing by means of the standard deviation of a corresponding Gaussian convolution kernel, as used in the operator isotropic_diffusion for isotropic image smoothing.

While the matrix G is given by

$$G_{MCF}(u) = I - \frac{1}{|\nabla u|^2}\nabla u(\nabla u)^T,$$

in the case of the operator mean_curvature_flow, where I denotes the unit matrix, G_{MCF} is again smoothed componentwise by a Gaussian filter of standard deviation Rho for coherence_enhancing_diff. Then, the final coefficient matrix

$$G_{CED} = g_1\left((\lambda_1 - \lambda_2)^2\right) w_1(w_1)^T + g_2\left((\lambda_1 - \lambda_2)^2\right) w_2(w_2)^T$$

is constructed from the eigenvalues λ_1, λ_2 and eigenvectors w_1, w_2 of the resulting intermediate matrix, where the functions

$$\begin{aligned} g_1(p) &= 0.001 \\ g_2(p) &= 0.001 + 0.999 \exp\left(\frac{-1}{p}\right) \end{aligned}$$

were determined empirically and taken from the publication of Weickert.

Hence, the diffusion direction in mean_curvature_flow is only determined by the local direction of the gray value gradient, while G_{CED} considers the macroscopic structure of the image objects on the scale Rho and the magnitude of the diffusion in coherence_enhancing_diff depends on how well this structure is defined.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **ImageCED** (output_object) . . . . . image(-array) ⇝ *object :* byte / uint2 / real
Output image.

▷ **Sigma** (input_control) . . . . . real ⇝ *real*
Smoothing for derivative operator.
Default: 0.5
Suggested values: Sigma ∈ {0.0, 0.1, 0.5, 1.0}
Restriction: Sigma >= 0

▷ **Rho** (input_control) . . . . . real ⇝ *real*
Smoothing for diffusion coefficients.
Default: 3.0
Suggested values: Rho ∈ {0.0, 1.0, 3.0, 5.0, 10.0, 30.0}
Restriction: Rho >= 0

▷ **Theta** (input_control) . . . . . real ⇝ *real*
Time step.
Default: 0.5
Suggested values: Theta ∈ {0.1, 0.2, 0.3, 0.4, 0.5}
Restriction: 0 < Theta <= 0.5

▷ **Iterations** (input_control) . . . . . integer ⇝ *integer*
Number of iterations.
Default: 10
Suggested values: Iterations ∈ {1, 5, 10, 20, 50, 100, 500}
Restriction: Iterations >= 1

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

References

J. Weickert, V. Hlavac, R. Sara; "Multiscale texture enhancement"; Computer analysis of images and patterns, Lecture Notes in Computer Science, Vol. 970, pp. 230-237; Springer, Berlin; 1995.

J. Weickert, B. ter Haar Romeny, L. Florack, J. Koenderink, M. Viergever; "A review of nonlinear diffusion filtering"; Scale-Space Theory in Computer Vision, Lecture Notes in Comp. Science, Vol. 1252, pp. 3-28; Springer, Berlin; 1997.

Module

Foundation

emphasize (Image : ImageEmphasize : MaskWidth, MaskHeight, Factor :)

Enhance contrast of the image.

The operator emphasize emphasizes high frequency areas of the image (edges and corners). The resulting images appears sharper.

First the procedure carries out a filtering with the low pass (mean_image). The resulting gray values (res) are calculated from the obtained gray values ($mean$) and the original gray values ($orig$) as follows:

$$res := round((orig - mean) * \texttt{Factor}) + orig$$

Factor serves as measurement of the increase in contrast. The division frequency is determined via the size of the filter matrix: The larger the matrix, the lower the division frequency.

As an edge treatment the gray values are mirrored at the edges of the image. Overflow and/or underflow of gray values is clipped.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Image to be enhanced.

▷ **ImageEmphasize** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
contrast enhanced image.

▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *integer*
Width of low pass mask.
Default: 7
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 15, 21, 25, 31, 39}
Value range: 3 ≤ MaskWidth
Minimum increment: 2
Recommended increment: 2

▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *integer*
Height of the low pass mask.
Default: 7
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 15, 21, 25, 31, 39}
Value range: 3 ≤ MaskHeight
Minimum increment: 2
Recommended increment: 2

▷ **Factor** (input_control) . . . . . real ⇝ *real*
Intensity of contrast emphasis.
Default: 1.0
Suggested values: Factor ∈ {0.3, 0.5, 0.7, 1.0, 1.4, 1.8, 2.0}
Value range: 0.0 ≤ Factor (sqrt)
Recommended increment: 0.2

Example

```
read_image(Image,'mreut')
dev_display (Image)
draw_region(Region,WindowHandle)
reduce_domain(Image,Region,Mask)
emphasize(Mask,Sharp,7,7,2.0)
dev_display (Sharp)
```

Result

If the parameter values are correct the operator emphasize returns the value 2 (H_MSG_TRUE) The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

disp_image

Alternatives

mean_image, sub_image, laplace, add_image

See also

mean_image, highpass_image

Module

Foundation

equu_histo_image (Image : ImageEquHisto : :)

Histogram linearization of images

The operator equ_histo_image enhances the contrast. The starting point is the histogram of the input images. The following simple gray value transformation $f(g)$ is carried out for byte images:

$$f(g) = 255 \sum_{x=0\ldots g} h(x)$$

$h(x)$ describes the relative frequency of the occurrence of the gray value x. For uint2 images, the only difference is that the value 255 is replaced with a different maximum value. The maximum value is computed from the number of significant bits stored with the input image, provided that this value is set. If not, the value of the system parameter *'int2_bits'* is used (see set_system), if this value is set (i.e., different from -1). If none of the two values is set, the number of significant bits is set to 16.

This transformation linearizes the cumulative histogram. Maxima in the original histogram are "spreaded" and thus the contrast in image regions with these frequently occurring gray values is increased. Supposedly homogeneous regions receive more easily visible structures. On the other hand, of course, the noise in the image increases correspondingly. Minima in the original histogram are dually "compressed". The transformed histogram contains gaps, but the remaining gray values used occur approximately at the same frequency ("histogram equalization").

Attention

The operator equ_histo_image primarily serves for optical processing of images for a human viewer. For example, the (local) contrast spreading can lead to a detection of fictitious edges.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
  Image to be enhanced.
- ▷ **ImageEquHisto** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
  Image with linearized gray values.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Successors

disp_image

Alternatives

equ_histo_image_rect, scale_image, scale_image_max, illuminate

See also

equ_histo_image_rect, scale_image

References

R.C. Gonzales, P. Wintz: "Digital Image Processing"; Second edition; Addison Wesley; 1987.

Module

Foundation

equ_histo_image_rect (Image : ImageEquHisto : Mode, MaskWidth, MaskHeight, MaxContrast :)

Histogram linearization within a rectangluar mask.

The operator equ_histo_image_rect enhances the contrast. Similar to equ_histo_image, it applies a transformation to linearize the cumulative histogram. However, instead of using the histogram of the whole image, only the local neighborhood of each pixel is considered to compute the transformation. The size of this rectangular neighborhood region can be set by MaskWidth and MaskHeight.

The local contrast maximization improves the visibility of low-contrast structures, but also amplifies noise. The parameter MaxContrast can be used to limit the maximum contrast in a local neighborhood, which effectively reduces the amplification of noise. Therefore, the method is also known as Contrast-Limited Adaptive Histogram Equalization (CLAHE).

The parameter Mode determines the processing mode. In the *'accurate'* mode, the transformation is computed for each pixel as described above. In the *'fast'* mode, the transformation is computed only for a subset of all pixels and interpolated between those points. Due to the interpolation, the results of both modes can differ significantly.

Attention

The operator equ_histo_image_rect primarily serves for optical processing of images for a human viewer. For example, the contrast spreading can lead to a detection of fictitious edges.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte
  Image to be enhanced.
- ▷ **ImageEquHisto** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte
  Image with linearized gray values.
- ▷ **Mode** (input_control) . . . . . string ⇝ *string*
  Processing mode.
  Default: 'accurate'
  Suggested values: Mode ∈ {'accurate', 'fast'}
- ▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *integer*
  Width of the filter mask.
  Default: 51
  Suggested values: MaskWidth ∈ {31, 51, 101, 151}
  Value range: 1 ≤ MaskWidth
  Minimum increment: 2
  Restriction: MaskWidth <= width(Image)
- ▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *integer*
  Height of the filter mask.
  Default: 51
  Suggested values: MaskHeight ∈ {31, 51, 101, 151}
  Value range: 1 ≤ MaskHeight
  Minimum increment: 2
  Restriction: MaskHeight <= height(Image)
- ▷ **MaxContrast** (input_control) . . . . . real ⇝ *real* / integer
  Maximum contrast.
  Default: 0.01
  Suggested values: MaxContrast ∈ {0, 0.01, 0.02, 0.05, 0.1, 0.5, 1}
  Value range: 0 ≤ MaxContrast ≤ 1

Complexity

In Mode='*accurate*', equ_histo_image_rect uses an algorithm with constant complexity per pixel, i.e., the runtime only depends on the size of the input image and not on the mask size.

In Mode='*fast*', the number of histograms to be computed depends on the mask size. Hence, the runtime increases with smaller mask sizes and decreases with larger mask sizes.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

disp_image

Alternatives

equ_histo_image, scale_image, scale_image_max, illuminate

See also

equ_histo_image, scale_image

References

S. Pizer et al.: "Adaptive Histogram Equalization and Its Variations"; Computer Vision, Graphics, and Image Processing, vol. 39, no. 3, pp. 355-368, 1987, doi: 10.1016/S0734-189X(87)80186-X.

Philipp Härtinger, Carsten Steger: "Adaptive histogram equalization in constant time"; Journal of Real-Time Image Processing, vol. 21, article 93, 2024, doi: 10.1007/s11554-024-01465-1.

Module

Foundation

illuminate (Image : ImageIlluminate : MaskWidth, MaskHeight, Factor :)

Illuminate image.

The operator illuminate enhances contrast. Very dark parts of the image are "illuminated" more strongly, very light ones are "darkened". If $orig$ is the original gray value and $mean$ is the corresponding gray value of the low pass filtered image detected via the operators mean_image and filter size MaskHeight x MaskWidth. For byte-images val equals 127, for int2-images and uint2-images val equals the median value. The resulting gray value is new:

$$new := round((val - mean) * \texttt{Factor} + orig)$$

The low pass should have rather large dimensions (30 x 30 to 200 x 200). Reasonable parameter combinations might be:

MaskHeight	MaskWidth	Factor
40	40	0.55
100	100	0.7
150	150	0.8

i.e. the larger the low pass mask is chosen, the larger Factor should be as well.

The following "spotlight effect" should be noted: If, for example, a dark object is in front of a light wall the object as well as the wall, which is already light in the immediate proximity of the object contours, are lightened by the operator illuminate. This corresponds roughly to the effect that is produced when the object is illuminated by a strong spotlight. The same applies to light objects in front of a darker background. In this case, however, the fictitious "spotlight" darkens objects.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Image to be enhanced.

▷ **ImageIlluminate** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
"Illuminated" image.

▷ **MaskWidth** (input_control) ..... extent.x ⇝ *integer*
Width of low pass mask.
Default: 101
Suggested values: MaskWidth ∈ {31, 41, 51, 71, 101, 121, 151, 201}
Minimum increment: 2
Recommended increment: 10
Restriction: MaskWidth < width(Image) * 2

▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *integer*
Height of low pass mask.
Default: 101
Suggested values: MaskHeight ∈ {31, 41, 51, 71, 101, 121, 151, 201}
Minimum increment: 2
Recommended increment: 10
Restriction: MaskHeight < height(Image) * 2

▷ **Factor** (input_control) . . . . . real ⇝ *real*
Scales the "correction gray value" added to the original gray values.
Default: 0.7
Suggested values: Factor ∈ {0.3, 0.5, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0}
Value range: 0.0 ≤ Factor
Minimum increment: 0.01
Recommended increment: 0.2

Example

```
read_image(Image,'fabrik')
dev_display (Image)
illuminate(Image,Better,40,40,0.55)
dev_display (Better)
```

Result

If the parameter values are correct the operator illuminate returns the value 2 (H_MSG_TRUE) The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Successors

disp_image

Alternatives

scale_image_max, equ_histo_image, equ_histo_image_rect, mean_image, sub_image

See also

emphasize, gray_histo

Module

Foundation

mean_curvature_flow (Image : ImageMCF : Sigma, Theta, Iterations :)

Apply the mean curvature flow to an image.

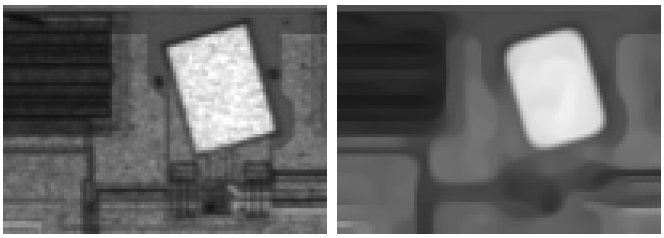
The operator mean_curvature_flow applies the mean curvature flow or intrinsic heat equatio

$$u_t = div(\frac{\nabla u}{|\nabla u|})|\nabla u| = curv(u)|\nabla u|$$

to the gray value function u defined by the input image Image at a time $t_0 = 0$. The discretized equation is solved in Iterations time steps of length Theta, so that the output image contains the gray value function at the time Iterations · Theta.

The mean curvature flow causes a smoothing of Image in the direction of the edges in the image, i.e. along the contour lines of u, while perpendicular to the edge direction no smoothing is performed and hence the boundaries of image objects are not smoothed. To detect the image direction more robustly, in particular on noisy input data, an additional isotropic smoothing step can precede the computation of the gray value gradients. The parameter Sigma determines the magnitude of the smoothing by means of the standard deviation of a corresponding Gaussian convolution kernel, as used in the operator isotropic_diffusion for isotropic image smoothing.

The following images show the effect of the parameters Sigma, Theta, and Iterations. First, the input image is shown together with the result that is achieved if all parameters are set to their default values.

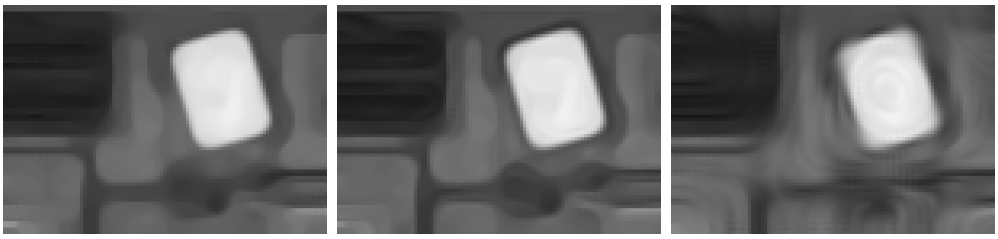


(1) (2)

(1) Input image. (2) Result when using the default values.

In the following images, the results are shown that are achieved if one parameter is varied while setting the other two parameters to their default values.

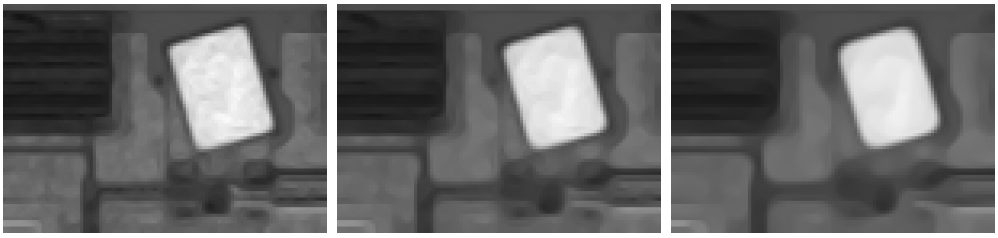
Sigma controls the amount of smoothing, prior to the computation of the gray value gradient. Be careful with very large values for Sigma, because they may lead to undesired effects.



(1) (2) (3)

(1) Sigma = 0.0. (2) Sigma = 1.0. (3) Sigma = 10.0.

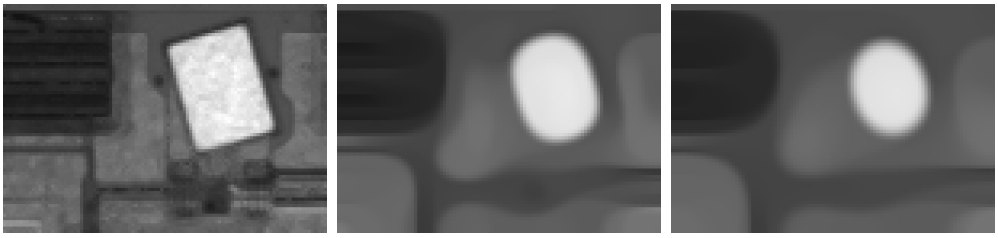
Theta controls the step size during the iterative smoothing process. Larger values lead to a stronger smoothing.



(1) (2) (3)

(1) Theta = 0.1. (2) Theta = 0.2. (3) Theta = 0.4.

Iterations controls the number of iterations that are performed. With an increasing number of iterations, the runtime increases, as well. Furthermore, a large number of iterations may lead to a loss of structure in the smoothed image.



(1) (2) (3)

(1) Iterations = 1. (2) Iterations = 50. (3) Iterations = 100.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **ImageMCF** (output_object) ..... image(-array) ⇝ *object :* byte / uint2 / real
Output image.

▷ **Sigma** (input_control) ..... real ⇝ *real*
Smoothing parameter for derivative operator.
Default: 0.5
Suggested values: `Sigma` ∈ {0.0, 0.1, 0.5, 1.0}
Restriction: Sigma >= 0

▷ **Theta** (input_control) ..... real ⇝ *real*
Time step.
Default: 0.5
Suggested values: `Theta` ∈ {0.1, 0.2, 0.3, 0.4, 0.5}
Restriction: 0 < Theta <= 0.5

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 10
Suggested values: `Iterations` ∈ {1, 5, 10, 20, 50, 100, 500}
Restriction: Iterations >= 1

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

References

M. G. Crandall, P. Lions; “Convergent Difference Schemes for Nonlinear Parabolic Equations and Mean Curvature Motion”; Numer. Math. 75 pp. 17-41; 1996.

G. Aubert, P. Kornprobst; “Mathematical Problems in Image Processing”; Applied Mathematical Sciences 147; Springer, New York; 2002.

Module

Foundation

scale_image_max (Image : ImageScaleMax : :)

Maximum gray value spreading in the value range 0 to 255.

The operator `scale_image_max` calculates the minimum and maximum and scales the image to the maximum value range of a byte image. This way the dynamics (value range) is fully exploited. The number of different gray scales does not change, but in general the visual impression is enhanced. The gray values of images of the *'real'*, *'int2'*, *'uint2'*, *'int4'*, and *'int8'* type are scaled to the range *0* to *255* and returned as *'byte'* images.

Attention

The output always is an image of the type *'byte'*.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / int8 / real
Image to be scaled.

▷ **ImageScaleMax** (output_object) . . . . . image(-array) ⇝ *object :* byte
contrast enhanced image.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Successors

`disp_image`

Alternatives

`equ_histo_image`, `equ_histo_image_rect`, `scale_image`, `illuminate`, `convert_image_type`

See also

`min_max_gray`, `gray_histo`

Module

Foundation

shock_filter (Image : SharpenedImage : Theta, Iterations, Mode, Sigma :)

Apply a shock filter to an image.

The operator `shock_filter` applies a shock filter to the input image `Image` to sharpen the edges contained in it. The principle of the shock filter is based on the transport of the gray values of the image towards an edge from both sides through dilation and erosion and satisfies the differential equation

$$u_t = s\,|\nabla u|$$

on the function u defined by the gray values in `Image` at a time $t_0 = 0$. The discretized equation is solved in `Iterations` time steps of length `Theta`, so that the output image `SharpenedImage` contains the gray value function at the time `Iterations` · `Theta`.

The decision between dilation and erosion is made using the sign function $s \in \{-1, 0, +1\}$ on a conventional edge detector. The detector of Canny

$$s = -sgn\left(D^2u(\frac{\nabla u}{|\nabla u|}, \frac{\nabla u}{|\nabla u|})\right)$$

is available with `Mode` $= 'canny'$ and the detector of Marr/Hildreth (the Laplace operator)

$$s = -sgn(\Delta u)$$

can be selected by `Mode` $= 'laplace'$.

To make the edge detection more robust, in particular on noisy images, it can be performed on a smoothed image matrix. This is done by giving the standard deviation of a Gaussian kernel for convolution with the image matrix in the parameter `Sigma`.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **SharpenedImage** (output_object) . . . . . image(-array) ⇝ *object :* byte / uint2 / real
Output image.

▷ **Theta** (input_control) . . . . . real ⇝ *real*
Time step.
Default: 0.5
Suggested values: Theta ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7}
Restriction: 0 < Theta <= 0.7

▷ **Iterations** (input_control) . . . . . integer ⇝ *integer*
Number of iterations.
Default: 10
Suggested values: Iterations ∈ {1, 3, 10, 100}
Restriction: Iterations >= 1

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Type of edge detector.
Default: 'canny'
List of values: Mode ∈ {'laplace', 'canny'}

▷ **Sigma** (input_control) . . . . . real ⇝ *real*
Smoothing of edge detector.
Default: 1.0
Suggested values: Sigma ∈ {0.0, 0.5, 1.0, 2.0, 5.0}
Restriction: Theta >= 0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

References

F. Guichard, J. Morel; "A Note on Two Classical Shock Filters and Their Asymptotics"; Michael Kerckhove (Ed.): Scale-Space and Morphology in Computer Vision, LNCS 2106, pp. 75-84; Springer, New York; 2001.

G. Aubert, P. Kornprobst; "Mathematical Problems in Image Processing"; Applied Mathematical Sciences 147; Springer, New York; 2002.

Module

Foundation

12.6 FFT

convol_fft (ImageFFT, ImageFilter : ImageConvol : :)

Multiply an image with a filter image in the frequency domain.

As part of calculating the convolution of an image with a filter image, convol_fft multiplies the Fourier transform of an image ImageFFT with the Fourier transform of a second image ImageFilter, which serves as the filter.

According to the convolution theorem, the non-normalized convolution of two images in pixel space can be obtained in three steps:

1. Transforming the images into frequency space using a Fourier transform (see, e.g., fft_generic).
2. Multiplying one transformed image with the transformed filter image (pixel-wise).
3. Transforming the result back into pixel space using an inverse Fourier transform (see, e.g., fft_generic).

The operator convol_fft is used to perform the second step, i.e., ImageFFT is pixel-wise multiplied with ImageFilter.

Attention

The filtering is always done on the entire image, i.e., the domain of the image is ignored.

Parameters

- ▷ **ImageFFT** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
  Complex input image.
- ▷ **ImageFilter** (input_object) ..... (multichannel-)image ⇝ *object :* real / complex
  Filter in frequency domain.
- ▷ **ImageConvol** (output_object) ..... image(-array) ⇝ *object :* complex
  Result in the frequency domain.

Example

```
gen_highpass(Highpass,0.2,'n','dc_edge',Width,Height)
fft_generic(Image,ImageFFT,'to_freq',-1,'none','dc_edge','complex')
convol_fft(ImageFFT,Highpass,ImageConvol)
fft_generic(ImageConvol,ImageResult,'from_freq',1,'none','dc_edge','byte')
```

Result

convol_fft returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Predecessors

fft_image, fft_generic, rft_generic, gen_highpass, gen_lowpass, gen_bandpass, gen_bandfilter

Possible Successors

power_byte, power_real, power_ln, fft_image_inv, fft_generic, rft_generic

Alternatives

convol_gabor

See also

gen_gabor, gen_highpass, gen_lowpass, gen_bandpass, convol_gabor, fft_image_inv

Module

Foundation

convol_gabor (ImageFFT, GaborFilter : ImageResultGabor, ImageResultHilbert : :)

Convolve an image with a Gabor filter in the frequency domain.

convol_gabor convolves a Fourier-transformed image with a Gabor filter GaborFilter (see gen_gabor) and its Hilbert transform in the frequency domain. The result image is of type *'complex'*.

Attention

The filtering is always done on the entire image, i.e., the domain of the image is ignored.

Parameters

- ▷ **ImageFFT** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
  Input image.
- ▷ **GaborFilter** (input_object) ..... multichannel-image ⇝ *object :* real
  Gabor/Hilbert-Filter.
- ▷ **ImageResultGabor** (output_object) ..... image(-array) ⇝ *object :* complex
  Result of the Gabor filter.
- ▷ **ImageResultHilbert** (output_object) ..... image(-array) ⇝ *object :* complex
  Result of the Hilbert filter.

Example

```
gen_gabor(Filter,1.4,0.4,1.0,1.5,'n','dc_edge',512,512)
fft_generic(Image,ImageFFT,'to_freq',-1,'none','dc_edge','complex')
convol_gabor(ImageFFT,Filter,Gabor,Hilbert)
fft_generic(Gabor,GaborInv,'from_freq',1,'none','dc_edge','byte')
fft_generic(Hilbert,HilbertInv,'from_freq',1,'none','dc_edge','byte')
energy_gabor(GaborInv,HilbertInv,Energy)
```

Result

`convol_gabor` returns `2` (`H_MSG_TRUE`) if all images are of correct type. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Predecessors

`fft_image`, `fft_generic`, `gen_gabor`

Possible Successors

`power_byte`, `power_real`, `power_ln`, `fft_image_inv`, `fft_generic`

Alternatives

`convol_fft`

See also

`convol_image`

Module

Foundation

correlation_fft (ImageFFT1, ImageFFT2 : ImageCorrelation : :)

Multiply one image with the complex conjugate of another image in the frequency domain.

As part of calculating the correlation between two images, `correlation_fft` multiplies the Fourier transform of the first image `ImageFFT1` with the complex conjugate of the Fourier-transformed second image `ImageFFT2`.

According to the correlation theorem, the non-normalized correlation of two images in pixel space can be obtained in three steps:

1. Transforming the images into frequency space using a Fourier transform (see, e.g., `fft_generic`).
2. Multiplying one transformed image with the complex conjugate of the other transformed image (pixel-wise).
3. Transforming the result back into pixel space using an inverse Fourier transform (see, e.g., `fft_generic`).

The operator correlation_fft is used to perform the second step, i.e., ImageFFT1 is pixel-wise multiplied with the complex conjugate of ImageFFT2.

It should be noted that in order to achieve a correct scaling of the correlation in the spatial domain, the operators fft_generic or rft_generic with Norm = *'none'* must be used for the forward transform (step 1) and fft_generic or rft_generic with Norm = *'n'* for the reverse transform (step 3). If ImageFFT1 and ImageFFT2 contain the same number of images, the corresponding images are multiplied pairwise. Otherwise, ImageFFT2 must contain only one single image. In this case, the multiplication is performed for each image of ImageFFT1 with ImageFFT2.

Attention

The filtering is always performed on the entire image, i.e., the domain of the image is ignored.

Parameters

- ▷ **ImageFFT1** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
  Fourier-transformed input image 1.
- ▷ **ImageFFT2** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
  Fourier-transformed input image 2.
  Number of elements: ImageFFT2 == ImageFFT1 || ImageFFT2 == 1
- ▷ **ImageCorrelation** (output_object) ..... image(-array) ⇝ *object :* complex
  Result in the frequency domain.

Example

```
* Compute the auto-correlation of an image.
get_image_size(Image,Width,Height)
rft_generic(Image,ImageFFT,'to_freq','none','complex',Width)
correlation_fft(ImageFFT,ImageFFT,Correlation)
rft_generic(Correlation,AutoCorrelation,'from_freq','n','real',Width)
```

Result

correlation_fft returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Predecessors

fft_generic, fft_image, rft_generic

Possible Successors

fft_generic, fft_image_inv, rft_generic

Alternatives

phase_correlation_fft

Module

Foundation

deserialize_fft_optimization_data (: : SerializedItemHandle :)

Deserialize FFT speed optimization data.

deserialize_fft_optimization_data deserializes data, that was serialized by serialize_fft_optimization_data, for optimizing the runtime of the FFT (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized data is defined by the handle SerializedItemHandle. The optimization data must

have been determined previously with optimize_fft_speed and must have been serialized with serialize_fft_optimization_data. If the serialized data has been determined for the image sizes to be used in the application, calling optimize_fft_speed is unnecessary. Note that the data should only be used on the same machine on which they were determined with optimize_fft_speed. Otherwise, the runtimes will not be optimal. Also note that optimization data that were created with Standard HALCON cannot be used with Parallel HALCON and vice versa.

deserialize_fft_optimization_data influences the runtime of the following operators, which use the FFT: fft_generic, fft_image, fft_image_inv, sfs_pentland, sfs_mod_lr, sfs_orig_lrwiener_filter.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

deserialize_fft_optimization_data returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_fft_optimization_data

Possible Successors

fft_generic, fft_image, fft_image_inv, rft_generic, wiener_filter, wiener_filter_ni, photometric_stereo, sfs_pentland, sfs_mod_lr, sfs_orig_lr

Alternatives

optimize_fft_speed, optimize_rft_speed

See also

serialize_fft_optimization_data

Module

Foundation

energy_gabor (ImageGabor, ImageHilbert : Energy : :)

Calculate the energy of a two-channel image.

energy_gabor calculates the local contrast (Energy) of the two input images. The energy of the resulting image is then defined as

$$\text{Energy} = \text{channel1}^2 + \text{channel2}^2 \ .$$

Often the calculation of the energy is preceded by the convolution of an image with a Gabor filter and the Hilbert transform of the Gabor filter (see convol_gabor). In this case, the first channel of the image passed to energy_gabor is the Gabor-filtered image, transformed back into the spatial domain (see fft_image_inv), and the second channel the result of the convolution with the Hilbert transform, also transformed back into the spatial domain. The local energy is a measure for the local contrast of structures (e.g., edges and lines) in the image.

Parameters

▷ **ImageGabor** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / real
1st channel of input image (usually: Gabor image).

▷ **ImageHilbert** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / real
2nd channel of input image (usually: Hilbert image).

▷ **Energy** (output_object) ..... image(-array) ⇝ *object :* real
Image containing the local energy.

Example

```
fft_image(Image,&FFT);
gen_gabor(&Filter,1.4,0.4,1.0,1.5,512);
convol_gabor(FFT,Filter,&Gabor,&Hilbert);
fft_image_inv(Gabor,&GaborInv);
fft_image_inv(Hilbert,&HilbertInv);
energy_gabor(GaborInv,HilbertInv,&Energy);
```

Result

`energy_gabor` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

`gen_gabor`, `convol_gabor`, `fft_image_inv`

Module

Foundation

fft_generic (Image : ImageFFT : Direction, Exponent, Norm, Mode, ResultType :)

Compute the fast Fourier transform of an image.

`fft_generic` computes the fast Fourier transform of the input image `Image`. Because several definitions of the forward and reverse transforms exist in the literature, this operator allows the user to select the most convenient definition.

The general definition of a Fourier transform is as follows:

$$F(m,n) = \frac{1}{c} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} e^{s2\pi i(km/M+ln/N)} f(k,l)$$

Opinions vary on whether the sign s in the exponent should be set to 1 or -1 for the forward transform, i.e., the transform for going to the frequency domain. There is also disagreement on the magnitude of the normalizing factor c. This is sometimes set to 1 for the forward transform, sometimes to MN, and sometimes (in case of the unitary FFT) to $\sqrt{MN}$. Especially in image processing applications the DC term is shifted to the center of the image.

`fft_generic` allows to select these choices individually. The parameter `Direction` allows to select the logical direction of the FFT. (This parameter is not unnecessary; it is needed to discern how to shift the image if the DC term should rest in the center of the image.) Possible values are *'to_freq'* and *'from_freq'*. The parameter

Exponent is used to determine the sign of the exponent. It can be set to 1 or -1. The normalizing factor can be set with Norm, and can take on the values *'none'*, *'sqrt'* and *'n'*. The parameter Mode determines the location of the DC term of the FFT. It can be set to *'dc_center'* or *'dc_edge'*.

In any case, the user must ensure the consistent use of the parameters. This means that the normalizing factors used for the forward and backward transform must yield MN when multiplied, the exponents must be of opposite sign, and Mode must be equal for both transforms.

A consistent combination is, for example *'(to_freq,-1,n,dc_edge)'* for the forward transform and *'(from_freq,1,none,dc_edge)'* for the reverse transform. In this case, the FFT can be interpreted as interpolation with trigonometric basis functions. Another possible combination is *'(to_freq,-1,sqrt,dc_center)'* and *'(from_freq,1,sqrt,dc_center)'*.

The parameter ResultType can be used to specify the result image type of the reverse transform (Direction = *'from_freq'*). In the forward transform (Direction = *'to_freq'*), ResultType must be set to *'complex'*.

Attention

The transformation is always performed for the entire image, i.e., the domain of the image is ignored.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex

Input image.

▷ **ImageFFT** (output_object) ..... image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex

Fourier-transformed image.

▷ **Direction** (input_control) ..... string ⇝ *string*

Calculate forward or reverse transform.
Default: 'to_freq'
List of values: Direction ∈ {'to_freq', 'from_freq'}

▷ **Exponent** (input_control) ..... integer ⇝ *integer*

Sign of the exponent.
Default: -1
List of values: Exponent ∈ {-1, 1}

▷ **Norm** (input_control) ..... string ⇝ *string*

Normalizing factor of the transform.
Default: 'sqrt'
List of values: Norm ∈ {'none', 'sqrt', 'n'}

▷ **Mode** (input_control) ..... string ⇝ *string*

Location of the DC term in the frequency domain.
Default: 'dc_center'
List of values: Mode ∈ {'dc_center', 'dc_edge'}

▷ **ResultType** (input_control) ..... string ⇝ *string*

Image type of the output image.
Default: 'complex'
List of values: ResultType ∈ {'complex', 'byte', 'int1', 'int2', 'uint2', 'int4', 'real', 'direction', 'cyclic'}

Example

```
/* simulation of fft_image */
void my_fft(Hobject In, Hobject *Out)
{
  fft_generic(In,Out,"to_freq",-1,"sqrt","dc_center","complex");
}

/* simulation of fft_image_inv */
void my_fft_image_inv(Hobject In, Hobject *Out)
{
  fft_generic(In,Out,"from_freq",1,"sqrt","dc_center","byte");
}
```

Result

`fft_generic` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system(::'no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

`optimize_fft_speed`, `read_fft_optimization_data`

Possible Successors

`convol_fft`, `correlation_fft`, `phase_correlation_fft`, `convol_gabor`, `convert_image_type`, `power_byte`, `power_real`, `power_ln`, `phase_deg`, `phase_rad`, `energy_gabor`

Alternatives

`fft_image`, `fft_image_inv`, `rft_generic`

Module

Foundation

fft_image (Image : ImageFFT : :)

Compute the fast Fourier transform of an image.

`fft_image` calculates the Fourier transform of the input image (`Image`), i.e., it transforms the image into the frequency domain. The algorithm used is the fast Fourier transform. This corresponds to the call `fft_generic(Image,ImageFFT,'to_freq',-1,'sqrt','dc_center','complex')`.

Attention

The transformation is always performed for the entire image, i.e., the domain of the image is ignored.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / real
Input image.

▷ **ImageFFT** (output_object) . . . . . image(-array) ⇝ *object :* complex
Fourier-transformed image.

Result

`fft_image` returns `2` (`H_MSG_TRUE`) if the input image is of correct type. If the input is empty the behavior can be set via `set_system(::'no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

`optimize_fft_speed`, `read_fft_optimization_data`

Possible Successors

`convol_fft`, `convol_gabor`, `convert_image_type`, `power_byte`, `power_real`, `power_ln`, `phase_deg`, `phase_rad`

Alternatives

fft_generic, rft_generic

See also

fft_image_inv

Module

Foundation

fft_image_inv (Image : ImageFFTInv : :)

Compute the inverse fast Fourier transform of an image.

fft_image_inv calculates the inverse Fourier transform of the input image (Image), i.e., it transforms the image back into the spatial domain. This corresponds to the call fft_generic(Image, ImageFFT, 'from_freq', 1, 'sqrt', 'dc_center', 'byte').

Attention

The transformation is always performed for the entire image, i.e., the domain of the image is ignored.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
  Input image.
- ▷ **ImageFFTInv** (output_object) ..... image(-array) ⇝ *object :* byte
  Inverse-Fourier-transformed image.

Result

fft_image_inv returns 2 (H_MSG_TRUE) if the input image is of correct type. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

convol_fft, convol_gabor, fft_image, optimize_fft_speed, read_fft_optimization_data

Possible Successors

convert_image_type, energy_gabor

Alternatives

fft_generic, rft_generic

See also

fft_image, fft_generic, energy_gabor

Module

Foundation

gen_bandfilter (: ImageFilter : MinFrequency, MaxFrequency, Norm, Mode, Width, Height :)

Generate an ideal band filter.

gen_bandfilter generates an ideal band filter in the frequency domain. The parameters MinFrequency and MaxFrequency determine the cutoff frequencies of the filter as a fraction of the maximum (horizontal and

vertical) frequency that can be represented in an image of size Width × Height, i.e., MinFrequency and MaxFrequency should lie between 0 and 1. To achieve a maximum overall efficiency of the filtering operation, the parameter Norm can be used to specify the normalization factor of the filter. If fft_generic and Norm = *'n'* is used the normalization in the FFT can be avoided. Mode can be used to determine where the DC term of the filter lies or whether the filter should be used in the real-valued FFT. If fft_generic is used, *'dc_edge'* can be used to gain efficiency. If fft_image and fft_image_inv are used for filtering, Norm = *'none'* and Mode = *'dc_center'* must be used. If rft_generic is used, Mode = *'rft'* must be used. The resulting image contains an annulus with the value 0, and a value determined by the normalization outside of this annulus.

Parameters

▷ **ImageFilter** (output_object) ..... image ⇝ *object :* real
Band filter in the frequency domain.

▷ **MinFrequency** (input_control) ..... real ⇝ *real*
Minimum frequency.
Default: 0.1
Suggested values: MinFrequency ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: MinFrequency >= 0

▷ **MaxFrequency** (input_control) ..... real ⇝ *real*
Maximum frequency.
Default: 0.2
Suggested values: MaxFrequency ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: MaxFrequency >= 0 && MaxFrequency >= MinFrequency

▷ **Norm** (input_control) ..... string ⇝ *string*
Normalizing factor of the filter.
Default: 'none'
List of values: Norm ∈ {'none', 'n'}

▷ **Mode** (input_control) ..... string ⇝ *string*
Location of the DC term in the frequency domain.
Default: 'dc_center'
List of values: Mode ∈ {'dc_center', 'dc_edge', 'rft'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the image (filter).
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}

▷ **Height** (input_control) ..... integer ⇝ *integer*
Height of the image (filter).
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Example

```
* Filtering with maximum efficiency with fft_generic.
gen_bandpass(Bandfilter,0.2,0.4,'n','dc_edge',Width,Height)
fft_generic(Image,ImageFFT,'to_freq',-1,'none','dc_edge','complex')
convol_fft(ImageFFT,Bandfilter,ImageConvol)
fft_generic(ImageConvol,ImageResult,'from_freq',1,'none','dc_edge','byte')
```

Result

gen_bandfilter returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

convol_fft

Alternatives

gen_circle, paint_region

See also

gen_highpass, gen_lowpass, gen_bandpass, gen_gauss_filter, gen_mean_filter, gen_derivative_filter

Module

Foundation

gen_bandpass (: ImageBandpass : MinFrequency, MaxFrequency, Norm, Mode, Width, Height :)

Generate an ideal bandpass filter.

gen_bandpass generates an ideal bandpass filter in the frequency domain. The parameters MinFrequency and MaxFrequency determine the cutoff frequencies of the filter as a fraction of the maximum (horizontal and vertical) frequency that can be represented in an image of size Width × Height, i.e., MinFrequency and MaxFrequency should lie between 0 and 1. To achieve a maximum overall efficiency of the filtering operation, the parameter Norm can be used to specify the normalization factor of the filter. If fft_generic and Norm = *'n'* is used the normalization in the FFT can be avoided. Mode can be used to determine where the DC term of the filter lies or whether the filter should be used in the real-valued FFT. If fft_generic is used, *'dc_edge'* can be used to gain efficiency. If fft_image and fft_image_inv are used for filtering, Norm = *'none'* and Mode = *'dc_center'* must be used. If rft_generic is used, Mode = *'rft'* must be used. The resulting image contains an annulus with the value 255, and the value 0 outside of this annulus.

Parameters

▷ **ImageBandpass** (output_object) ..... image ⇝ *object :* real
Bandpass filter in the frequency domain.

▷ **MinFrequency** (input_control) ..... real ⇝ *real*
Minimum frequency.
Default: 0.1
Suggested values: MinFrequency ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: MinFrequency >= 0

▷ **MaxFrequency** (input_control) ..... real ⇝ *real*
Maximum frequency.
Default: 0.2
Suggested values: MaxFrequency ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: MaxFrequency >= 0 && MaxFrequency >= MinFrequency

▷ **Norm** (input_control) ..... string ⇝ *string*
Normalizing factor of the filter.
Default: 'none'
List of values: Norm ∈ {'none', 'n'}

▷ **Mode** (input_control) ..... string ⇝ *string*
Location of the DC term in the frequency domain.
Default: 'dc_center'
List of values: Mode ∈ {'dc_center', 'dc_edge', 'rft'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the image (filter).
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}

▷ **Height** (input_control) ..... integer ⇝ *integer*
Height of the image (filter).
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Example

```
* Filtering with maximum efficiency with fft_generic.
```

```
gen_bandpass(Bandpass,0.2,0.4,'n','dc_edge',Width,Height)
fft_generic(Image,ImageFFT,'to_freq',-1,'none','dc_edge','complex')
convol_fft(ImageFFT,Bandpass,ImageConvol)
fft_generic(ImageConvol,ImageResult,'from_freq',1,'none','dc_edge','byte')
```

Result

`gen_bandpass` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`convol_fft`

See also

`gen_highpass`, `gen_lowpass`, `gen_bandfilter`, `gen_gauss_filter`, `gen_mean_filter`, `gen_derivative_filter`

Module

Foundation

gen_derivative_filter (: ImageDerivative : Derivative, Exponent, Norm, Mode, Width, Height :)

Generate a derivative filter in the frequency domain.

`gen_derivative_filter` generates a derivative filter in the frequency domain. The derivative to be computed is determined by `Derivative`. `Exponent` specifies the exponent used in the reverse transform. It must be set to the same value that is used in `fft_generic`. If `fft_image_inv` is used in the reverse transform, `Exponent` = *1* must be used. However, since the derivative image typically contains negative values, `fft_generic` should always be used for the reverse transform. To achieve a maximum overall efficiency of the filtering operation, the parameter `Norm` can be used to specify the normalization factor of the filter. If `fft_generic` and `Norm` = *'n'* is used the normalization in the FFT can be avoided. `Mode` can be used to determine where the DC term of the filter lies or whether the filter should be used in the real-valued FFT. If `fft_generic` is used, *'dc_edge'* can be used to gain efficiency. If `fft_image` and `fft_image_inv` are used for filtering, `Norm` = *'none'* and `Mode` = *'dc_center'* must be used. If `rft_generic` is used, `Mode` = *'rft'* must be used.

Parameters

▷ **ImageDerivative** (output_object) ..... image ⇝ *object : complex*
Derivative filter as image in the frequency domain.

▷ **Derivative** (input_control) ..... string ⇝ *string*
Derivative to be computed.
Default: 'x'
Suggested values: `Derivative` ∈ {'x', 'y', 'xx', 'xy', 'yy', 'xxx', 'xxy', 'xyy', 'yyy'}

▷ **Exponent** (input_control) ..... integer ⇝ *integer*
Exponent used in the reverse transform.
Default: 1
Suggested values: `Exponent` ∈ {-1, 1}

▷ **Norm** (input_control) ..... string ⇝ *string*
Normalizing factor of the filter.
Default: 'none'
List of values: `Norm` ∈ {'none', 'n'}

▷ **Mode** (input_control) ..... string ⇝ *string*
Location of the DC term in the frequency domain.
Default: 'dc_center'
List of values: `Mode` ∈ {'dc_center', 'dc_edge', 'rft'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the image (filter).
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}

▷ **Height** (input_control) ..... integer ⇝ *integer*
Height of the image (filter).
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Example

```
* Generate a smoothed derivative filter.
gen_gauss_filter (ImageGauss, Sigma, Sigma, 0, 'n', 'dc_edge', 512, 512)
convert_image_type (ImageGauss, ImageGaussComplex, 'complex')
gen_derivative_filter (ImageDerivX, 'x', 1, 'none', 'dc_edge', 512, 512)
mult_image (ImageGaussComplex, ImageDerivX, ImageDerivXGauss, 1, 0)
* Filter an image with the smoothed derivative filter.
fft_generic (Image, ImageFFT, 'to_freq', -1, 'none', 'dc_edge', 'complex')
convol_fft (ImageFFT, ImageDerivXGauss, Filtered)
fft_generic (Filtered, ImageX, 'from_freq', 1, 'none', 'dc_edge', 'real')
```

Result

gen_derivative_filter returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fft_image, fft_generic, rft_generic

Possible Successors

convol_fft

See also

fft_image_inv, gen_gauss_filter, gen_mean_filter, gen_lowpass, gen_bandpass, gen_bandfilter, gen_highpass

Module

Foundation

gen_filter_mask (: ImageFilter : FilterMask, Scale, Width, Height :)

Store a filter mask in the spatial domain as a real-image.

gen_filter_mask stores a filter mask in the spatial domain as a real-image. The center of the filter mask lies in the center of the resulting image. The parameter Scale determines by which amount the values of the filter mask are multiplied (this results in larger values of the Fourier transform of the filter). The corresponding filter matrix, which is given in FilterMask can be generated either from a file or a tuple. The format of the filter matrix is described with the operator convol_image. Example filter masks can be found in the directory "filter" in the HALCON home directory. This operator is useful for visualizing the frequency response of filter masks (by applying a Fourier transform to the result image of this operator).

Parameters

▷ **ImageFilter** (output_object) . . . . . image ⇝ *object :* real
Filter in the spatial domain.

▷ **FilterMask** (input_control) . . . . . filename.read(-array) ⇝ *string* / integer
Filter mask as file name or tuple.
Default: 'gauss'
Suggested values: FilterMask ∈ {'gauss', 'laplace4', 'laplace8', 'lowpas_3_3', 'lowpas_5_5', 'lowpas_7_7', 'lowpas_9_9', 'sobel_c', 'sobel_l'}

▷ **Scale** (input_control) . . . . . real ⇝ *real*
Scaling factor.
Default: 1.0
Suggested values: Scale ∈ {0.3, 0.5, 0.75, 1.0, 1.25, 1.5, 2.0}
Minimum increment: 0.001
Recommended increment: 0.1

▷ **Width** (input_control) . . . . . integer ⇝ *integer*
Width of the image (filter).
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}

▷ **Height** (input_control) . . . . . integer ⇝ *integer*
Height of the image (filter).
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Example

```
* If the filter should be read from a file:
gen_filter_mask (Filter, 'lowpas_3_3', 1.0, 512, 512)
* If the filter should be directly passed as a tuple:
gen_filter_mask (Filter, [3,3,9,1,1,1,1,1,1,1,1,1], 1.0, 512, 512)
fft_image (Filter, FilterFFT)
dev_set_paint ('3d_plot')
dev_display (FilterFFT)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fft_image, fft_generic

See also

convol_image

Module

Foundation

gen_gabor (: ImageFilter : Angle, Frequency, Bandwidth, Orientation, Norm, Mode, Width, Height :)

Generate a Gabor filter.

gen_gabor generates a Gabor filter with a user-definable bandpass frequency range and sign for the Hilbert transform. This is done by calculating a symmetrical filter in the frequency domain, which can be adapted by the parameters Angle, Frequency, Bandwidth and Orientation such that a certain frequency band and a certain direction range in the spatial domain is filtered out in the frequency domain.

The parameters Frequency (central frequency = distance from the DC term) and Orientation (direction) determine the center of the filter. Larger values of Frequency result in higher frequencies being passed. A value of 0 for Orientation generates a horizontally oriented "crescent" (the bulge of the crescent points upward). Higher values of Orientation result in the counterclockwise rotation of the crescent.

The parameters Angle and Bandwidth are used to determine the range of frequencies and angles being passed by the filter. The larger Angle is, the smaller the range of angles passed by the filter gets (because the "crescent" gets narrower). The larger Bandwidth is, the smaller the frequency band being passed gets (because the "crescent" gets thinner).

To achieve a maximum overall efficiency of the filtering operation, the parameter Norm can be used to specify the normalization factor of the filter. If fft_generic and Norm = *'n'* is used the normalization in the FFT can be avoided. Mode can be used to determine where the DC term of the filter lies. If fft_generic is used, *'dc_edge'* can be used to gain efficiency. If fft_image and fft_image_inv are used for filtering, Norm = *'none'* and Mode = *'dc_center'* must be used. Note that gen_gabor cannot create a filter that can be used with rft_generic.

The resulting image is a two-channel real-image, containing the Gabor filter in the first channel and the corresponding Hilbert filter in the second channel.

Parameters

▷ **ImageFilter** (output_object) ..... multichannel-image ⇝ *object :* real
Gabor and Hilbert filter.

▷ **Angle** (input_control) ..... real ⇝ *real*
Angle range, inversely proportional to the range of orientations.
Default: 1.4
Suggested values: Angle ∈ {1.0, 1.2, 1.4, 1.6, 2.0, 2.5, 3.0, 5.0, 6.0, 10.0, 20.0, 30.0, 50.0, 70.0, 100.0}
Value range: 1.0 ≤ Angle ≤ 500.0
Minimum increment: 0.001
Recommended increment: 0.1

▷ **Frequency** (input_control) ..... real ⇝ *real*
Distance of the center of the filter to the DC term.
Default: 0.4
Suggested values: Frequency ∈ {0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.50, 0.55, 0.60, 0.65, 0.699}
Value range: 0.0 ≤ Frequency ≤ 0.7
Minimum increment: 0.00001
Recommended increment: 0.005

▷ **Bandwidth** (input_control) ..... real ⇝ *real*
Bandwidth range, inversely proportional to the range of frequencies being passed.
Default: 1.0
Suggested values: Bandwidth ∈ {0.1, 0.3, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0, 7.0, 10.0, 15.0, 20.0, 30.0, 50.0}
Value range: 0.05 ≤ Bandwidth ≤ 100.0
Minimum increment: 0.001
Recommended increment: 0.1

▷ **Orientation** (input_control) ..... real ⇝ *real*
Angle of the principal orientation.
Default: 1.5
Suggested values: Orientation ∈ {0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.2, 2.4, 2.6, 2.8, 3.0, 3.14}
Value range: 0.0 ≤ Orientation ≤ 3.1416
Minimum increment: 0.0001
Recommended increment: 0.05

▷ **Norm** (input_control) ..... string ⇝ *string*
Normalizing factor of the filter.
Default: 'none'
List of values: Norm ∈ {'none', 'n'}

▷ **Mode** (input_control) ..... string ⇝ *string*
Location of the DC term in the frequency domain.
Default: 'dc_center'
List of values: Mode ∈ {'dc_center', 'dc_edge'}

▷ **Width** (input_control) . . . . . integer ⇝ *integer*
Width of the image (filter).
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}

▷ **Height** (input_control) . . . . . integer ⇝ *integer*
Height of the image (filter).
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Example

```
gen_gabor(Filter,1.4,0.4,1.0,1.5,'n','dc_edge',512,512)
fft_generic(Image,ImageFFT,'to_freq',-1,'none','dc_edge','complex')
convol_gabor(ImageFFT,Filter,Gabor,Hilbert)
fft_generic(Gabor,GaborInv,'from_freq',1,'none','dc_edge','byte')
fft_generic(Hilbert,HilbertInv,'from_freq',1,'none','dc_edge','byte')
energy_gabor(GaborInv,HilbertInv,Energy)
```

Result

gen_gabor returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fft_image, fft_generic

Possible Successors

convol_gabor

Alternatives

gen_bandpass, gen_bandfilter, gen_highpass, gen_lowpass

See also

fft_image_inv, energy_gabor

Module

Foundation

gen_gauss_filter (: ImageGauss : Sigma1, Sigma2, Phi, Norm, Mode, Width, Height :)

Generate a Gaussian filter in the frequency domain.

gen_gauss_filter generates a (possibly anisotropic) Gaussian filter in the frequency domain. The standard deviations (i.e., the amount of smoothing) of the Gaussian in the spatial domain are determined by Sigma1 and Sigma2. Sigma1 is the standard deviation in the principal direction of the filter in the spatial domain determined by the angle Phi. To achieve a maximum overall efficiency of the filtering operation, the parameter Norm can be used to specify the normalization factor of the filter. If fft_generic and Norm = *'n'* is used the normalization in the FFT can be avoided. Mode can be used to determine where the DC term of the filter lies or whether the filter should be used in the real-valued FFT. If fft_generic is used, *'dc_edge'* can be used to gain efficiency. If fft_image and fft_image_inv are used for filtering, Norm = *'none'* and Mode = *'dc_center'* must be used. If rft_generic is used, Mode = *'rft'* must be used.

Parameters

▷ **ImageGauss** (output_object) . . . . . image ⇝ *object :* real
Gaussian filter as image in the frequency domain.

▷ **Sigma1** (input_control) . . . . . real ⇝ *real*
Standard deviation of the Gaussian in the principal direction of the filter in the spatial domain.
Default: 1.0
Suggested values: `Sigma1` ∈ {0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0}
Restriction: Sigma1 >= 0

▷ **Sigma2** (input_control) . . . . . real ⇝ *real*
Standard deviation of the Gaussian perpendicular to the principal direction of the filter in the spatial domain.
Default: 1.0
Suggested values: `Sigma2` ∈ {0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0}
Restriction: Sigma2 >= 0

▷ **Phi** (input_control) . . . . . angle.rad ⇝ *real*
Principal direction of the filter in the spatial domain.
Default: 0.0
Suggested values: `Phi` ∈ {0.0, 0.523599, 0.785398, 1.047198, 1.570796, 2.094395, 2.356194, 2.617994, 3.141593}

▷ **Norm** (input_control) . . . . . string ⇝ *string*
Normalizing factor of the filter.
Default: 'none'
List of values: `Norm` ∈ {'none', 'n'}

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Location of the DC term in the frequency domain.
Default: 'dc_center'
List of values: `Mode` ∈ {'dc_center', 'dc_edge', 'rft'}

▷ **Width** (input_control) . . . . . integer ⇝ *integer*
Width of the image (filter).
Default: 512
Suggested values: `Width` ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}

▷ **Height** (input_control) . . . . . integer ⇝ *integer*
Height of the image (filter).
Default: 512
Suggested values: `Height` ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Example

```
* Generate a smoothed derivative filter.
gen_gauss_filter (ImageGauss, Sigma, Sigma, 0, 'n', 'dc_edge', 512, 512)
convert_image_type (ImageGauss, ImageGaussComplex, 'complex')
gen_derivative_filter (ImageDerivX, 'x', 1, 'none', 'dc_edge', 512, 512)
mult_image (ImageGaussComplex, ImageDerivX, ImageDerivXGauss, 1, 0)
* Filter an image with the smoothed derivative filter.
fft_generic (Image, ImageFFT, 'to_freq', -1, 'none', 'dc_edge', 'complex')
convol_fft (ImageFFT, ImageDerivXGauss, Filtered)
fft_generic (Filtered, ImageX, 'from_freq', 1, 'none', 'dc_edge', 'real')
```

Result

`gen_gauss_filter` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fft_image, fft_generic, rft_generic

Possible Successors

convol_fft

See also

fft_image_inv, gen_mean_filter, gen_derivative_filter, gen_lowpass, gen_bandpass, gen_bandfilter, gen_highpass

Module

Foundation

gen_highpass (: ImageHighpass : Frequency, Norm, Mode, Width, Height :)

Generate an ideal highpass filter.

gen_highpass generates an ideal highpass filter in the frequency domain. The parameter Frequency determines the cutoff frequency of the filter as a fraction of the maximum (horizontal and vertical) frequency that can be represented in an image of size Width × Height, i.e., Frequency should lie between 0 and 1. To achieve a maximum overall efficiency of the filtering operation, the parameter Norm can be used to specify the normalization factor of the filter. If fft_generic and Norm = *'n'* is used the normalization in the FFT can be avoided. Mode can be used to determine where the DC term of the filter lies or whether the filter should be used in the real-valued FFT. If fft_generic is used, *'dc_edge'* can be used to gain efficiency. If fft_image and fft_image_inv are used for filtering, Norm = *'none'* and Mode = *'dc_center'* must be used. If rft_generic is used, Mode = *'rft'* must be used. The resulting image has an inner part with the value 0, and an outer part with the value determined by the normalization factor.

Parameters

- ▷ **ImageHighpass** (output_object) ..... image ⇝ *object :* real
  Highpass filter in the frequency domain.
- ▷ **Frequency** (input_control) ..... real ⇝ *real*
  Cutoff frequency.
  Default: 0.1
  Suggested values: Frequency ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
  Restriction: Frequency >= 0
- ▷ **Norm** (input_control) ..... string ⇝ *string*
  Normalizing factor of the filter.
  Default: 'none'
  List of values: Norm ∈ {'none', 'n'}
- ▷ **Mode** (input_control) ..... string ⇝ *string*
  Location of the DC term in the frequency domain.
  Default: 'dc_center'
  List of values: Mode ∈ {'dc_center', 'dc_edge', 'rft'}
- ▷ **Width** (input_control) ..... integer ⇝ *integer*
  Width of the image (filter).
  Default: 512
  Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}
- ▷ **Height** (input_control) ..... integer ⇝ *integer*
  Height of the image (filter).
  Default: 512
  Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Example

```
* Filtering with maximum efficiency with fft_generic.
gen_highpass(Highpass,0.2,'n','dc_edge',Width,Height)
fft_generic(Image,ImageFFT,'to_freq',-1,'none','dc_edge','complex')
convol_fft(ImageFFT,Highpass,ImageConvol)
fft_generic(ImageConvol,ImageResult,'from_freq',1,'none','dc_edge','byte')
```

Result

gen_highpass returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

convol_fft

See also

convol_fft, gen_lowpass, gen_bandpass, gen_bandfilter, gen_gauss_filter, gen_mean_filter, gen_derivative_filter

Module

Foundation

gen_lowpass (: ImageLowpass : Frequency, Norm, Mode, Width, Height :)

Generate an ideal lowpass filter.

gen_lowpass generates an ideal lowpass filter in the frequency domain. The parameter Frequency determines the cutoff frequency of the filter as a fraction of the maximum (horizontal and vertical) frequency that can be represented in an image of size Width × Height, i.e., Frequency should lie between 0 and 1. To achieve a maximum overall efficiency of the filtering operation, the parameter Norm can be used to specify the normalization factor of the filter. If fft_generic and Norm = *'n'* is used the normalization in the FFT can be avoided. Mode can be used to determine where the DC term of the filter lies or whether the filter should be used in the real-valued FFT. If fft_generic is used, *'dc_edge'* can be used to gain efficiency. If fft_image and fft_image_inv are used for filtering, Norm = *'none'* and Mode = *'dc_center'* must be used. If rft_generic is used, Mode = *'rft'* must be used. The resulting image has an inner part with the value set to the normalization factor, and an outer part with the value 0.

Parameters

▷ **ImageLowpass** (output_object) ..... image ⇝ *object :* real
Lowpass filter in the frequency domain.

▷ **Frequency** (input_control) ..... real ⇝ *real*
Cutoff frequency.
Default: 0.1
Suggested values: Frequency ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: Frequency >= 0

▷ **Norm** (input_control) ..... string ⇝ *string*
Normalizing factor of the filter.
Default: 'none'
List of values: Norm ∈ {'none', 'n'}

▷ **Mode** (input_control) ..... string ⇝ *string*
Location of the DC term in the frequency domain.
Default: 'dc_center'
List of values: Mode ∈ {'dc_center', 'dc_edge', 'rft'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the image (filter).
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}

▷ **Height** (input_control) ..... integer ⇝ *integer*
Height of the image (filter).
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Example

```
* Filtering with maximum efficiency with fft_generic.
gen_lowpass(Lowpass,0.2,'n','dc_edge',Width,Height)
fft_generic(Image,ImageFFT,'to_freq',-1,'none','dc_edge','complex')
convol_fft(ImageFFT,Lowpass,ImageConvol)
fft_generic(ImageConvol,ImageResult,'from_freq',1,'none','dc_edge','byte')
```

Result

`gen_lowpass` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`convol_fft`

See also

`gen_highpass`, `gen_bandpass`, `gen_bandfilter`, `gen_gauss_filter`, `gen_mean_filter`, `gen_derivative_filter`

Module

Foundation

gen_mean_filter (: ImageMean : MaskShape, Diameter1, Diameter2, Phi, Norm, Mode, Width, Height :)

Generate a mean filter in the frequency domain.

`gen_mean_filter` generates a mean filter in the frequency domain. The shape of the mean filter is determined by `MaskShape`. For `MaskShape` = *'rectangle'*, a rectangular mean filter is generated. For `MaskShape` = *'ellipse'*, an elliptical mean filter is generated. The diameters (i.e., the amount of smoothing) of the mean filter in the spatial domain are determined by `Diameter1` and `Diameter2`. `Diameter1` is the diameter in the principal direction of the filter in the spatial domain determined by the angle `Phi`. To achieve a maximum overall efficiency of the filtering operation, the parameter `Norm` can be used to specify the normalization factor of the filter. If `fft_generic` and `Norm` = *'n'* is used the normalization in the FFT can be avoided. `Mode` can be used to determine where the DC term of the filter lies or whether the filter should be used in the real-valued FFT. If `fft_generic` is used, *'dc_edge'* can be used to gain efficiency. If `fft_image` and `fft_image_inv` are used for filtering, `Norm` = *'none'* and `Mode` = *'dc_center'* must be used. If `rft_generic` is used, `Mode` = *'rft'* must be used.

Parameters

▷ **ImageMean** (output_object) ..... image ⇝ *object :* real
Mean filter as image in the frequency domain.

▷ **MaskShape** (input_control) ..... string ⇝ *string*
Shape of the filter mask in the spatial domain.
Default: 'ellipse'
List of values: `MaskShape` ∈ {'ellipse', 'rectangle'}

▷ **Diameter1** (input_control) ..... real ⇝ *real*
Diameter of the mean filter in the principal direction of the filter in the spatial domain.
Default: 11.0
Suggested values: `Diameter1` ∈ {3.0, 5.0, 7.0, 9.0, 11.0, 15.0, 21.0, 31.0, 51.0}
Restriction: Diameter1 > 0

▷ **Diameter2** (input_control) ..... real ⇝ *real*
Diameter of the mean filter perpendicular to the principal direction of the filter in the spatial domain.
Default: 11.0
Suggested values: `Diameter2` ∈ {3.0, 5.0, 7.0, 9.0, 11.0, 15.0, 21.0, 31.0, 51.0}
Restriction: Diameter2 > 0

▷ **Phi** (input_control) ..... angle.rad ⇝ *real*
Principal direction of the filter in the spatial domain.
Default: 0.0
Suggested values: Phi ∈ {0.0, 0.523599, 0.785398, 1.047198, 1.570796, 2.094395, 2.356194, 2.617994, 3.141593}

▷ **Norm** (input_control) ..... string ⇝ *string*
Normalizing factor of the filter.
Default: 'none'
List of values: Norm ∈ {'none', 'n'}

▷ **Mode** (input_control) ..... string ⇝ *string*
Location of the DC term in the frequency domain.
Default: 'dc_center'
List of values: Mode ∈ {'dc_center', 'dc_edge', 'rft'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the image (filter).
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}

▷ **Height** (input_control) ..... integer ⇝ *integer*
Height of the image (filter).
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Example

```
* Generate a circular mean filter.
gen_mean_filter (FilterMean, 'ellipse', 15, 15, 0, 'n', 'dc_edge', 512, 512)
* Filter an image with the circular mean filter.
fft_generic (Image, ImageFFT, 'to_freq', -1, 'none', 'dc_edge', 'complex')
convol_fft (ImageFFT, FilterMean, Filtered)
fft_generic (Filtered, ImageMean, 'from_freq', 1, 'none', 'dc_edge', 'real')
```

Result

gen_mean_filter returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fft_image, fft_generic, rft_generic

Possible Successors

convol_fft

See also

fft_image_inv, gen_gauss_filter, gen_derivative_filter, gen_lowpass, gen_bandpass, gen_bandfilter, gen_highpass

Module

Foundation

gen_sin_bandpass (: ImageFilter : Frequency, Norm, Mode, Width, Height :)

Generate a bandpass filter with sinusoidal shape.

gen_sin_bandpass generates a rotationally invariant bandpass filter with the response being a sinusoidal function in the frequency domain. The maximum of the sine is determined by Frequency, which is given as a fraction

of the maximum (horizontal and vertical) frequency that can be represented in an image of size Width × Height, i.e., Frequency should lie between 0 and 1. To achieve a maximum overall efficiency of the filtering operation, the parameter Norm can be used to specify the normalization factor of the filter. If fft_generic and Norm = *'n'* is used the normalization in the FFT can be avoided. Mode can be used to determine where the DC term of the filter lies or whether the filter should be used in the real-valued FFT. If fft_generic is used, *'dc_edge'* can be used to gain efficiency. If fft_image and fft_image_inv are used for filtering, Norm = *'none'* and Mode = *'dc_center'* must be used. If rft_generic is used, Mode = *'rft'* must be used. The filter is always zero for the DC term, rises with the sine function up to Frequency, and drops for higher frequencies accordingly. The range of the sine used is from 0 to π. All other points are set to zero.

Parameters

- ▷ **ImageFilter** (output_object) ..... image ⇝ *object : real*
  Bandpass filter as image in the frequency domain.
- ▷ **Frequency** (input_control) ..... real ⇝ *real*
  Distance of the filter's maximum from the DC term.
  Default: 0.1
  Suggested values: Frequency ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
  Restriction: Frequency >= 0
- ▷ **Norm** (input_control) ..... string ⇝ *string*
  Normalizing factor of the filter.
  Default: 'none'
  List of values: Norm ∈ {'none', 'n'}
- ▷ **Mode** (input_control) ..... string ⇝ *string*
  Location of the DC term in the frequency domain.
  Default: 'dc_center'
  List of values: Mode ∈ {'dc_center', 'dc_edge', 'rft'}
- ▷ **Width** (input_control) ..... integer ⇝ *integer*
  Width of the image (filter).
  Default: 512
  Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}
- ▷ **Height** (input_control) ..... integer ⇝ *integer*
  Height of the image (filter).
  Default: 512
  Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Result

gen_sin_bandpass returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fft_image, fft_generic, rft_generic

Possible Successors

convol_fft

Alternatives

gen_std_bandpass

See also

fft_image_inv, gen_gauss_filter, gen_mean_filter, gen_derivative_filter, gen_bandpass, gen_bandfilter, gen_highpass, gen_lowpass

Module

Foundation

gen_std_bandpass (: ImageFilter : Frequency, Sigma, Type, Norm, Mode, Width, Height :)

Generate a bandpass filter with Gaussian or sinusoidal shape.

gen_std_bandpass generates a rotationally invariant bandpass filter with the response being determined by the parameters Frequency and Sigma: Frequency determines the location of the maximum response with respect to the DC term, while Sigma determines the width of the frequency band that passes the filter. Frequency and Sigma are specified as a fraction of the maximum (horizontal and vertical) frequency that can be represented in an image of size Width × Height. Frequency should lie between 0 and 1. For Type = *'gauss'*, a Gaussian response is generated with Sigma being the standard deviation. For Type = *'sin'*, a sine function is generated with the maximum at Frequency and the extent Sigma. To achieve a maximum overall efficiency of the filtering operation, the parameter Norm can be used to specify the normalization factor of the filter. If fft_generic and Norm = *'n'* is used the normalization in the FFT can be avoided. Mode can be used to determine where the DC term of the filter lies or whether the filter should be used in the real-valued FFT. If fft_generic is used, *'dc_edge'* can be used to gain efficiency. If fft_image and fft_image_inv are used for filtering, Norm = *'none'* and Mode = *'dc_center'* must be used. If rft_generic is used, Mode = *'rft'* must be used.

Parameters

▷ **ImageFilter** (output_object) ..... image ⇝ *object : real*
Bandpass filter as image in the frequency domain.

▷ **Frequency** (input_control) ..... real ⇝ *real*
Distance of the filter's maximum from the DC term.
Default: 0.1
Suggested values: Frequency ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: Frequency >= 0

▷ **Sigma** (input_control) ..... real ⇝ *real*
Bandwidth of the filter (standard deviation).
Default: 0.01
Suggested values: Sigma ∈ {0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 1.0}
Restriction: Sigma >= 0

▷ **Type** (input_control) ..... string ⇝ *string*
Filter type.
Default: 'sin'
List of values: Type ∈ {'sin', 'gauss'}

▷ **Norm** (input_control) ..... string ⇝ *string*
Normalizing factor of the filter.
Default: 'none'
List of values: Norm ∈ {'none', 'n'}

▷ **Mode** (input_control) ..... string ⇝ *string*
Location of the DC term in the frequency domain.
Default: 'dc_center'
List of values: Mode ∈ {'dc_center', 'dc_edge', 'rft'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the image (filter).
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048, 4096, 8192}

▷ **Height** (input_control) ..... integer ⇝ *integer*
Height of the image (filter).
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048, 4096, 8192}

Result

gen_std_bandpass returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fft_image, fft_generic, rft_generic

Possible Successors

convol_fft

Alternatives

gen_sin_bandpass

See also

fft_image_inv, gen_gauss_filter, gen_mean_filter, gen_derivative_filter, gen_bandpass, gen_bandfilter, gen_highpass, gen_lowpass

Module

Foundation

optimize_fft_speed (: : Width, Height, Mode :)

Optimize the runtime of the FFT.

optimize_fft_speed determines a method that achieves an optimum runtime of the FFT for an image of size Width × Height. The data that are determined for one image size do not influence the methods used for other image sizes. Consequently, optimize_fft_speed can be called multiple times with different values for Width and Height to achieve an optimum runtime for all image sizes that are used in an application. The parameter Mode determines the thoroughness of the search for the fastest method. For Mode = *'standard'* a fast search is used, which typically takes a few seconds. The method thus determined results in very good runtimes, which are not always optimal. For Mode = *'patient'* a more thorough search is performed, which typically takes several seconds and in most cases leads to optimum runtimes. For Mode = *'exhaustive'* an exhaustive search is performed, which typically takes several minutes and always results in the optimum runtime. In most applications, Mode = *'standard'* results in the best compromise between the runtime of the FFT and the time required for the search of the optimum runtime. The data determined with optimize_fft_speed can be saved with write_fft_optimization_data and can be loaded with read_fft_optimization_data.

Please note that this optimization is performed for the particular computer on which the operator is called. The results are not suited to be transferred and used on other computers unless they have the same hardware and software configuration including the driver versions.

optimize_fft_speed influences the runtime of the following operators, which use the FFT: fft_generic, fft_image, fft_image_inv, photometric_stereo, sfs_pentland, sfs_mod_lr, sfs_orig_lr wiener_filter.

Parameters

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the image for which the runtime should be optimized.
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048}

▷ **Height** (input_control) ..... integer ⇝ *integer*
Height of the image for which the runtime should be optimized.
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048}

▷ **Mode** (input_control) ..... string ⇝ *string*
Thoroughness of the search for the optimum runtime.
Default: 'standard'
List of values: Mode ∈ {'standard', 'patient', 'exhaustive'}

Result

optimize_fft_speed returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fft_generic, fft_image, fft_image_inv, wiener_filter, wiener_filter_ni, photometric_stereo, sfs_pentland, sfs_mod_lr, sfs_orig_lr, write_fft_optimization_data

Alternatives

read_fft_optimization_data

See also

optimize_rft_speed

Module

Foundation

optimize_rft_speed (: : Width, Height, Mode :)

Optimize the runtime of the real-valued FFT.

optimize_rft_speed determines a method that achieves an optimum runtime of the real-valued FFT for an image of size Width × Height. The data that are determined for one image size do not influence the methods used for other image sizes. Consequently, optimize_rft_speed can be called multiple times with different values for Width and Height to achieve an optimum runtime for all image sizes that are used in an application. The parameter Mode determines the thoroughness of the search for the fastest method. For Mode = *'standard'* a fast search is used, which typically takes a few seconds. The method thus determined results in very good runtimes, which are not always optimal. For Mode = *'patient'* a more thorough search is performed, which typically takes several seconds and in most cases leads to optimum runtimes. For Mode = *'exhaustive'* an exhaustive search is performed, which typically takes several minutes and always results in the optimum runtime. In most applications, Mode = *'standard'* results in the best compromise between the runtime of the real-valued FFT and the time required for the search of the optimum runtime. The data determined with optimize_rft_speed can be saved with write_fft_optimization_data and can be loaded with read_fft_optimization_data.

Please note that this optimization is performed for the particular computer on which the operator is called. The results are not suited to be transferred and used on other computers unless they have the same hardware and software configuration including the driver versions.

optimize_rft_speed influences the runtime of rft_generic.

Parameters

▷ **Width** (input_control) . . . integer ⇝ *integer*
Width of the image for which the runtime should be optimized.
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048}

▷ **Height** (input_control) . . . integer ⇝ *integer*
Height of the image for which the runtime should be optimized.
Default: 512
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576, 1024, 2048}

▷ **Mode** (input_control) . . . string ⇝ *string*
Thoroughness of the search for the optimum runtime.
Default: 'standard'
List of values: Mode ∈ {'standard', 'patient', 'exhaustive'}

Result

optimize_rft_speed returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

rft_generic, write_fft_optimization_data

Alternatives

read_fft_optimization_data

See also

optimize_fft_speed

Module

Foundation

phase_correlation_fft (ImageFFT1, ImageFFT2 : ImagePhaseCorrelation : :)

Compute the phase correlation of two images in the frequency domain.

phase_correlation_fft calculates the phase correlation of the Fourier-transformed input images in the frequency domain. The phase correlation is calculated by multiplying ImageFFT1 with the complex conjugate of ImageFFT2 and dividing by the absolute value of this product. It should be noted that in order to achieve a correct scaling of the phase correlation in the spatial domain, the operators fft_generic or rft_generic with Norm = *'none'* must be used for the forward transform and fft_generic or rft_generic with Norm = *'n'* for the reverse transform. If ImageFFT1 and ImageFFT2 contain the same number of images, the corresponding images are phase-correlated pairwise. Otherwise, ImageFFT2 must contain only one single image. In this case, the phase correlation is performed for each image of ImageFFT1 with ImageFFT2 .

Attention

The filtering is always performed on the entire image, i.e., the domain of the image is ignored.

Parameters

▷ **ImageFFT1** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
Fourier-transformed input image 1.

▷ **ImageFFT2** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
Fourier-transformed input image 2.
Number of elements: ImageFFT2 == ImageFFT1 || ImageFFT2 == 1

▷ **ImagePhaseCorrelation** (output_object) ..... image(-array) ⇝ *object :* complex
Phase correlation of the input images in the frequency domain.

Example

```
* Compute the phase correlation of two images.
get_image_size(Image1,Width,Height)
rft_generic(Image1,ImageFFT1,'to_freq','none','complex',Width)
rft_generic(Image2,ImageFFT2,'to_freq','none','complex',Width)
phase_correlation_fft(ImageFFT1,ImageFFT2,PhaseCorrelationFFT)
rft_generic(PhaseCorrelationFFT,PhaseCorrelation,'from_freq','n', \
            'real',Width)
* Determine the translation between the two images.
local_max_sub_pix (PhaseCorrelation, 'facet', 1, 0.02, Row, Column)
```

Result

phase_correlation_fft returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Predecessors

fft_generic, fft_image, rft_generic

Possible Successors

fft_generic, fft_image_inv, rft_generic

Alternatives

correlation_fft

References

C. D. Kuglin, D. C. Hines: "The Phase Correlation Image Alignment Method"; IEEE International Conference on Cybernetics and Society; pp. 163-165; 1975.

Module

Foundation

phase_deg (ImageComplex : ImagePhase : :)

Return the phase of a complex image in degrees.

phase_deg computes the phase of a complex image in degrees. The following formula is used:

$$phase = \frac{90}{\pi}\text{atan2}(imaginary\ part, real\ part) \quad .$$

Hence, ImagePhase contains half the phase angle. For negative phase angles, 180 is added.

Parameters

▷ **ImageComplex** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
Input image in frequency domain.

▷ **ImagePhase** (output_object) ..... image(-array) ⇝ *object :* direction
Phase of the image in degrees.

Example

```
read_image(&Image,"monkey");
disp_image(Image,WindowHandle);
fft_image(Image,&FFT);
phase_deg(FFT,&Phase);
disp_image(Phase,WindowHandle);
```

Result

phase_deg returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

fft_image, fft_generic, rft_generic

Possible Successors

disp_image

Alternatives

phase_rad

See also

fft_image_inv

Module

Foundation

phase_rad (ImageComplex : ImagePhase : :)

Return the phase of a complex image in radians.

phase_rad computes the phase of a complex image in radians. The following formula is used:

$$phase = \text{atan2}(imaginary\ part, real\ part) \quad .$$

Parameters

- ▷ **ImageComplex** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
  Input image in frequency domain.
- ▷ **ImagePhase** (output_object) ..... image(-array) ⇝ *object :* real
  Phase of the image in radians.

Example

```
read_image(&Image,"monkey");
disp_image(Image,WindowHandle);
fft_image(Image,&FFT);
phase_rad(FFT,&Phase);
disp_image(Phase,WindowHandle);
```

Result

phase_rad returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

fft_image, fft_generic, rft_generic

Possible Successors

disp_image

Alternatives

phase_deg

See also

fft_image_inv, fft_generic, rft_generic

Module

Foundation

power_byte (Image : PowerByte : :)

Return the power spectrum of a complex image.

power_byte computes the power spectrum from the real and imaginary parts of a Fourier-transformed image (see fft_image), i.e., the modulus of the frequencies. The result image is of type *'byte'*. The following formula is used:

$$\sqrt{realpart^2 + imaginarypart^2} \ .$$

Please note, that resulting gray values that exceed the value of 255 are clipped at 255 because of the resulting image type *'byte'*.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
Input image in frequency domain.

▷ **PowerByte** (output_object) ..... image(-array) ⇝ *object :* byte
Power spectrum of the input image.

Example

```
read_image(&Image,"monkey");
disp_image(Image,WindowHandle);
fft_image(Image,&FFT);
power_byte(FFT,&Power);
disp_image(Power,WindowHandle);
```

Result

power_byte returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

fft_image, fft_generic, rft_generic, convol_fft, convol_gabor

Possible Successors

disp_image

Alternatives

abs_image, convert_image_type, power_real, power_ln

See also

fft_image, fft_generic, rft_generic

Module

Foundation

power_ln (Image : ImageResult : :)

Return the power spectrum of a complex image.

power_ln computes the power spectrum from the real and imaginary parts of a Fourier-transformed image (see fft_image), i.e., the modulus of the frequencies. Additionally, the natural logarithm is applied to the result. The result image is of type real. The following formula is used:

$$\ln\left(\sqrt{realpart^2 + imaginarypart^2}\right) \quad .$$

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
Input image in frequency domain.

▷ **ImageResult** (output_object) ..... image(-array) ⇝ *object :* real
Power spectrum of the input image.

Example

```
read_image(&Image,"monkey");
disp_image(Image,WindowHandle);
fft_image(Image,&FFT);
power_ln(FFT,&Power);
disp_image(Power,WindowHandle);
```

Result

power_ln returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

fft_image, fft_generic, rft_generic, convol_fft, convol_gabor

Possible Successors

disp_image, convert_image_type, scale_image

Alternatives

abs_image, convert_image_type, power_real, power_byte

See also

fft_image, fft_generic, rft_generic

Module

Foundation

power_real (Image : ImageResult : :)

Return the power spectrum of a complex image.

power_real computes the power spectrum from the real and imaginary parts of a Fourier-transformed image (see fft_image), i.e., the modulus of the frequencies. The result image is of type real. The following formula is used:

$$\sqrt{realpart^2 + imaginarypart^2} \quad .$$

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* complex
Input image in frequency domain.

▷ **ImageResult** (output_object) ..... image(-array) ⇝ *object :* real
Power spectrum of the input image.

Example

```
read_image(&Image,"monkey");
disp_image(Image,WindowHandle);
fft_image(Image,&FFT);
power_real(FFT,&Power);
disp_image(Power,WindowHandle);
```

Result

power_real returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.
- Automatically parallelized on domain level.

Possible Predecessors

fft_image, fft_generic, rft_generic, convol_fft, convol_gabor

Possible Successors

disp_image, convert_image_type, scale_image

Alternatives

abs_image, convert_image_type, power_byte, power_ln

See also

fft_image, fft_generic, rft_generic

Module

Foundation

read_fft_optimization_data (: : FileName :)

Load FFT speed optimization data from a file.

read_fft_optimization_data loads data for optimizing the runtime of the FFT from the file given by FileName. The optimization data must have been determined previously with optimize_fft_speed and must have been stored with write_fft_optimization_data. If the stored data have been determined for the image sizes to be used in the application, a call to optimize_fft_speed is unnecessary. It should be noted that the data should only be used on the same machine on which they were determined with optimize_fft_speed. If this is not observed the runtimes will not be optimal. Furthermore, it should be noted that optimization data that were created with Standard HALCON cannot be used with Parallel HALCON and vice versa.

read_fft_optimization_data influences the runtime of the following operators, which use the FFT: fft_generic, fft_image, fft_image_inv, sfs_pentland, sfs_mod_lr, sfs_orig_lrwiener_filter.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name of the optimization data.
Default: 'fft_opt.dat'

Result

`read_fft_optimization_data` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`fft_generic`, `fft_image`, `fft_image_inv`, `rft_generic`, `wiener_filter`, `wiener_filter_ni`, `photometric_stereo`, `sfs_pentland`, `sfs_mod_lr`, `sfs_orig_lr`

Alternatives

`optimize_fft_speed`, `optimize_rft_speed`

See also

`write_fft_optimization_data`

Module

Foundation

rft_generic (Image : ImageFFT : Direction, Norm, ResultType, Width :)

Compute the real-valued fast Fourier transform of an image.

`rft_generic` computes the fast Fourier transform of the input image `Image`. In contrast to `fft_generic`, `fft_image`, and `fft_image_inv`, the fact that the input image in the forward transform is a real-valued image (i.e., not a complex image) is used. In this case, the complex output image has a redundancy. The values in the right half of the image are the complex conjugates of the corresponding values in the left half of the image. Consequently, runtime and memory can be saved by only computing and storing the left half of the complex image.

The parameter `ResultType` can be used to specify the result image type of the reverse transform (`Direction` = *'from_freq'*). In the forward transform (`Direction` = *'to_freq'*), `ResultType` must be set to *'complex'*.

The parameter direction determines whether the transform should be performed to the frequency domain or back into the spatial domain. For `Direction` = *'to_freq'* the input image must have a real-valued type, i.e., a complex image may not be used as input. All image types that can be converted into an image of type real are supported. In this case, the output is a complex image of dimension $(w/2 + 1) \times h$, where w and h are the width and height of the input image. In this mode, the exponent -1 is used in the transform (see `fft_generic`). For `Direction` = *'from_freq'*, the input image must be complex. In this case, the size of the input image is insufficient to determine the size of the output image. This must be done by setting `Width` to a valid value, i.e., to $2w - 2$ or $2w - 1$, where w is the width of the complex image. In this mode, the exponent 1 is used in the transform.

The normalizing factor can be set with `Norm`, and can take on the values *'none'*, *'sqrt'* and *'n'*. The user must ensure the consistent use of the parameters. This means that the normalizing factors used for the forward and backward transform must yield wh when multiplied.

Attention

The transformation is always performed for the entire image, i.e., the domain of the image is ignored.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex

Input image.

▷ **ImageFFT** (output_object) ..... image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex
Fourier-transformed image.

▷ **Direction** (input_control) ..... string ⇝ *string*
Calculate forward or reverse transform.
Default: 'to_freq'
List of values: Direction ∈ {'to_freq', 'from_freq'}

▷ **Norm** (input_control) ..... string ⇝ *string*
Normalizing factor of the transform.
Default: 'sqrt'
List of values: Norm ∈ {'none', 'sqrt', 'n'}

▷ **ResultType** (input_control) ..... string ⇝ *string*
Image type of the output image.
Default: 'complex'
List of values: ResultType ∈ {'complex', 'byte', 'int1', 'int2', 'uint2', 'int4', 'real', 'direction', 'cyclic'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the image for which the runtime should be optimized.
Default: 512
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768, 1024, 2048}

Result

rft_generic returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

optimize_rft_speed, read_fft_optimization_data

Possible Successors

convol_fft, correlation_fft, phase_correlation_fft, convert_image_type, power_byte, power_real, power_ln, phase_deg, phase_rad

Alternatives

fft_generic, fft_image, fft_image_inv

Module

Foundation

serialize_fft_optimization_data (: : : SerializedItemHandle)

Serialize FFT speed optimization data.

serialize_fft_optimization_data serializes the data for the optimization of the runtime of the FFT that were determined with optimize_fft_speed (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_fft_optimization_data is converted to a serialized item. The serialized data is returned by the handle SerializedItemHandle and can be deserialized by deserialize_fft_optimization_data.

Parameters

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

serialize_fft_optimization_data returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

optimize_fft_speed, optimize_rft_speed

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_fft_optimization_data

See also

fft_generic, fft_image, fft_image_inv, wiener_filter, wiener_filter_ni, photometric_stereo, sfs_pentland, sfs_mod_lr, sfs_orig_lr, deserialize_fft_optimization_data

Module

Foundation

write_fft_optimization_data (: : FileName :)

Store FFT speed optimization data in a file.

write_fft_optimization_data stores the data for the optimization of the runtime of the FFT that were determined with optimize_fft_speed in the file given by FileName. The data can be loaded with read_fft_optimization_data.

Parameters

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name of the optimization data.
Default: 'fft_opt.dat'

Result

write_fft_optimization_data returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

optimize_fft_speed, optimize_rft_speed

See also

fft_generic, fft_image, fft_image_inv, wiener_filter, wiener_filter_ni, photometric_stereo, sfs_pentland, sfs_mod_lr, sfs_orig_lr, read_fft_optimization_data

Module

Foundation

12.7 Geometric Transformations

affine_trans_image (Image : ImageAffineTrans : HomMat2D, Interpolation, AdaptImageSize :)

Apply an arbitrary affine 2D transformation to images.

affine_trans_image applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the images given in Image and returns the transformed images in ImageAffineTrans. The affine transformation is described by the homogeneous transformation matrix given in HomMat2D, which can be created using the operators hom_mat2d_identity, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_translate, etc., or be the result of operators like vector_angle_to_rigid.

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to x and the *column* coordinate corresponds to y of the coordinate system in which the transformation matrix was defined. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

The domain of the input image is ignored, i.e., assumed to be the full rectangle of the image. The domain of the output image is the intersection of the transformed rectangle and the rectangle of the output image.

Generally, transformed points will lie between pixel coordinates. Therefore, an appropriate interpolation scheme must be used. The interpolation can also be used to avoid aliasing effects for scaled images. The quality and speed of the interpolation can be set by the parameter Interpolation:

nearest_neighbor Nearest-neighbor interpolation: The gray value is determined from the nearest pixel's gray value (possibly low quality, very fast).

bilinear Bilinear interpolation. The gray value is determined from the four nearest pixels through bilinear interpolation. If the affine transformation contains a scaling with a scale factor < 1, no smoothing is performed, which may cause severe aliasing effects (medium quality and run time).

bicubic Bicubic interpolation. The gray value is determined from the 4×4 nearest pixels through bicubic interpolation. If the affine transformation contains a scaling with a scale factor < 1, no smoothing is performed, which may cause severe aliasing effects (high quality for enlargements, slow).

constant Bilinear interpolation. The gray value is determined from the four nearest pixels through bilinear interpolation. If the affine transformation contains a scaling with a scale factor < 1, a kind of mean filter is used to prevent aliasing effects (medium quality and run time).

weighted Bilinear interpolation. The gray value is determined from the four nearest pixels through bilinear interpolation. If the affine transformation contains a scaling with a scale factor < 1, a kind of Gaussian filter is used to prevent aliasing effects (high quality, slow).

In addition, the system parameter *'int_zooming'* (see set_system) affects the accuracy of the transformation. If *'int_zooming'* is set to *'true'*, the transformation for byte, int2 and uint2 images is carried out internally using fixed point arithmetic, leading to much shorter execution times. However, the accuracy of the transformed gray values is smaller in this case. For byte images, the differences to the more accurate calculation (using *'int_zooming'* = *'false'*) is typically less than two gray levels. Correspondingly, for int2 and uint2 images, the gray value differences are less than 1/128 times the dynamic gray value range of the image, i.e., they can be as large as 512 gray levels if the entire dynamic range of 16 bit is used. When using fixed point arithmetic, the domain of resulting images can differ as well. Additionally, if a large scale factor is applied and a large output image is obtained, then undefined gray values at the lower and at the right image border may result. The maximum width B_{max} of this border of undefined gray values can be estimated as $B_{max} = 0.5 \cdot S \cdot I/2^{15}$, where S is the scale factor in one dimension and I is the size of the output image in the corresponding dimension. For real images, the parameter *'int_zooming'* does not affect the accuracy, since the internal calculations are always done using floating point arithmetic.

The size of the target image can be controlled by the parameter AdaptImageSize: If set to *'true'*, the size will be adapted so that no clipping occurs at the right or lower edge. If set to *'false'*, the target image has the same size as the input image. Note that, independent of AdaptImageSize, the image is always clipped at the left and upper edge, i.e., all image parts that have negative coordinates after the transformation are clipped.

Attention

The region of the input image is ignored.

affine_trans_image does not use the HALCON standard coordinate system (with the origin in the center of the first pixel), but instead uses the same coordinate system as in affine_trans_pixel, i.e., the origin lies in the upper left corner of the first pixel. Therefore, applying affine_trans_image corresponds to a chain of transformations (see affine_trans_pixel), which is applied to each point of the image (input and output pixels as homogeneous vectors). As an effect, you might get unexpected results when creating affine transformations based on coordinates that are derived from the image, e.g., by operators like area_center_gray. For example, if you use this operator to calculate the center of gravity of a rotationally symmetric image and then rotate the image around this point using hom_mat2d_rotate, the resulting image will not lie on the original one. In such a case, you can compensate this effect by applying the following translations to HomMat2D before using it in affine_trans_image:

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
affine_trans_image(Image, ImageAffineTrans, HomMat2DAdapted,
'constant', 'false')
```

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
  Input image.
- ▷ **ImageAffineTrans** (output_object) .... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
  Transformed image.
- ▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
  Input transformation matrix.
- ▷ **Interpolation** (input_control) ..... string ⇝ *string*
  Type of interpolation.
  Default: 'constant'
  List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'constant', 'weighted'}
- ▷ **AdaptImageSize** (input_control) ..... string ⇝ *string*
  Adaption of size of result image.
  Default: 'false'
  List of values: AdaptImageSize ∈ {'true', 'false'}

Example

```
* Reduction of an image (512 x 512 Pixels) by 50%, rotation
* by 180 degrees and translation to the upper-left corner:

read_image (Image, 'ic0')
hom_mat2d_identity(Matrix1)
hom_mat2d_scale(Matrix1,0.5,0.5,256.0,256.0,Matrix2)
hom_mat2d_rotate(Matrix2,3.14,256.0,256.0,Matrix3)
hom_mat2d_translate(Matrix3,-128.0,-128.0,Matrix4)
affine_trans_image(Image,TransImage,Matrix4,'constant','true')

* Enlarging the part of an image in the interactively
* chosen rectangular window sector:

dev_get_window (WindowHandle)
draw_rectangle2(WindowHandle,L,C,Phi,L1,L2)
hom_mat2d_identity(Matrix1)
get_system('width',Width)
get_system('height',Height)
hom_mat2d_translate(Matrix1,Height/2.0-L,Width/2.0-C,Matrix2)
hom_mat2d_rotate(Matrix2,3.14-Phi,Height/2.0,Width/2.0,Matrix3)
```

```
hom_mat2d_scale(Matrix3,Height/(2.0*L2),Width/(2.0*L1), \
                Height/2.0,Width/2.0,Matrix4)
affine_trans_image(Image,TransImage,Matrix4,'constant','true')
```

Result

If the matrix HomMat2D represents an affine transformation (i.e., not a projective transformation), affine_trans_image returns 2 (H_MSG_TRUE). If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate, hom_mat2d_rotate, hom_mat2d_scale, hom_mat2d_reflect

Alternatives

affine_trans_image_size, zoom_image_size, zoom_image_factor, mirror_image, rotate_image, affine_trans_region

See also

set_part_style

Module

Foundation

affine_trans_image_size (Image : ImageAffineTrans : HomMat2D, Interpolation, Width, Height :)

Apply an arbitrary affine 2D transformation to an image and specify the output image size.

affine_trans_image_size applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the images given in Image and returns the transformed images in ImageAffineTrans. The affine transformation is described by the homogeneous transformation matrix given in HomMat2D, which can be created using the operators hom_mat2d_identity, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_translate, etc., or be the result of operators like vector_angle_to_rigid.

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to x and the *column* coordinate corresponds to y of the coordinate system in which the transformation matrix was defined. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

The region of the input image is ignored, i.e., assumed to be the full rectangle of the image. The region of the resulting image is set to the transformed rectangle of the input image. If necessary, the resulting image is filled with zero (black) outside of the region of the original image.

Generally, transformed points will lie between pixel coordinates. Therefore, an appropriate interpolation scheme has to be used. The interpolation can also be used to avoid aliasing effects for scaled images. The quality and speed of the interpolation can be set by the parameter Interpolation:

nearest_neighbor Nearest-neighbor interpolation: The gray value is determined from the nearest pixel's gray value (possibly low quality, very fast).

bilinear Bilinear interpolation. The gray value is determined from the four nearest pixels through bilinear interpolation. If the affine transformation contains a scaling with a scale factor < 1, no smoothing is performed, which may cause severe aliasing effects (medium quality and run time).

bicubic Bicubic interpolation. The gray value is determined from the 4×4 nearest pixels through bicubic interpolation. If the affine transformation contains a scaling with a scale factor < 1, no smoothing is performed, which may cause severe aliasing effects (high quality for enlargements, slow).

constant Bilinear interpolation. The gray value is determined from the four nearest pixels through bilinear interpolation. If the affine transformation contains a scaling with a scale factor < 1, a kind of mean filter is used to prevent aliasing effects (medium quality and run time).

weighted Bilinear interpolation. The gray value is determined from the four nearest pixels through bilinear interpolation. If the affine transformation contains a scaling with a scale factor < 1, a kind of Gaussian filter is used to prevent aliasing effects (high quality, slow).

In addition, the system parameter *'int_zooming'* (see `set_system`) affects the accuracy of the transformation. If *'int_zooming'* is set to *'true'*, the transformation for byte, int2 and uint2 images is carried out internally using fixed point arithmetic, leading to much shorter execution times. However, the accuracy of the transformed gray values is smaller in this case. For byte images, the differences to the more accurate calculation (using *'int_zooming'* = *'false'*) is typically less than two gray levels. Correspondingly, for int2 and uint2 images, the gray value differences are less than 1/128 times the dynamic gray value range of the image, i.e., they can be as large as 512 gray levels if the entire dynamic range of 16 bit is used. When using fixed point arithmetic, the domain of resulting images can differ as well. Additionally, if a large scale factor is applied and a large output image is obtained, then undefined gray values at the lower and at the right image border may result. The maximum width B_{max} of this border of undefined gray values can be estimated as $B_{max} = 0.5 \cdot S \cdot I/2^{15}$, where S is the scale factor in one dimension and I is the size of the output image in the corresponding dimension. For real images, the parameter *'int_zooming'* does not affect the accuracy, since the internal calculations are always done using floating point arithmetic.

The size of the target image is specified by the parameters `Width` and `Height`. Note that the image is always clipped at the left and upper edge, i.e., all image parts that have negative coordinates after the transformation are clipped. If the affine transformation (in particular, the translation) is chosen appropriately, a part of the image can be transformed as well as cropped in one call. This is useful, for example, when using the variation model (see `compare_variation_model`), because with this mechanism only the parts of the image that should be examined, are transformed.

Attention

The region of the input image is ignored.

`affine_trans_image_size` does not use the HALCON standard coordinate system (with the origin in the center of the upper left pixel), but instead uses the same coordinate system as in `affine_trans_pixel`, i.e., the origin lies in the upper left corner of the upper left pixel. Therefore, applying `affine_trans_image_size` corresponds to a chain of transformations (see `affine_trans_pixel`), which is applied to each point of the image (input and output pixels as homogeneous vectors). As an effect, you might get unexpected results when creating affine transformations based on coordinates that are derived from the image, e.g., by operators like `area_center_gray`. For example, if you use this operator to calculate the center of gravity of a rotationally symmetric image and then rotate the image around this point using `hom_mat2d_rotate`, the resulting image will not lie on the original one. In such a case, you can compensate this effect by applying the following translations to `HomMat2D` before using it in `affine_trans_image_size`:

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
affine_trans_image_size(Image, ImageAffineTrans, HomMat2DAdapted,
'constant', Width, Height)
```

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Input image.

▷ **ImageAffineTrans** (output_object) .... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Transformed image.

▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Interpolation** (input_control) ..... string ⇝ *string*
Type of interpolation.
Default: 'constant'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'constant', 'weighted'}

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the output image.
Default: 640
Suggested values: Width ∈ {128, 160, 192, 256, 320, 384, 512, 640, 768}

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the output image.
Default: 480
Suggested values: Height ∈ {120, 128, 144, 240, 256, 288, 480, 512, 576}

Result

If the matrix HomMat2D represents an affine transformation (i.e., not a projective transformation), affine_trans_image_size returns 2 (H_MSG_TRUE). If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate, hom_mat2d_rotate, hom_mat2d_scale, hom_mat2d_reflect

Alternatives

affine_trans_image, zoom_image_size, zoom_image_factor, mirror_image, rotate_image, affine_trans_region

See also

set_part_style

Module

Foundation

convert_map_type (Map : MapConverted : NewType, ImageWidth :)

Convert image maps into other map types.

convert_map_type converts the input Map into a different map type. The main usage of convert_map_type is to convert a nearest neighbor or bilinear map (e.g., obtained by gen_image_to_world_plane_map) to a coordinate map, as this is the only type supported on compute devices.

NewType specifies the type of conversion. The following values are supported:

'coord_map_sub_pix' creates a map that consists of a vector_field image which contains the absolute subpixel precise row and column coordinates of each pixel's mapping. (input map must be *'bilinear'* or *'nearest_neighbor'*)

'bilinear' creates a map with linear coordinates and bilinear interpolation coefficients (see map_image). (input map must be *'coord_map_sub_pix'*)

'nearest_neighbor' creates a nearest neighbor map with linear coordinates (see map_image). (input map must be *'coord_map_sub_pix'*)

In ImageWidth the width of the images to be mapped by MapConverted must be given. If this width is identical to the width of Map, you can pass the string *'map_width'*.

Parameters

- ▷ **Map** (input_object) ..... (multichannel-)image ⇝ *object :* vector_field / int4 / int8 / uint2
  Input map.
- ▷ **MapConverted** (output_object) ..... (multichannel-)image ⇝ *object :* vector_field / int4 / int8 / uint2
  Converted map.
- ▷ **NewType** (input_control) ..... string ⇝ *string*
  Type of MapConverted.
  Default: 'coord_map_sub_pix'
  List of values: NewType ∈ {'coord_map_sub_pix', 'bilinear', 'nearest_neighbor'}
- ▷ **ImageWidth** (input_control) ..... integer ⇝ *integer* / string
  Width of images to be mapped.
  Default: 'map_width'

Example

```
gen_radial_distortion_map (MapFixed, CamParOriginal, CamParVirtualFixed, \
                           'bilinear')
convert_map_type(MapFixed, MapConverted, 'coord_map_sub_pix', 'map_width')
query_available_compute_devices (DeviceIdentifier)
open_compute_device (DeviceIdentifier[0], DeviceHandle)
activate_compute_device (DeviceHandle)
map_image (GrayImage, MapConverted, ImageRectifiedFixed)
```

Result

convert_map_type returns 2 (H_MSG_TRUE) if all parameter values are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_image_to_world_plane_map, gen_radial_distortion_map, gen_arbitrary_distortion_map, gen_grid_rectification_map, find_local_deformable_model

Possible Successors

map_image

Module

Foundation

map_image (Image, Map : ImageMapped : :)

Apply a general transformation to an image.

map_image transforms an image Image using an arbitrary transformation Map which, for example, was previously generated using gen_image_to_world_plane_map or gen_radial_distortion_map. The multi-channel image Map must be organized as follows:

The height and the width of Map define the size of the output image ImageMapped. The number of channels in Map defines whether no interpolation or bilinear interpolation should be used. If Map only consists of one channel, no interpolation is applied during the transformation. This channel contains 'int4' (resp. 'int8' in HALCON XL if the value range of 'int4' is not sufficient) values that describe the geometric transformation: For each pixel in the output image ImageMapped the linearized coordinate of the pixel in the input image Image from which the gray value should be taken is stored.

If bilinear interpolation between the pixels in the input image should be applied, Map must consist of 5 channels. The first channel again consists of an 'int4' resp. 'int8' image and describes the geometric transformation. The channels 2-5 consist of an 'uint2' image each and contain the weights [0...1] of the four neighboring pixels that are used during bilinear interpolation. If the overall brightness of the output image ImageMapped should not differ from the overall brightness of the input image Image, the sum of the four unscaled weights must be 1 for each pixel. The weights [0...1] are scaled to the range of values of the 'uint2' image and therefore hold integer values from 0 to 65535.

Furthermore, the weights must be chosen in a way that the range of values of the output image ImageMapped is not exceeded. The geometric relation between the four channels 2-5 is illustrated in the following sketch:

2	3
4	5

The reference point of the four pixels is the upper left pixel. The linearized coordinate of the reference point is stored in the first channel.

It is also possible to use a Map that consists of a vector field containing absolute subpixel precise row and column coordinates (i.e., the field must be of the semantic type 'vector_field_absolute'). The two Map types described above can be converted into this type using convert_map_type. This type is the only type supported on compute devices!

Attention

The weights must be chosen in a way that the range of values of the output image ImageMapped is not exceeded.

For runtime reasons during the mapping process, it is not checked whether the linearized coordinates lie inside the input image. Thus, it must be ensured by the user that this constraint is fulfilled. In case interpolation is used, this also applies to the pixels to the right, below, and below to the right of this coordinate. Otherwise, the program may crash!

map_image is parallelized automatically if and only if the specified type for Map is either *'bilinear'* or *'coord_map_sub_pix'*.

map_image is executed on an OpenCL compute device only if the input map is of type *'coord_map_sub_pix'* and if the input image does not exceed the maximum size of image objects of the selected device.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Image to be mapped.
- ▷ **Map** (input_object) ..... (multichannel-)image ⇝ *object :* int4 / int8 / uint2 / vector_field
  Image containing the mapping data.
- ▷ **ImageMapped** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Mapped image.

Result

map_image returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

gen_image_to_world_plane_map, gen_radial_distortion_map, convert_map_type

See also

affine_trans_image, rotate_image

Module

Foundation

mirror_image (Image : ImageMirror : Mode :)

Mirror an image.

mirror_image reflects an image Image about one of three possible axes. If Mode is set to *'row'*, it is reflected about the horizontal axis, if Mode is set to *'column'*, about the vertical axis, and if Mode is set to *'diagonal'*, about the main diagonal $x = y$.

Attention

mirror_image can be executed on OpenCL devices if the input image does not exceed the maximum size of image objects of the selected device. However, execution might be faster on the CPU, especially for the mode *'row'*.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
  Input image.
- ▷ **ImageMirror** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
  Reflected image.
- ▷ **Mode** (input_control) . . . . . string ⇝ *string*
  Axis of reflection.
  Default: 'row'
  List of values: Mode ∈ {'row', 'column', 'diagonal'}

Example

```
read_image(Image,'monkey')
dev_display (Image)
mirror_image(Image,MirrorImage,'row')
dev_display (MirrorImage)
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Alternatives

hom_mat2d_rotate, hom_mat2d_reflect, affine_trans_image, rotate_image

See also

rotate_image, hom_mat2d_rotate

Module

Foundation

polar_trans_image_ext (Image : PolarTransImage : Row, Column, AngleStart, AngleEnd, RadiusStart, RadiusEnd, Width, Height, Interpolation :)

Transform an annular arc in an image to polar coordinates.

polar_trans_image_ext transforms the annular arc specified by the center point (Row, Column), the radii RadiusStart and RadiusEnd and the angles AngleStart and AngleEnd in the image Image to its polar coordinate version in the image PolarTransImage of the dimensions Width × Height.

The upper left pixel in the output image always corresponds to the point in the input image that is specified by RadiusStart and AngleStart. Analogously, the lower right pixel in the output image always corresponds to the point in the input image that is specified by RadiusEnd and AngleEnd. In the usual mode (AngleStart < AngleEnd and RadiusStart < RadiusEnd), the polar transformation is performed in the mathematically positive orientation (counterclockwise). Furthermore, points with smaller radii lie in the upper part of the output image. By suitably exchanging the values of these parameters (e.g., AngleStart > AngleEnd or RadiusStart > RadiusEnd), any desired orientation of the output image can be achieved.

The parameter Interpolation is used to select the interpolation method *'bilinear'* or *'nearest_neighbor'*. With *'nearest_neighbor'*, the gray value of a pixel in the output image is determined by the gray value of the closest pixel in the input image. With *'bilinear'*, the gray value of a pixel in the output image is determined by bilinear interpolation of the gray values of the four closest pixels in the input image. The mode *'bilinear'* results in images of better quality, but is slower than the mode *'nearest_neighbor'*.

The angles can be chosen from all real numbers. Center point and radii can be real as well. However, if they are both integers and the difference of RadiusEnd and RadiusStart equals the height Height of the destination image, calculation will be sped up through an optimized routine.

The radii and angles are inclusive, which means that the first row of the target image contains the circle with radius RadiusStart and the last row contains the circle with radius RadiusEnd. For complete circles, where the difference between AngleStart and AngleEnd equals 2π (360 degrees), this also means that the first column of the target image will be the same as the last.

To avoid this, do not make this difference 2π, but $2\pi(1 - \frac{1}{\texttt{Width}})$ degrees instead.

The call `polar_trans_image(Image, PolarTransImage, Row, Column, Width, Height)` produces the same result as the call `polar_trans_image_ext(Image, PolarTransImage, Row-0.5, Column-0.5, 6.2831853, 6.2831853/Width, 0, Height-1, Width, Height, 'nearest_neighbor')`.

The offset of 0.5 is necessary since polar_trans_image does not do exact nearest neighbor interpolation and the radii and angles can be calculated using the information in the above paragraph and knowing that polar_trans_image does not handle its arguments inclusively. The start angle is bigger than the end angle to make polar_trans_image_ext go clockwise, just like polar_trans_image does.

polar_trans_image_ext can be executed on an OpenCL device if the input image does not exceed the maximum size of image objects of the selected device. Due to numerical reasons, there can be slight differences in the output compared to the execution on the CPU.

Additionally, for images of type byte, int2 or uint2 the system parameter *'int_zooming'* selects between fast calculation in fixed point arithmetics (*'int_zooming'* = *'true'*) and highly accurate calculation in floating point arithmetics (*'int_zooming'* = *'false'*). Fixed point calculation can lead to minor gray value deviations and pixels with undefined gray values.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Attention

For speed reasons, the domain of the input image is ignored. The output image always has a complete rectangle as its domain.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
  Input image.
- ▷ **PolarTransImage** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
  Output image.
- ▷ **Row** (input_control) ..... number ⇝ *real* / integer
  Row coordinate of the center of the arc.
  Default: 256
  Suggested values: Row ∈ {0, 16, 32, 64, 128, 240, 256, 480, 512}
- ▷ **Column** (input_control) ..... number ⇝ *real* / integer
  Column coordinate of the center of the arc.
  Default: 256
  Suggested values: Column ∈ {0, 16, 32, 64, 128, 256, 320, 512, 640}

▷ **AngleStart** (input_control) ..... angle.rad ⇝ *real*
Angle of the ray to be mapped to the first column of the output image.
Default: 0.0
Suggested values: AngleStart ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853, 12.566370616}

▷ **AngleEnd** (input_control) ..... angle.rad ⇝ *real*
Angle of the ray to be mapped to the last column of the output image.
Default: 6.2831853
Suggested values: AngleEnd ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853, 12.566370616}

▷ **RadiusStart** (input_control) ..... number ⇝ *real* / integer
Radius of the circle to be mapped to the first row of the output image.
Default: 0
Suggested values: RadiusStart ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ RadiusStart

▷ **RadiusEnd** (input_control) ..... number ⇝ *real* / integer
Radius of the circle to be mapped to the last row of the output image.
Default: 100
Suggested values: RadiusEnd ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ RadiusEnd

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the output image.
Default: 512
Suggested values: Width ∈ {256, 320, 512, 640, 800, 1024}
Value range: 0 ≤ Width ≤ 32767

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the output image.
Default: 512
Suggested values: Height ∈ {240, 256, 480, 512, 600, 1024}
Value range: 0 ≤ Height ≤ 32767

▷ **Interpolation** (input_control) ..... string ⇝ *string*
Interpolation method for the transformation.
Default: 'nearest_neighbor'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear'}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

See also

polar_trans_image_inv, polar_trans_region, polar_trans_region_inv, polar_trans_contour_xld, polar_trans_contour_xld_inv

Module

Foundation

polar_trans_image_inv (PolarImage : XYTransImage : Row, Column, AngleStart, AngleEnd, RadiusStart, RadiusEnd, Width, Height, Interpolation :)

Transform an image in polar coordinates back to Cartesian coordinates

polar_trans_image_inv transforms the polar coordinate representation of an image, stored in PolarImage, back onto an annular arc in Cartesian coordinates, described by the radii RadiusStart and

RadiusEnd and the angles AngleStart and AngleEnd with the center point located at (Row, Column). All of these values can be chosen as real numbers. The overall size of the target image will be Width × Height pixels.

The parameter Interpolation is used to select the interpolation method *'bilinear'* or *'nearest_neighbor'*. With *'nearest_neighbor'*, the gray value of a pixel in the output image is determined by the gray value of the closest pixel in the input image. With *'bilinear'*, the gray value of a pixel in the output image is determined by bilinear interpolation of the gray values of the four closest pixels in the input image. The mode *'bilinear'* results in images of better quality, but is slower than the mode *'nearest_neighbor'*.

The angles and radii are inclusive, which means that the first row of the input image will be mapped onto a circle with a distance of RadiusStart pixels from the specified center and the last row will be mapped onto a circle of radius RadiusEnd.

polar_trans_image_inv is the inverse function of polar_trans_image_ext.

The call sequence:

```
polar_trans_image_ext(Image, PolarImage, Row, Column, rad(360)
polar_trans_image_inv(PolarImage, XYTransImage, Row, Column,
rad(360), 0, 0, Radius, Width, Height, Interpolation)
```

returns the image Image, restricted to the circle around (Row, Column) with radius Radius, as its output image XYTransImage.

polar_trans_image_inv can be executed on an OpenCL device. There can be slight differences in the output compared to the execution on the CPU.

Additionally, for images of type byte, int2 or uint2 the system parameter *'int_zooming'* selects between fast calculation in fixed point arithmetics (*'int_zooming'* = *'true'*) and highly accurate calculation in floating point arithmetics (*'int_zooming'* = *'false'*). Fixed point calculation can lead to minor gray value deviations. In this case, the domain of resulting images can differ as well.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **PolarImage** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Input image.

▷ **XYTransImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Output image.

▷ **Row** (input_control) . . . . . number ⇝ *real* / integer
Row coordinate of the center of the arc.
Default: 256
Suggested values: Row ∈ {0, 16, 32, 64, 128, 240, 256, 480, 512}
Value range: 0 ≤ Row ≤ 32767

▷ **Column** (input_control) . . . . . number ⇝ *real* / integer
Column coordinate of the center of the arc.
Default: 256
Suggested values: Column ∈ {0, 16, 32, 64, 128, 256, 320, 512, 640}
Value range: 0 ≤ Column ≤ 32767

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Angle of the ray to map the first column of the input image to.
Default: 0.0
Suggested values: AngleStart ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853}

▷ **AngleEnd** (input_control) . . . . . angle.rad ⇝ *real*
Angle of the ray to map the last column of the input image to.
Default: 6.2831853
Suggested values: AngleEnd ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853}

▷ **RadiusStart** (input_control) ..... number ⇝ *real* / integer
Radius of the circle to map the first row of the input image to.
Default: 0
Suggested values: RadiusStart ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ RadiusStart

▷ **RadiusEnd** (input_control) ..... number ⇝ *real* / integer
Radius of the circle to map the last row of the input image to.
Default: 100
Suggested values: RadiusEnd ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ RadiusEnd

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the output image.
Default: 512
Suggested values: Width ∈ {256, 320, 512, 640, 800, 1024}
Value range: 0 ≤ Width ≤ 32767

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the output image.
Default: 512
Suggested values: Height ∈ {240, 256, 480, 512, 600, 1024}
Value range: 0 ≤ Height ≤ 32767

▷ **Interpolation** (input_control) ..... string ⇝ *string*
Interpolation method for the transformation.
Default: 'nearest_neighbor'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear'}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

See also

polar_trans_image_ext, polar_trans_region, polar_trans_region_inv, polar_trans_contour_xld, polar_trans_contour_xld_inv

Module

Foundation

projective_trans_image (Image : TransImage : HomMat2D, Interpolation, AdaptImageSize, TransformDomain :)

Apply a projective transformation to an image.

projective_trans_image applies the projective transformation (homography) determined by the homogeneous transformation matrix HomMat2D on the input image Image and stores the result into the output image TransImage.

If the parameter AdaptImageSize is set to *'false'*, TransImage will have the same size as Image; if AdaptImageSize is *'true'*, the output image size will be automatically adapted so that all non-negative points of the transformed image are visible.

The parameter Interpolation determines, which interpolation method is used to determine the gray values of the output image. For Interpolation = *'nearest_neighbor'*, the gray value is determined from the nearest pixel in the input image. This mode is very fast, but also leads to the typical "jagged" appearance for large enlargements of the image. For Interpolation = *'bilinear'*, the gray values are interpolated bilinearly, leading to longer runtimes, but also to significantly improved results.

The parameter TransformDomain can be used to determine whether the domain of Image is also transformed. Since the transformation of the domain costs runtime, this parameter should be used to specify whether this is desired or not. If TransformDomain is set to *'false'* the domain of the input image is ignored and the complete image is transformed.

The projective transformation matrix could for example be created using the operator vector_to_proj_hom_mat2d.

In a homography the points to be projected are represented by homogeneous vectors of the form (x, y, w). A Euclidean point can be derived as (x',y') = $(\frac{x}{w}, \frac{y}{w})$.

Just like in affine_trans_image, x represents the row coordinate while y represents the column coordinate in projective_trans_image. With this convention, affine transformations are a special case of projective transformations in which the last row of HomMat2D is of the form $(0, 0, c)$.

For images of type byte or uint2 the system parameter *'int_zooming'* selects between fast calculation in floating point arithmetics (*'int_zooming'* = *'true'*) and highly accurate floating point arithmetics (*'int_zooming'* = *'false'*). Especially for Interpolation = *'bilinear'*, however, the faster calculation can lead to minor gray value deviations since the faster algorithm is less accurate and only has an accuracy around 10^{-7} times the size of the image. Therefore, when applying large scales *'int_zooming'* = *'false'* is recommended.

Attention

The used coordinate system is the same as in affine_trans_pixel. This means that in fact not HomMat2D is applied but a modified version. Therefore, applying projective_trans_image corresponds to the following chain of transformations, which is applied to each point (Row_i, Col_i) of the image (input and output pixels as homogeneous vectors):

$$\begin{pmatrix} RowTrans_i \\ ColTrans_i \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & -0.5 \\ 0 & 1 & -0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D} \cdot \begin{bmatrix} 1 & 0 & +0.5 \\ 0 & 1 & +0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} Row_i \\ Col_i \\ 1 \end{pmatrix}$$

As an effect, you might get unexpected results when creating projective transformations based on coordinates that are derived from the image, e.g., by operators like area_center_gray. For example, if you use this operator to calculate the center of gravity of a rotationally symmetric image and then rotate the image around this point using hom_mat2d_rotate, the resulting image will not lie on the original one. In such a case, you can compensate this effect by applying the following translations to HomMat2D before using it in projective_trans_image:

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
projective_trans_image(Image, TransImage, HomMat2DAdapted,
'bilinear', 'false', 'false')
```

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

projective_trans_image can be executed on OpenCL devices if the input image does not exceed the maximum size of image objects of the selected device and the parameter TransformDomain is set to *'false'*. The result can diverge slightly from that calculated on the CPU.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **TransImage** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
  Homogeneous projective transformation matrix.
- ▷ **Interpolation** (input_control) ..... string ⇝ *string*
  Interpolation method for the transformation.
  Default: 'bilinear'
  List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear'}

▷ **AdaptImageSize** (input_control) ..... string ⇝ *string*
Adapt the size of the output image automatically?
Default: 'false'
List of values: AdaptImageSize ∈ {'true', 'false'}

▷ **TransformDomain** (input_control) ..... string ⇝ *string*
Should the domain of the input image also be transformed?
Default: 'false'
List of values: TransformDomain ∈ {'true', 'false'}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

vector_to_proj_hom_mat2d, hom_vector_to_proj_hom_mat2d, proj_match_points_ransac, proj_match_points_ransac_guided, hom_mat3d_project

See also

projective_trans_image_size, projective_trans_contour_xld, projective_trans_region, projective_trans_point_2d, projective_trans_pixel

Module

Foundation

projective_trans_image_size (Image : TransImage : HomMat2D, Interpolation, Width, Height, TransformDomain :)

Apply a projective transformation to an image and specify the output image size.

projective_trans_image_size applies the projective transformation (homography) determined by the homogeneous transformation matrix HomMat2D on the input image Image and stores the result into the output image TransImage.

TransImage will be clipped at the output dimensions Height×Width. Apart from this, projective_trans_image_size is identical to its alternative version projective_trans_image.

Attention

The used coordinate system is the same as in affine_trans_pixel. This means that in fact not HomMat2D is applied but a modified version. Therefore, applying projective_trans_image_size corresponds to the following chain of transformations, which is applied to each point (Row_i, Col_i) of the image (input and output pixels as homogeneous vectors):

$$\begin{pmatrix} RowTrans_i \\ ColTrans_i \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & -0.5 \\ 0 & 1 & -0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D} \cdot \begin{bmatrix} 1 & 0 & +0.5 \\ 0 & 1 & +0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} Row_i \\ Col_i \\ 1 \end{pmatrix}$$

As an effect, you might get unexpected results when creating projective transformations based on coordinates that are derived from the image, e.g., by operators like area_center_gray. For example, if you use this operator to calculate the center of gravity of a rotationally symmetric image and then rotate the image around this point using hom_mat2d_rotate, the resulting image will not lie on the original one. In such a case, you can compensate this effect by applying the following translations to HomMat2D before using it in projective_trans_image_size:

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
projective_trans_image_size(Image, TransImage, HomMat2DAdapted,
'bilinear', Width, Height, 'false')
```

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

projective_trans_image_size can be executed on OpenCL devices if the input image does not exceed the maximum size of image objects of the selected device and the parameter TransformDomain is set to *'false'*. The result can diverge slightly from that calculated on the CPU.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **TransImage** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
  Homogeneous projective transformation matrix.
- ▷ **Interpolation** (input_control) ..... string ⇝ *string*
  Interpolation method for the transformation.
  Default: 'bilinear'
  List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear'}
- ▷ **Width** (input_control) ..... extent.x ⇝ *integer*
  Output image width.
- ▷ **Height** (input_control) ..... extent.y ⇝ *integer*
  Output image height.
- ▷ **TransformDomain** (input_control) ..... string ⇝ *string*
  Should the domain of the input image also be transformed?
  Default: 'false'
  List of values: TransformDomain ∈ {'true', 'false'}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

vector_to_proj_hom_mat2d, hom_vector_to_proj_hom_mat2d, proj_match_points_ransac, proj_match_points_ransac_guided, hom_mat3d_project

See also

projective_trans_image, projective_trans_contour_xld, projective_trans_region, projective_trans_point_2d, projective_trans_pixel

Module

Foundation

rotate_image (Image : ImageRotate : Phi, Interpolation :)

Rotate an image about its center.

rotate_image rotates the Image counterclockwise by Phi degrees about its center and stores the result in the output image ImageRotate. The output image has the same size as the input image. The only exception are rotations by 90 and 270 degrees where width and height will be exchanged. The domain of the input image is ignored, i.e., assumed to be the full rectangle of the image. The domain of the output image is the intersection of the transformed rectangle and the rectangle of the output image.

If Phi is a multiple of 90 degrees, this operator is much faster, especially than the general operator affine_trans_image. The effect of the parameter Interpolation is the same as in affine_trans_image. It is ignored for rotations by 90, 180, and 270 degrees. If it is necessary to rotate the domain, too, the operator projective_trans_image must be used.

Additionally, for images of type byte, int2 or uint2 the system parameter *'int_zooming'* selects between fast calculation in fixed point arithmetics (*'int_zooming'* = *'true'*) and highly accurate calculation in floating point arithmetics (*'int_zooming'* = *'false'*). Except for rotations by 90, 180, or 270 degrees, fixed point calculation can lead to minor gray value deviations. Furthermore, the domain of resulting images can differ as well.

Attention

The domain of the input image is ignored, i.e., assumed to be the full rectangle of the image. The domain of the output image is the intersection of the transformed rectangle and the rectangle of the output image. The angle Phi is given in degrees, not in radians. For rotations by 90, 180, and 270 degrees rotate_image is not parallelized internally.

rotate_image can be executed on OpenCL devices if the input image does not exceed the maximum size of image objects of the selected device. Due to numerical reasons, there can be slight differences in the output compared to the execution on the CPU.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
  Input image.
- ▷ **ImageRotate** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
  Rotated image.
- ▷ **Phi** (input_control) ..... angle.deg ⇝ *real* / integer
  Rotation angle.
  Default: 90
  Suggested values: Phi ∈ {90, 180, 270}
  Value range: 0 ≤ Phi ≤ 360
  Minimum increment: 0.001
  Recommended increment: 0.2
- ▷ **Interpolation** (input_control) ..... string ⇝ *string*
  Type of interpolation.
  Default: 'constant'
  List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'constant', 'weighted'}

Example

```
read_image(Image,'monkey')
dev_display (Image)
rotate_image(Image,RotImage,270,'constant')
dev_display (RotImage)
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Alternatives

hom_mat2d_rotate, affine_trans_image

See also

mirror_image

Module

Foundation

zoom_image_factor (Image : ImageZoomed : ScaleWidth, ScaleHeight, Interpolation :)

Zoom an image by a given factor.

zoom_image_factor scales the image Image by a factor of ScaleWidth in width and a factor ScaleHeight in height. The parameter Interpolation determines the type of interpolation used (see affine_trans_image). The domain of the input image is ignored, i.e., assumed to be the full rectangle of the image.

Attention

If the system parameter *'int_zooming'* is set to *'true'*, the internally used integer arithmetic may lead to errors in the following three cases: First, if zoom_image_factor is used on an uint2 or int2 image with high dynamics (i.e. images containing values close to the respective limits) in combination with scale factors smaller than 0.5, then the gray values of the output image may be erroneous. Second, if Interpolation is set to a value other than *'nearest_neighbor'*, a large scale factor is applied, and a large output image is obtained, then undefined gray values at the lower and at the right image border may result. The maximum width B_{max} of this border of undefined gray values can be estimated as $B_{max} = 0.5 \cdot S \cdot I/2^{15}$, where S is the scale factor in one dimension and I is the size of the output image in the corresponding dimension. Third, for real images undefined gray values can occur on all image borders, if Interpolation is set to *'nearest_neighbor'*, and if both the input and output image is large. In all cases, it is recommended to set *'int_zooming'* to *'false'* via the operator set_system.

If *'nearest_neighbor'* is set as interpolation method the results can differ slightly for different image types. This is due to the usage of image type specific optimizations of the applied interpolation algorithm.

zoom_image_factor is not parallelized internally if ScaleWidth = 0.5 and ScaleHeight = 0.5. Further zoom_image_factor is not parallelized internally with Interpolation='*nearest_neighbor*'.

zoom_image_factor can be executed on OpenCL devices if the input image does not exceed the maximum size of image objects of the selected device. Due to numerical reasons, there can be slight differences in the output compared to the execution on the CPU.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Input image.

▷ **ImageZoomed** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Scaled image.

▷ **ScaleWidth** (input_control) ..... extent.x ⇝ *real*
Scale factor for the width of the image.
Default: 0.5
Suggested values: ScaleWidth ∈ {0.25, 0.5, 1.5, 2.0}
Value range: 0.001 ≤ ScaleWidth ≤ 10.0
Minimum increment: 0.001
Recommended increment: 0.1

▷ **ScaleHeight** (input_control) ..... extent.y ⇝ *real*
Scale factor for the height of the image.
Default: 0.5
Suggested values: ScaleHeight ∈ {0.25, 0.5, 1.5, 2.0}
Value range: 0.001 ≤ ScaleHeight ≤ 10.0
Minimum increment: 0.001
Recommended increment: 0.1

▷ **Interpolation** (input_control) ..... string ⇝ *string*
Type of interpolation.
Default: 'constant'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'constant', 'weighted'}

Example

```
read_image(Image,'monkey')
dev_display (Image)
zoom_image_factor (Image, ZoomImage, 0.25, 0.25, 'constant')
dev_display (ZoomImage)
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Alternatives

zoom_image_size, affine_trans_image, hom_mat2d_scale

See also

hom_mat2d_scale, affine_trans_image

Module

Foundation

zoom_image_size (Image : ImageZoom : Width, Height, Interpolation :)

Zoom an image to a given size.

zoom_image_size scales the image Image to the size given by Width and Height. The parameter Interpolation determines the type of interpolation used (see affine_trans_image). The domain of the input image is ignored, i.e., assumed to be the full rectangle of the image.

Attention

If the system parameter *'int_zooming'* is set to *'true'*, the internally used integer arithmetic may lead to errors in the following three cases: First, if zoom_image_size is used on an uint2 or int2 image with high dynamics (i.e. images containing values close to the respective limits) in combination with scale factors (ratio of output to input image size) smaller than 0.5, then the gray values of the output image may be erroneous. Second, if Interpolation is set to a value other than *'nearest_neighbor'*, a large scale factor is applied, and a large output image is obtained, then undefined gray values at the lower and at the right image border may result. The maximum width B_{max} of this border of undefined gray values can be estimated as $B_{max} = 0.5 \cdot S \cdot I/2^{15}$, where S is the scale factor in one dimension and I is the size of the output image in the corresponding dimension. Third, for real images undefined gray values can occur on all image borders, if Interpolation is set to *'nearest_neighbor'*, and if both the input and output image is large. In all cases, it is recommended to set *'int_zooming'* to *'false'* via the operator set_system.

If *'nearest_neighbor'* is set as interpolation method the results can differ slightly for different image types. This is due to the usage of image type specific optimizations of the applied interpolation algorithm.

zoom_image_size is not parallelized internally if Width and Height correspond to half the dimensions of Image. Further zoom_image_size is not parallelized internally with Interpolation='*nearest_neighbor*'.

zoom_image_size can be executed on OpenCL devices if the input image does not exceed the maximum size of image objects of the selected device. Due to numerical reasons, there can be slight differences in the output compared to the execution on the CPU.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Input image.

▷ **ImageZoom** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Scaled image.

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the resulting image.
Default: 512
Suggested values: `Width` ∈ {128, 256, 512}
Value range: 2 ≤ `Width` ≤ 512
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the resulting image.
Default: 512
Suggested values: `Height` ∈ {128, 256, 512}
Value range: 2 ≤ `Height` ≤ 512
Minimum increment: 1
Recommended increment: 10

▷ **Interpolation** (input_control) . . . . . string ⇝ *string*
Type of interpolation.
Default: 'constant'
List of values: `Interpolation` ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'constant', 'weighted'}

Example

```
read_image(Image,'monkey')
dev_display (Image)
zoom_image_size(Image,ZoomImage,200,200,'constant')
dev_display (ZoomImage)
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Alternatives

`zoom_image_factor`, `affine_trans_image`, `hom_mat2d_scale`

See also

`hom_mat2d_scale`, `affine_trans_image`

Module

Foundation

12.8 Inpainting

```
harmonic_interpolation ( Image,
    Region :  InpaintedImage :  Precision :  )
```

Perform a harmonic interpolation on an image region.

The operator harmonic_interpolation reconstructs the destroyed image data of Image inside the region Region by solving the discrete Laplace equation $u_{xx} + u_{yy} = 0$ for the corresponding gray value function u. The unique solution, which exists under Dirichlet boundary conditions given by Image outside of Region, is returned in InpaintedImage.

This technique is called harmonic interpolation since in function theory the solutions of the Laplace equation are referred to as harmonic functions.

If Region touches the border of the gray value matrix of Image and thus some Dirichlet boundary values do not exist, von Neumann boundary conditions are used instead. This means that the gray values are mirrored at the border of Image. If no Dirichlet boundary values exist at all, a constant image with gray value *0* is returned.

The spatial derivatives are discretized as $u_{xx}(x,y) = u(x-1,y) - 2u(x,y) + u(x+1,y)$ and $u_{yy}(x,y) = u(x,y-1) - 2u(x,y) + u(x,y+1)$. The equation is solved by an iterative conjugate gradient solver, which iteratively improves the computational error until the maximum norm of its update step becomes a smaller fraction than Precision of the norm of the input data or a maximum of *1000* iterations is reached. $\texttt{Precision} = 0.01$ thus means a relative computational accuracy of 1%.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **Region** (input_object) ..... region ⇝ *object*
  Inpainting region.
- ▷ **InpaintedImage** (output_object) ..... image(-array) ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **Precision** (input_control) ..... real ⇝ *real*
  Computational accuracy.
  Default: 0.001
  Suggested values: Precision ∈ {0.0, 0.0001, 0.001, 0.01}
  Restriction: Precision >= 0.0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Alternatives

inpainting_ct, inpainting_aniso, inpainting_mcf, inpainting_texture, inpainting_ced

References

L.C. Evans; “Partial Differential Equations”; AMS, Providence; 1998.

W. Hackbusch; “Iterative Lösung großer schwachbesetzter Gleichungssysteme”; Teubner, Stuttgart;1991.

Module

Foundation

inpainting_aniso (Image, Region : InpaintedImage : Mode, Contrast, Theta, Iterations, Rho :)

Perform an inpainting by anisotropic diffusion.

The operator inpainting_aniso uses the anisotropic diffusion according to the model of Perona and Malik, to continue image edges that cross the border of the region Region and to connect them inside of Region.

With this, the structure of the edges in Region will be made consistent with the surrounding image matrix, so that an occlusion of errors or unwanted objects in the input image, a so called inpainting, is less visible to the human beholder, since there remain no obvious artifacts or smudges.

Considering the image as a gray value function u, the algorithm is a discretization of the partial differential equation

$$u_t = div(g(|\nabla u|^2, c)\nabla u)$$

with the initial value $u = u_0$ defined by Image at a time $t_0 = 0$. The equation is iterated Iterations times in time steps of length Theta, so that the output image InpaintedImage contains the gray value function at the time Iterations $\cdot$ Theta.

The primary goal of the anisotropic diffusion, which is also referred to as nonlinear isotropic diffusion, is the elimination of image noise in constant image patches while preserving the edges in the image. The distinction between edges and constant patches is achieved using the threshold Contrast on the magnitude of the gray value differences between adjacent pixels. Contrast is referred to as the contrast parameter and is abbreviated with the letter c. If the edge information is distributed in an environment of the already existing edges by smoothing the edge amplitude matrix, it is furthermore possible to continue edges into the computation area Region. The standard deviation of this smoothing process is determined by the parameter Rho.

The algorithm used is basically the same as in the anisotropic diffusion filter anisotropic_diffusion, except that here, border treatment is not done by mirroring the gray values at the border of Region. Instead, this procedure is only applicable on regions that keep a distance of at least 3 pixels to the border of the image matrix of Image, since the gray values on this band around Region are used to define the boundary conditions for the respective differential equation and thus assure consistency with the neighborhood of Region. Please note that the inpainting progress is restricted to those pixels that are included in the ROI of the input image Image. If the ROI does not include the entire region Region, a band around the intersection of Region and the ROI is used to define the boundary values.

The result of the diffusion process depends on the gray values in the computation area of the input image Image. It must be pointed out that already existing image edges are preserved within Region. In particular, this holds for gray value jumps at the border of Region, which can result for example from a previous inpainting with constant gray value. If the procedure is to be used for inpainting, it is recommended to apply the operator harmonic_interpolation first to remove all unwanted edges inside the computation area and to minimize the gray value difference between adjacent pixels, unless the input image already contains information inside Region that should be preserved.

The variable diffusion coefficient g can be chosen to follow different monotonically decreasing functions with values between *0* and *1* and determines the response of the diffusion process to an edge. With the parameter Mode, the following functions can be selected:

$$g_1(x, c) = \frac{1}{\sqrt{1 + 2\frac{x}{c^2}}}$$

Choosing the function g_1 by setting Mode to *'parabolic'* guarantees that the associated differential equation is parabolic, so that a well-posedness theory exists for the problem and the procedure is stable for an arbitrary step size Theta. In this case however, there remains a slight diffusion even across edges of an amplitude larger than c.

$$g_2(x, c) = \frac{1}{1 + \frac{x}{c^2}}$$

The choice of *'perona-malik'* for Mode, as used in the publication of Perona and Malik, does not possess the theoretical properties of g_1, but in practice it has proved to be sufficiently stable and is thus widely used. The theoretical instability results in a slight sharpening of strong edges.

$$g_3(x, c) = 1 - \exp(-C\frac{c^8}{x^4})$$

The function g_3 with the constant $C = 3.31488$, proposed by Weickert, and selectable by setting Mode to *'weickert'* is an improvement of g_2 with respect to edge sharpening. The transition between smoothing and sharpening happens very abruptly at $x = c^2$.

Furthermore, the choice of the value *'shock'* is possible for `Mode` to select a contrast invariant modification of the anisotropic diffusion. In this variant, the generation of edges is not achieved by variation of the diffusion coefficient g, but the constant coefficient $g = 1$ and thus isotropic diffusion is used. Additionally, a shock filter of type

$$u_t = -sgn(\nabla|\nabla u|)|\nabla u|$$

is applied, which, just like a negative diffusion coefficient, causes a sharpening of the edges, but works independent of the absolute value of $|\nabla u|$. In this mode, `Contrast` does not have the meaning of a contrast parameter, but specifies the ratio between the diffusion and the shock filter part applied at each iteration step. Hence, the value *0* would correspond to pure isotropic diffusion, as used in the operator `isotropic_diffusion`. The parameter is scaled in such a way that diffusion and sharpening cancel each other out for `Contrast` = *1*. A value `Contrast` > *1* should not be used, since it would make the algorithm unstable.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **`Region`** (input_object) ..... region ⇝ *object*
  Inpainting region.
- ▷ **`InpaintedImage`** (output_object) ..... image(-array) ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **`Mode`** (input_control) ..... string ⇝ *string*
  Type of edge sharpening algorithm.
  Default: 'weickert'
  List of values: `Mode` ∈ {'weickert', 'perona-malik', 'parabolic', 'shock'}
- ▷ **`Contrast`** (input_control) ..... real ⇝ *real*
  Contrast parameter.
  Default: 5.0
  Suggested values: `Contrast` ∈ {0.5, 2.0, 5.0, 10.0, 20.0, 50.0, 100.0}
  Restriction: Contrast > 0
- ▷ **`Theta`** (input_control) ..... real ⇝ *real*
  Step size.
  Default: 0.5
  Suggested values: `Theta` ∈ {0.5, 1.0, 5.0, 10.0, 30.0, 100.0}
  Restriction: Theta > 0
- ▷ **`Iterations`** (input_control) ..... integer ⇝ *integer*
  Number of iterations.
  Default: 10
  Suggested values: `Iterations` ∈ {1, 3, 10, 100, 500}
  Restriction: Iterations >= 1
- ▷ **`Rho`** (input_control) ..... real ⇝ *real*
  Smoothing coefficient for edge information.
  Default: 3.0
  Suggested values: `Rho` ∈ {0.0, 0.1, 0.5, 1.0, 3.0, 10.0}
  Restriction: Rho >= 0

Example

```
read_image (Image, 'fabrik')
gen_rectangle1 (Rectangle, 270, 180, 320, 230)
harmonic_interpolation (Image, Rectangle, InpaintedImage, 0.01)
inpainting_aniso (InpaintedImage, Rectangle, InpaintedImage2, \
                  'perona-malik', 5.0, 100, 50, 0.5)
dev_display(InpaintedImage2)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Alternatives

harmonic_interpolation, inpainting_ct, inpainting_mcf, inpainting_texture, inpainting_ced

References

J. Weickert; "Anisotropic Diffusion in Image Processing"; PhD Thesis; Fachbereich Mathematik, Universität Kaiserslautern; 1996.

P. Perona, J. Malik; "Scale-space and edge detection using anisotropic diffusion"; Transactions on Pattern Analysis and Machine Intelligence 12(7), pp. 629-639; IEEE; 1990.

G. Aubert, P. Kornprobst; "Mathematical Problems in Image Processing"; Applied Mathematical Sciences 147; Springer, New York; 2002.

Module

Foundation

inpainting_ced (Image, Region : InpaintedImage : Sigma, Rho, Theta, Iterations :)

Perform an inpainting by coherence enhancing diffusion.

The operator inpainting_ced performs an anisotropic diffusion process on the region Region of the input image Image with the objective of completing discontinuous image edges diffusively by increasing the coherence of the image structures contained in Image and without smoothing these edges perpendicular to their dominating direction. The mechanism is the same as in the operator coherence_enhancing_diff, which is based on a discretization of the anisotropic diffusion equation

$$u_t = div(G(u)\nabla u)$$

formulated by Weickert. With a 2×2 coefficient matrix G that depends on the gray values in Image, this is an enhancement of the mean curvature flow or intrinsic heat equation

$$u_t = div(\frac{\nabla u}{|\nabla u|})|\nabla u| = curv(u)|\nabla u|$$

on the gray value function u defined by the input image Image at a time $t_0 = 0$. The smoothing operator mean_curvature_flow is a direct application of the mean curvature flow equation. With the operator inpainting_mcf, it can also be used for image inpainting. The discrete diffusion equation is solved in Iterations time steps of length Theta, so that the output image InpaintedImage contains the gray value function at the time Iterations $\cdot$ Theta.

To detect the image direction more robustly, in particular on noisy input data, an additional isotropic smoothing step can precede the computation of the gray value gradients. The parameter Sigma determines the magnitude of the smoothing by means of the standard deviation of a corresponding Gaussian convolution kernel, as used in the operator isotropic_diffusion for isotropic image smoothing.

Similar to the operator inpainting_mcf, the structure of the image data in Region is simplified by smoothing the level lines of Image. By this, image errors and unwanted objects can be removed from the image, while the edges in the neighborhood are extended continuously. This procedure is called image inpainting. The objective is to introduce a minimum amount of artifacts or smoothing effects, so that the image manipulation is least visible to a human beholder.

While the matrix G is given by

$$G_{MCF}(u) = I - \frac{1}{|\nabla u|^2}\nabla u(\nabla u)^T,$$

in the case of the operator inpainting_mcf, where I denotes the unit matrix, G_{MCF} is again smoothed componentwise by a Gaussian filter of standard deviation Rho for coherence_enhancing_diff. Then, the final coefficient matrix

$$G_{CED} = g_1\left((\lambda_1 - \lambda_2)^2\right) w_1 (w_1)^T + g_2\left((\lambda_1 - \lambda_2)^2\right) w_2 (w_2)^T$$

is constructed from the eigenvalues λ_1, λ_2 and eigenvectors w_1, w_2 of the resulting intermediate matrix, where the functions

$$\begin{aligned} g_1(p) &= 0.001 \\ g_2(p) &= 0.001 + 0.999 \exp\left(\frac{-1}{p}\right) \end{aligned}$$

were determined empirically and taken from the publication of Weickert.

Hence, the diffusion direction in mean_curvature_flow is only determined by the local direction of the gray value gradient, while G_{CED} considers the macroscopic structure of the image objects on the scale Rho and the magnitude of the diffusion in coherence_enhancing_diff depends on how well this structure is defined.

To achieve the highest possible consistency of the newly created edges with the image data from the neighborhood, the gray values are not mirrored at the border of Region to compute the convolution with the smoothing filter mask of scale Rho on the pixels close to the border, although this would be the common approach for filter operators. Instead, the existence of gray values on a band of width $ceil(3.1 * Rho) + 2$ pixels around Region is presumed and these values are used in the convolution. This means that Region must keep this much distance to the border of the image matrix Image. By involving the gray values and directional information from this extended area, it can be achieved that the continuation of the edges is not only continuous, but also smooth, which means without kinks. Please note that the inpainting progress is restricted to those pixels that are included in the ROI of the input image Image. If the ROI does not include the entire region Region, a band around the intersection of Region and the ROI is used to define the boundary values.

To decrease the number of iterations required for attaining a satisfactory result, it may be useful to initialize the gray value matrix in Region with the harmonic interpolant, a continuous function of minimal curvature, by applying the operator harmonic_interpolation to Image before calling inpainting_ced.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **Region** (input_object) ..... region ⇝ *object*
  Inpainting region.
- ▷ **InpaintedImage** (output_object) ..... image(-array) ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **Sigma** (input_control) ..... real ⇝ *real*
  Smoothing for derivative operator.
  Default: 0.5
  Suggested values: Sigma ∈ {0.0, 0.1, 0.5, 1.0}
  Restriction: Sigma >= 0
- ▷ **Rho** (input_control) ..... real ⇝ *real*
  Smoothing for diffusion coefficients.
  Default: 3.0
  Suggested values: Rho ∈ {0.0, 1.0, 3.0, 5.0, 10.0, 30.0}
  Restriction: Rho >= 0
- ▷ **Theta** (input_control) ..... real ⇝ *real*
  Time step.
  Default: 0.5
  Suggested values: Theta ∈ {0.1, 0.2, 0.3, 0.4, 0.5}
  Restriction: 0 < Theta <= 0.5

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 10
Suggested values: Iterations ∈ {1, 5, 10, 20, 50, 100, 500}
Restriction: Iterations >= 1

Example

```
read_image (Image, 'fabrik')
gen_rectangle1 (Rectangle, 270, 180, 320, 230)
harmonic_interpolation (Image, Rectangle, InpaintedImage, 0.01)
inpainting_ced (InpaintedImage, Rectangle, InpaintedImage2, \
                0.5, 3.0, 0.5, 1000)
dev_display(InpaintedImage2)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Alternatives

harmonic_interpolation, inpainting_ct, inpainting_aniso, inpainting_mcf, inpainting_texture

References

J. Weickert, V. Hlavac, R. Sara; "Multiscale texture enhancement"; Computer analysis of images and patterns, Lecture Notes in Computer Science, Vol. 970, pp. 230-237; Springer, Berlin; 1995.

J. Weickert, B. ter Haar Romeny, L. Florack, J. Koenderink, M. Viergever; "A review of nonlinear diffusion filtering"; Scale-Space Theory in Computer Vision, Lecture Notes in Comp. Science, Vol. 1252, pp. 3-28; Springer, Berlin; 1997.

Module

Foundation

inpainting_ct (Image, Region : InpaintedImage : Epsilon, Kappa, Sigma, Rho, ChannelCoefficients :)

Perform an inpainting by coherence transport.

The operator inpainting_ct inpaints a missing region Region of an image Image by transporting image information from the region's boundary along the coherence direction into this region.

Since this operator's basic concept is inpainting by continuing broken contour lines, the image content and inpainting region must be such that this idea makes sense. That is, if a contour line hits the region to inpaint at a pixel p, there should be some opposite point q where this contour line continues so that the continuation of contour lines from two opposite sides can succeed. In cases where there is less geometry in the image, a diffusion-based inpainter, e.g., harmonic_interpolation may yield better results. Alternatively, Kappa can be set to *0*. An extreme situation with little global geometries are pure textures. Then the idea behind this operator will fail to produce good results (think of a checkerboard with a big region to inpaint relative to the checker fields). For these kinds of images, a texture-based inpainting, e.g., inpainting_texture, can be used instead.

The operator uses a so-called upwind scheme to assign gray values to the missing pixels, i.e.,:

- The order of the pixels to process is given by their Euclidean distance to the boundary of the region to inpaint.
- A new value u_i is computed as a weighted average of already known values u_j within a disc of radius Epsilon around the current pixel. The disc is restricted to already known pixels.
- The size of this scheme's mask depends on Epsilon.

The initially used image data comes from a stripe of thickness Epsilon around the region to inpaint. Thus, Epsilon must be at least 1 for the scheme to work, but should be greater. The maximum value for Epsilon depends on the gray values that should be transported into the region. Choosing Epsilon = *5* can be used in many cases.

Since the goal is to close broken contour lines, the direction of the level lines must be estimated and used in the weight. This estimated direction is called the coherence direction, and is computed by means of the structure tensor S.

$$S = G_\rho * DvDv^T$$

and

$$v = G_\sigma * u$$

where $*$ denotes the convolution, u denotes the gray value image, D the derivative and G Gaussian kernels with standard deviation σ and ρ. These standard deviations are defined by the operator's parameters Sigma and Rho. Sigma should have the size of the noise or unimportant little objects, which are then not considered in the estimation step by the pre-smoothing. Rho gives the size of the window around a pixel that will be used for direction estimation. The coherence direction c then is given by the eigendirection of S with respect to the minimal eigenvalue λ, i.e.

$$Sc = \lambda c, |c| = 1$$

For multichannel or color images, the scheme above is applied to each channel separately, but the weights must be the same for all channels to propagate information in the same direction. Since the weight depends on the coherence direction, the common direction is given by the eigendirection of a composite structure tensor. If $u_1, ..., u_n$ denote the n channels of the image, the channel structure tensors $S_1, ..., S_n$ are computed and then combined to the composite structure tensor S.

$$S = \sum_{i=1}^{n} a_i S_i$$

The coefficients a_i are passed in ChannelCoefficients, which is a tuple of length n or length 1. If the tuple length is 1, the arithmetic mean is used, i.e., $a_i = 1/n$. If the length of ChannelCoefficients matches the number of channels, the a_i are set to

$$a_i = \frac{\texttt{ChannelCoefficients}_i}{\sum_{i=1}^{n} \texttt{ChannelCoefficients}_i}$$

in order to get a well-defined convex combination. Hence, the ChannelCoefficients must be greater than or equal to zero and their sum must be greater than zero. If the tuple length is neither 1 nor the number of channels or the requirement above is not satisfied, the operator returns an error message.

The purpose of using other ChannelCoefficients than the arithmetic mean is to adapt to different color codes. The coherence direction is a geometrical information of the composite image, which is given by high contrasts such as edges. Thus the more contrast a channel has, the more geometrical information it contains, and consequently the greater its coefficient should be chosen (relative to the others). For RGB images, *[0.299, 0.587, 0.114]* is a good choice.

The weight in the scheme is the product of a directional component and a distance component. If p is the 2D coordinate vector of the current pixel to be inpainted and q the 2D coordinate of a pixel in the neighborhood (the disc restricted to already known pixels), the directional component measures the deviation of the vector $p - q$ from the coherence direction. If the deviation exponentially scaled by β is large, a low directional component is assigned, whereas if it is small, a large directional component is assigned. β is controlled by Kappa (in percent):

$$\beta = 20 * \texttt{Epsilon} * \texttt{Kappa}/100$$

Kappa defines how important it is to propagate information along the coherence direction, so a large Kappa yields sharp edges, while a low Kappa allows for more diffusion.

A special case is when Kappa is zero: In this case the directional component of the weight is constant (one). The direction estimation step is then skipped to save computational costs and the parameters Sigma, Rho, ChannelCoefficients become meaningless, i.e, the propagation of information is not based on the structures visible in the image.

The distance component is $1/|p-q|$. Consequently, if q is far away from p, a low distance component is assigned, whereas if it is near to p, a high distance component is assigned.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **Region** (input_object) ..... region ⇝ *object*
  Inpainting region.
- ▷ **InpaintedImage** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **Epsilon** (input_control) ..... number ⇝ *real*
  Radius of the pixel neighborhood.
  Default: 5.0
  Value range: $1.0 \leq$ Epsilon ≤ 20.0
  Minimum increment: 1.0
  Recommended increment: 1.0
- ▷ **Kappa** (input_control) ..... number ⇝ *real*
  Sharpness parameter in percent.
  Default: 25.0
  Value range: $0.0 \leq$ Kappa ≤ 100.0
  Minimum increment: 1.0
  Recommended increment: 1.0
- ▷ **Sigma** (input_control) ..... number ⇝ *real*
  Pre-smoothing parameter.
  Default: 1.41
  Value range: $0.0 \leq$ Sigma ≤ 20.0
  Minimum increment: 0.001
  Recommended increment: 0.01
- ▷ **Rho** (input_control) ..... number ⇝ *real*
  Smoothing parameter for the direction estimation.
  Default: 4.0
  Value range: $0.001 \leq$ Rho ≤ 20.0
  Minimum increment: 0.001
  Recommended increment: 0.01
- ▷ **ChannelCoefficients** (input_control) ..... number(-array) ⇝ *real*
  Channel weights.
  Default: 1

Example

```
read_image (Image, 'claudia')
gen_circle (Circle, 333, 164, 35)
inpainting_ct (Image, Circle, InpaintedImage, 15, 25, 1.5, 3,1.0)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Alternatives

harmonic_interpolation, inpainting_aniso, inpainting_mcf, inpainting_ced, inpainting_texture

References

Folkmar Bornemann, Tom März: "Fast Image Inpainting Based On Coherence Transport"; Journal of Mathematical Imaging and Vision; vol. 28, no. 3; pp. 259-278; 2007.

Module

Foundation

inpainting_mcf (Image, Region : InpaintedImage : Sigma, Theta, Iterations :)

Perform an inpainting by smoothing of level lines.

The operator inpainting_mcf extends the image edges that adjoin the region Region of the input image Image into the interior of Region and connects their ends by smoothing the level lines of the gray value function of Image.

This happens through the application of the mean curvature flow or intrinsic heat equation

$$u_t = div(\frac{\nabla u}{|\nabla u|})|\nabla u| = curv(u)|\nabla u|$$

on the gray value function u defined in the region Region by the input image Image at a time $t_0 = 0$. The discretized equation is solved in Iterations time steps of length Theta, so that the output image InpaintedImage contains the gray value function at the time Iterations · Theta.

A stationary state of the mean curvature flow equation, which is also the basis of the operator mean_curvature_flow, has the special property that the level lines of u all have the curvature *0*. This means that after sufficiently many iterations there are only straight edges left inside the computation area of the output image InpaintedImage. By this, the structure of objects inside of Region can be simplified, while the remaining edges are continuously connected to those of the surrounding image matrix. This allows for a removal of image errors and unwanted objects in the input image, a so called image inpainting, which is only weakly visible to a human beholder since there remain no obvious artifacts or smudges.

To detect the image direction more robustly, in particular on noisy input data, an additional isotropic smoothing step can precede the computation of the gray value gradients. The parameter Sigma determines the magnitude of the smoothing by means of the standard deviation of a corresponding Gaussian convolution kernel, as used in the operator isotropic_diffusion for isotropic image smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **Region** (input_object) ..... region ⇝ *object*
  Inpainting region.
- ▷ **InpaintedImage** (output_object) ..... image(-array) ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **Sigma** (input_control) ..... real ⇝ *real*
  Smoothing for derivative operator.
  Default: 0.5
  Suggested values: Sigma ∈ {0.0, 0.1, 0.5, 1.0}
  Restriction: Sigma >= 0
- ▷ **Theta** (input_control) ..... real ⇝ *real*
  Time step.
  Default: 0.5
  Suggested values: Theta ∈ {0.1, 0.2, 0.3, 0.4, 0.5}
  Restriction: 0 < Theta <= 0.5

▷ **Iterations** (input_control) . . . . . integer ⇝ *integer*
Number of iterations.
Default: 10
Suggested values: Iterations ∈ {1, 5, 10, 20, 50, 100, 500}
Restriction: Iterations >= 1

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Alternatives

harmonic_interpolation, inpainting_ct, inpainting_aniso, inpainting_ced, inpainting_texture

References

M. G. Crandall, P. Lions; "Convergent Difference Schemes for Nonlinear Parabolic Equations and Mean Curvature Motion"; Numer. Math. 75 pp. 17-41; 1996.

G. Aubert, P. Kornprobst; "Mathematical Problems in Image Processing"; Applied Mathematical Sciences 147; Springer, New York; 2002.

Module

Foundation

inpainting_texture (Image, Region : InpaintedImage : MaskSize, SearchSize, Anisotropy, PostIteration, Smoothness :)

Perform an inpainting by texture propagation.

The operator inpainting_texture is used for removing large objects and image errors from the region Region of the input image Image. Image blocks of side length MaskSize are copied from the intact part of the image to the border of the computation area, until that area has been filled up with new gray values. This process is called image inpainting. Hence, the computation area is also referred to as the inpainting area and is reduced with every inserted rectangle, starting with Region. Let the center of the current block be at the point x. Since x is always chosen from the border of the inpainting area the current block overlaps with the known or already filled-in gray values. The gray value correlation with the overlapping part of this block is used to determine which other image block fits at the position x. As the correlation function, the sum of the squared gray value differences is used. The image blocks that are taken into account for the correlation, and hence as candidates for the data source of the next inpainting step, are called comparison blocks. The search area for suitable gray value patterns in which the centers of the comparison blocks is searched is limited to a square of side length $2 \cdot$ SearchSize around the point x.

On the one hand, the order in which the pixels of Region are filled in depends on the size of the overlapping area and thus the number of pixels available for the correlation. On the other hand, the absolute value of the derivative of the gray value function tangential to the border of the computation area is also considered. The larger the value of the parameter Anisotropy is, the more the points in which the derivative is large are preferred. This way it can be achieved that, e.g., straight lines which are represented by large gradients, are continued through the entire computation area without being interrupted by the inpainting of image structures from other parts of the border when the size of the inpainting area becomes small. On the other hand, a large value of Anisotropy also means that possible phantom edges, i.e., unwanted random structures that have developed during the inpainting process, are also propagated and the magnitude of those image disturbances is increased.

To confine the formation of such artifacts, the original algorithm can be extended by a post-iteration step that selects smooth and inconspicuous image patches as data sources for the inpainting. If the parameter PostIteration is set to *'min_grad'* the sum of the squares of the gray value gradients is minimized on the comparison blocks. With the value *'min_range_extension'*, the growth of the gray value interval of the comparison blocks with respect to the reference block around the point x is minimized. If PostIteration has the value *'none'* no post-iteration is performed. The choice of feasible blocks for this minimization process is determined by the parameter

Smoothness, which is an upper limit to the permitted increase of the mean absolute gray value difference between the comparison blocks and the reference block with respect to the block that was selected by the original algorithm. With increasing value of Smoothness, the inpainting result becomes smoother and loses structure. The matching accuracy of the selected comparison blocks decreases. If Smoothness is set to *0*, the post-iteration only considers comparison blocks with an equally high correlation to the reference block.

If the inpainting process cannot be completed because there are points x, for which no complete block of intact gray value information is contained in the search area of size SearchSize, the remaining pixels keep their initial gray value and the ROI of the output image InpaintedImage is reduced by the region that could not be processed. If the structure size of the ROI of Image or of the computation area Region is smaller than MaskSize, the execution time of the algorithm can increase extremely. Hence, it is recommended to only use clearly structured input regions.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **Region** (input_object) ..... region ⇝ *object*
  Inpainting region.
- ▷ **InpaintedImage** (output_object) ..... image(-array) ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **MaskSize** (input_control) ..... integer ⇝ *integer*
  Size of the inpainting blocks.
  Default: 9
  Suggested values: MaskSize ∈ {7, 9, 11, 15, 21}
  Restriction: MaskSize >= 3 && odd(MaskSize)
- ▷ **SearchSize** (input_control) ..... integer ⇝ *integer*
  Size of the search window.
  Default: 30
  Suggested values: SearchSize ∈ {15, 30, 50, 100, 1000}
  Restriction: 2 * SearchSize > MaskSize
- ▷ **Anisotropy** (input_control) ..... real ⇝ *real*
  Influence of the edge amplitude on the inpainting order.
  Default: 1.0
  Suggested values: Anisotropy ∈ {0.0, 0.01, 0.1, 0.5, 1.0, 10.0}
  Restriction: Anisotropy >= 0
- ▷ **PostIteration** (input_control) ..... string ⇝ *string*
  Post-iteration for artifact reduction.
  Default: 'none'
  List of values: PostIteration ∈ {'none', 'min_grad', 'min_range_extension'}
- ▷ **Smoothness** (input_control) ..... real ⇝ *real*
  Gray value tolerance for post-iteration.
  Default: 1.0
  Suggested values: Smoothness ∈ {0.0, 0.1, 0.2, 0.5, 1.0}
  Restriction: Smoothness >= 0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Module

Foundation

12.9 Lines

bandpass_image (Image : ImageBandpass : FilterType :)

Edge extraction using bandpass filters.

bandpass_image serves as an edge filter. It applies a linear filter with the following convolution mask to Image:

FilterType: *'lines'*
In contrast to the edge operator sobel_amp this filter detects lines instead of edges, i.e., two closely adjacent edges.

$$\begin{array}{rrrrr} 0 & -2 & -2 & -2 & 0 \\ -2 & 0 & 3 & 0 & -2 \\ -2 & 3 & 12 & 3 & -2 \\ -2 & 0 & 3 & 0 & -2 \\ 0 & -2 & -2 & -2 & 0 \end{array}$$

At the border of the image the gray values are mirrored. Over- and underflows of gray values are clipped. The resulting images are returned in ImageBandpass.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2
  Input images.
- ▷ **ImageBandpass** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2
  Bandpass-filtered images.
- ▷ **FilterType** (input_control) ..... string ⇝ *string*
  Filter type: currently only 'lines' is supported.
  Default: 'lines'
  List of values: FilterType ∈ {'lines'}

Example

```
bandpass_image(Image,&LineImage,"lines");
threshold(LineImage,&Lines,60.0,255.0);
skeleton(Lines,&ThinLines);
```

Result

bandpass_image returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

threshold, skeleton

Alternatives

convol_image, topographic_sketch, texture_laws

See also

highpass_image, gray_skeleton

Module

Foundation

lines_color (Image : Lines : Sigma, Low, High, ExtractWidth, CompleteJunctions :)

Detect color lines and their width.

lines_color extracts color lines from the input image Image and returns the extracted lines as subpixel precise XLD-contours in Lines. Color lines are defined as dark lines in the amplitude image of the color edge filter (see edges_color). lines_color always uses the Canny color edge filter. Hence, the required partial derivatives of the image are always computed by convolution with the respective partial derivatives of the Gaussian smoothing masks (see derivate_gauss). The corresponding smoothing is determined by the parameter Sigma.

By defining color lines as dark lines in the amplitude image, in contrast to lines_gauss, for single-channel images no distinction is made whether the lines are darker or brighter than their surroundings. Furthermore, lines_color also returns staircase lines, i.e., lines for which the gray value of the lines lies between the gray values in the surrounding area to the left and right sides of the line. In multi-channel images, the above definition allows each channel to have a different line type. For example, in a three-channel image the first channel may have a dark line, the second channel a bright line, and the third channel a staircase line at the same position.

If ExtractWidth is set to *'true'* the line width is extracted for each line point. Because the line extractor is unable to extract certain junctions because of differential geometric reasons, it tries to extract these by different means if CompleteJunctions is set to *'true'*.

lines_color links the line points into lines by using an algorithm similar to a hysteresis threshold operation, which is also used in lines_gauss and edges_color_sub_pix. Points with an amplitude larger than High are immediately accepted as belonging to a line, while points with an amplitude smaller than Low are rejected. All other points are accepted as lines if they are connected to accepted line points (see also lines_gauss). Here, amplitude means the line amplitude of the dark line (see lines_gauss and lines_facet). This value corresponds to the third directional derivative of the smoothed input image in the direction perpendicular to the line.

For the choice of the thresholds High and Low one has to keep in mind that the third directional derivative depends on the amplitude and width of the line as well as the choice of Sigma. The value of the third derivative depends linearly on the amplitude, i.e., the larger the amplitude, the larger the response. For the width of the line there is an inverse dependence: The wider the line is, the smaller the response gets. This holds analogously for the dependence on Sigma: The larger Sigma is chosen, the smaller the second derivative will be. This means that for larger smoothing correspondingly smaller values for High and Low should be chosen.

The extracted lines are returned in a topologically sound data structure in Lines. This means that lines are correctly split at junction points.

lines_color defines the following attributes for each line point if ExtractWidth was set to *'false'*:

- *'angle'* The angle of the direction perpendicular to the line (oriented such that the normal vectors point to the right side of the line as the line is traversed from start to end point; the angles are given with respect to the row axis of the image.)
- *'response'*: The magnitude of the second derivative

If ExtractWidth was set to *'true'*, additionally the following attributes are defined:

- *'width_left'*: The line width to the left of the line
- *'width_right'*: The line width to the right of the line

Use get_contour_attrib_xld to obtain attribute values. See the operator reference of get_contour_attrib_xld for further information about contour attributes.

Attention

In general, but in particular if the line width is to be extracted, Sigma $\geq w/\sqrt{3}$ should be selected, where w is the width (half the diameter) of the lines in the image. As the lowest allowable value Sigma $\geq w/2.5$ *must* be

selected. If, for example, lines with a width of 4 pixels (diameter 8 pixels) are to be extracted, Sigma $\geq$ 2.3 should be selected. If it is expected that staircase lines are present in at least one channel, and if such lines should be extracted, in addition to the above restriction, Sigma $\leq w$ should be selected. This is necessary because staircase lines turn into normal step edges for large amounts of smoothing, and therefore no longer appear as dark lines in the amplitude image of the color edge filter.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
  Input image.
- ▷ **Lines** (output_object) . . . . . xld_cont-array ⇝ *object*
  Extracted lines.
- ▷ **Sigma** (input_control) . . . . . number ⇝ *real* / integer
  Amount of Gaussian smoothing to be applied.
  Default: 1.5
  Suggested values: Sigma ∈ {1, 1.2, 1.5, 1.8, 2, 2.5, 3, 4, 5}
  Recommended increment: 0.1
  Restriction: Sigma > 0.0
- ▷ **Low** (input_control) . . . . . number ⇝ *real* / integer
  Lower threshold for the hysteresis threshold operation.
  Default: 3
  Suggested values: Low ∈ {0, 0.5, 1, 2, 3, 4, 5, 8, 10}
  Value range: 0 ≤ Low
  Recommended increment: 0.5
- ▷ **High** (input_control) . . . . . number ⇝ *real* / integer
  Upper threshold for the hysteresis threshold operation.
  Default: 8
  Suggested values: High ∈ {0, 0.5, 1, 2, 3, 4, 5, 8, 10, 12, 15, 18, 20, 25}
  Value range: 0 ≤ High
  Recommended increment: 0.5
  Restriction: High >= Low
- ▷ **ExtractWidth** (input_control) . . . . . string ⇝ *string*
  Should the line width be extracted?
  Default: 'true'
  List of values: ExtractWidth ∈ {'true', 'false'}
- ▷ **CompleteJunctions** (input_control) . . . . . string ⇝ *string*
  Should junctions be added where they cannot be extracted?
  Default: 'true'
  List of values: CompleteJunctions ∈ {'true', 'false'}

Complexity

The amount of temporary memory required is dependent on the height H of the domain of Image.

Result

lines_color returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

gen_polygons_xld

Alternatives

lines_gauss, lines_facet

See also

edges_color, edges_color_sub_pix

References

C. Steger: "Subpixel-Precise Extraction of Lines and Edges"; International Archives of Photogrammetry and Remote Sensing, vol. XXXIII, part B3; pp. 141-156; 2000.

C. Steger: "An Unbiased Detector of Curvilinear Structures"; IEEE Transactions on Pattern Analysis and Machine Intelligence; vol. 20, no. 2; pp. 113-125; 1998.

C. Steger: "Unbiased Extraction of Curvilinear Structures from 2D and 3D Images"; Herbert Utz Verlag, München; 1998.

Module

2D Metrology

lines_facet (Image : Lines : MaskSize, Low, High, LightDark :)

Detection of lines using the facet model.

The operator lines_facet can be used to extract lines (curvilinear structures) from the image Image. The extracted lines are returned in Lines as subpixel precise XLD-contours. The parameter LightDark determines, whether bright or dark lines are extracted.

The extraction is done by using the facet model, i.e., a least squares fit, to determine the parameters of a quadratic polynomial in x and y for each point of the image. The parameter MaskSize determines the size of the window used for the least squares fit. Larger values of MaskSize lead to a larger smoothing of the image, but can lead to worse localization of the line. The parameters of the polynomial are used to calculate the line direction for each pixel. Pixels which exhibit a local maximum in the second directional derivative perpendicular to the line direction are marked as line points. The line points found in this manner are then linked to contours. This is done by immediately accepting line points that have a second derivative larger than High. Points that have a second derivative smaller than Low are rejected. All other line points are accepted if they are connected to accepted points by a connected path. This is similar to a hysteresis threshold operation with infinite path length (see hysteresis_threshold). However, this function is not used internally since it does not allow the extraction of subpixel precise contours.

The gist of how to select the thresholds in the description of lines_gauss also holds for this operator. A value of Sigma = 1.5 there roughly corresponds to a MaskSize of 5 here.

The extracted lines are returned in a topologically sound data structure in Lines. This means that lines are correctly split at junction points.

lines_facet defines the following attributes for each line point:

'angle': The angle of the direction perpendicular to the line

'response': The magnitude of the second derivative

Use get_contour_attrib_xld to obtain attribute values. See the operator reference of get_contour_attrib_xld for further information about contour attributes.

Attention

The smaller the filter size MaskSize is chosen, the more short, fragmented lines will be extracted. This can lead to considerably longer execution times.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
  Input image.
- ▷ **Lines** (output_object) ..... xld_cont-array ⇝ *object*
  Extracted lines.

▷ **MaskSize** (input_control) . . . . . integer ⇝ *integer*
Size of the facet model mask.
Default: 5
List of values: MaskSize ∈ {3, 5, 7, 9, 11}

▷ **Low** (input_control) . . . . . number ⇝ *real* / integer
Lower threshold for the hysteresis threshold operation.
Default: 3
Suggested values: Low ∈ {0, 0.5, 1, 2, 3, 4, 5, 8, 10}
Value range: 0 ≤ Low
Recommended increment: 0.5

▷ **High** (input_control) . . . . . number ⇝ *real* / integer
Upper threshold for the hysteresis threshold operation.
Default: 8
Suggested values: High ∈ {0, 0.5, 1, 2, 3, 4, 5, 8, 10, 12, 15, 18, 20, 25}
Value range: 0 ≤ High
Recommended increment: 0.5
Restriction: High >= Low

▷ **LightDark** (input_control) . . . . . string ⇝ *string*
Extract bright or dark lines.
Default: 'light'
List of values: LightDark ∈ {'dark', 'light'}

Example

```
* Detection of lines in an aerial image
read_image(Image,'mreut4_3')
lines_facet(Image,Lines,5,3,8,'light')
dev_display(Lines)
```

Complexity

Let A be the number of pixels in the domain of Image. Then the runtime complexity is $O(A * \texttt{MaskSize})$.

The amount of temporary memory required is dependent on the height H of the domain of Image and the width W of Image. Let $S = W * H$, then lines_facet requires at least $55 * S$ bytes of temporary memory during execution.

Result

lines_facet returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

gen_polygons_xld

Alternatives

lines_gauss

See also

bandpass_image, dyn_threshold, topographic_sketch

References

A. Busch: "Fast Recognition of Lines in Digital Images Without User-Supplied Parameters". In H. Ebner, C. Heipke, K.Eder, eds., "Spatial Information from Digital Photogrammetry and Computer Vision", International Archives of Photogrammetry and Remote Sensing, Vol. 30, Part 3/1, pp. 91-97, 1994.

Module

2D Metrology

lines_gauss (Image : Lines : Sigma, Low, High, LightDark, ExtractWidth, LineModel, CompleteJunctions :)

Detect lines and their width.

The operator lines_gauss can be used to extract lines (curvilinear structures) from the image Image. The extracted lines are returned in Lines as subpixel precise XLD-contours.

The parameter LightDark determines, whether bright (*'light'*) or dark (*'dark'*) lines are extracted.

If ExtractWidth is set to *'true'* the line width is extracted for each line point.

LineModel determines if and how the position and width of asymmetric lines (lines having different contrast on each side of the line) should be compensated. The following values can be set for LineModel:

'bar-shaped': Bar-shaped line model. Covers most use cases.

'parabolic': Parabolic line model. Can be used for the extraction of backlit tubular objects (e.g., blood vessels in X-ray images) where the lines appear very sharp.

'gaussian': Gaussian line model. Can be used for the extraction of backlit tubular objects (e.g., blood vessels in X-ray images) where the lines appear less sharp.

'none': The effects of asymmetric lines are not compensated.

The parameter LineModel is only effective if ExtractWidth=*'true'*.

Because the line extractor is unable to extract certain junctions because of differential geometric reasons, it tries to extract these by different means if CompleteJunctions is set to *'true'*.

The extraction is done by using partial derivatives of a Gaussian smoothing kernel to determine the parameters of a quadratic polynomial in x and y for each point of the image. The parameter Sigma determines the amount of smoothing to be performed. Larger values of Sigma lead to a larger smoothing of the image, but can lead to worse localization of the line. Generally, the localization will be much better than that of lines returned by lines_facet with comparable parameters. The parameters of the polynomial are used to calculate the line direction for each pixel. Pixels which exhibit a local maximum in the second directional derivative perpendicular to the line direction are marked as line points. The line points found in this manner are then linked to contours. This is done by immediately accepting line points that have a second derivative larger than High. Points that have a second derivative smaller than Low are rejected. All other line points are accepted if they are connected to accepted points by a connected path. This is similar to a hysteresis threshold operation with infinite path length (see hysteresis_threshold), here, however contours are extracted with subpixel precision.

For the choice of the thresholds High and Low one has to keep in mind that the second directional derivative depends on the amplitude and width of the line as well as the choice of Sigma. The value of the second derivative depends linearly on the amplitude, i.e., the larger the amplitude, the larger the response. For the width of the line there is an approximately inverse exponential dependence: The wider the line is, the smaller the response gets. This holds analogously for the dependence on Sigma: The larger Sigma is chosen, the smaller the second derivative will be. This means that for larger smoothing correspondingly smaller values for High and Low have to be chosen. Two examples help to illustrate this: If 5 pixel wide lines with an amplitude larger than 100 are to be extracted from an image with a smoothing of Sigma = 1.5, High should be chosen larger than 14. If, on the other hand, 10 pixel wide lines with an amplitude larger than 100 and a Sigma = 3 are to be detected, High should be chosen larger than 3.5. For the choice of Low values between 0.25 High and 0.5 High are appropriate.

The parameters Low and High can be calculated from the respective gray value contrast of the lines to be extracted ($ContrastLow$ and $ContrastHigh$) and from the chosen value for Sigma with the following formula:

$$\begin{pmatrix} \texttt{Low} \\ \texttt{High} \end{pmatrix} = \left| -2 \cdot \begin{pmatrix} ContrastLow \\ ContrastHigh \end{pmatrix} \cdot \frac{w}{\sqrt{2\pi} \cdot \texttt{Sigma}^3} \cdot e^{-\frac{w^2}{2 \cdot \texttt{Sigma}^2}} \right|$$

where w is the width (half the diameter) of the lines in the image. $ContrastLow$ and $ContrastHigh$ determine the gray value range the lines are expected to differ from the background. Suitable values for Low and High can be calculated using the procedure calculate_lines_gauss_parameters.

The extracted lines are returned in a topologically sound data structure in Lines. This means that lines are correctly split at junction points.

lines_gauss defines the following attributes for each line point if ExtractWidth was set to *'false'*:

'angle': The angle of the direction perpendicular to the line
'response': The magnitude of the second derivative

If `ExtractWidth` was set to *'true'*, the following attributes are defined in addition to *'angle'* and *'response'*:

'width_left': The line width to the left of the line
'width_right': The line width to the right of the line

If `ExtractWidth` was set to *'true'* and `LineModel` to a value different from *'none'*, the following attributes are defined in addition to *'angle'*, *'response'*, *'width_left'*, and *'width_right'*:

'asymmetry': The asymmetry of the line point
'contrast': The contrast of the line point

Here, the asymmetry is positive if the asymmetric part, i.e., the part with the weaker gradient, is on the right side of the line, while it is negative if the asymmetric part is on the left side of the line.

The contrast results from the difference between the gray value of the line and the gray value of the background. The contrast is positive if bright lines are extracted, while it is negative if dark lines are extracted. The returned contrast may be larger than the maximum gray value the input image type is able to represent, especially if the line model specified by `LineModel` is not present in the image. For example, for byte images, the contrast may be greater than 255.

Use `get_contour_attrib_xld` to obtain attribute values. See the operator reference of `get_contour_attrib_xld` for further information about contour attributes.

`lines_gauss` can be executed on OpenCL devices.

Attention

In general, but in particular if the line width is to be extracted, `Sigma` $\geq w/\sqrt{3}$ should be selected, where w is the width (half the diameter) of the lines in the image. As the lowest allowable value `Sigma` $\geq w/2.5$ must be selected. If, for example, lines with a width of 4 pixels (diameter 8 pixels) are to be extracted, `Sigma` ≥ 2.3 should be selected. Note that the attributes *'width_left'*, *'width_right'*, *'asymmetry'*, and *'contrast'* are set to zero if `Sigma` is set too low.

`lines_gauss` uses a special implementation that is optimized using SSE2 or AVX2 instructions, if the system parameters *'sse2_enable'* or *'avx2_enable'* are set to *'true'* (see `set_system`). These implementations are slightly inaccurate compared to the pure C version due to numerical issues. If you prefer accuracy over performance you can set the respective system parameter to *'false'* (using `set_system`) before you call `lines_gauss`. This way `lines_gauss` does not use SSE2 or AVX2 accelerations. Don't forget to set *'sse2_enable'* or *'avx2_enable'* back to *'true'* afterwards.

When `lines_gauss` is run on OpenCL devices, the same limitations apply as for `derivate_gauss`: `Sigma` must be chosen so that the required filter mask is smaller than 129 pixels. Also note that the results can vary compared to the CPU implementation.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **`Image`** (input_object) ..... singlechannelimage ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
  Input image.
- ▷ **`Lines`** (output_object) ..... xld_cont-array ⇝ *object*
  Extracted lines.
- ▷ **`Sigma`** (input_control) ..... number ⇝ *real* / integer
  Amount of Gaussian smoothing to be applied.
  Default: 1.5
  Suggested values: `Sigma` ∈ {1, 1.2, 1.5, 1.8, 2, 2.5, 3, 4, 5}
  Value range: 0 ≤ Sigma
  Recommended increment: 0.1
- ▷ **`Low`** (input_control) ..... number ⇝ *real* / integer
  Lower threshold for the hysteresis threshold operation.
  Default: 3
  Suggested values: `Low` ∈ {0, 0.5, 1, 2, 3, 4, 5, 8, 10}
  Value range: 0 ≤ Low
  Recommended increment: 0.5

▷ **High** (input_control) . . . . . number ⇝ *real* / integer
Upper threshold for the hysteresis threshold operation.
Default: 8
Suggested values: High ∈ {0, 0.5, 1, 2, 3, 4, 5, 8, 10, 12, 15, 18, 20, 25}
Value range: 0 ≤ High
Recommended increment: 0.5
Restriction: High >= Low

▷ **LightDark** (input_control) . . . . . string ⇝ *string*
Extract bright or dark lines.
Default: 'light'
List of values: LightDark ∈ {'dark', 'light'}

▷ **ExtractWidth** (input_control) . . . . . string ⇝ *string*
Should the line width be extracted?
Default: 'true'
List of values: ExtractWidth ∈ {'true', 'false'}

▷ **LineModel** (input_control) . . . . . string ⇝ *string*
Line model used to correct the line position and width.
Default: 'bar-shaped'
List of values: LineModel ∈ {'none', 'bar-shaped', 'parabolic', 'gaussian'}

▷ **CompleteJunctions** (input_control) . . . . . string ⇝ *string*
Should junctions be added where they cannot be extracted?
Default: 'true'
List of values: CompleteJunctions ∈ {'true', 'false'}

Example

```
* Detection of lines in an aerial image
read_image(Image,'mreut4_3')
lines_gauss(Image,Lines,1.5,3,8,'light','true','bar-shaped','true')
dev_display(Lines)
```

Complexity

Let A be the number of pixels in the domain of Image. Then the runtime complexity is $O(A * \text{Sigma})$.

The amount of temporary memory required is dependent on the height H of the domain of Image and the width W of Image. Let $S = W * H$, then lines_gauss requires at least $55 * S$ bytes of temporary memory during execution.

Result

lines_gauss returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

gen_polygons_xld

Alternatives

lines_facet

See also

bandpass_image, dyn_threshold, topographic_sketch

References

C. Steger: "Extracting Curvilinear Structures: A Differential Geometric Approach". In B. Buxton, R. Cipolla, eds.,

"Fourth European Conference on Computer Vision", Lecture Notes in Computer Science, Volume 1064, Springer Verlag, pp. 630-641, 1996.

C. Steger: "Extraction of Curved Lines from Images". In "13th International Conference on Pattern Recognition", Volume II, pp. 251-255, 1996.

C. Steger: "An Unbiased Detector of Curvilinear Structures". IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20, no. 2, pp. 113-125, 1998.

C. Steger: "Unbiased extraction of lines with parabolic and Gaussian profiles". Computer Vision and Image Understanding, vol. 117, no. 2, pp. 97-112, 2013.

Module

2D Metrology

12.10 Match

exhaustive_match (Image, RegionOfInterest, ImageTemplate : ImageMatch : Mode :)

Matching of a template and an image.

The operator exhaustive_match matches ImageTemplate and Image within the region of interest RegionOfInterest. Hereby the ImageTemplate will be moved over all points of Image which lie within the RegionOfInterest. With regard to the parameter Mode, a matching criterion will be calculated. The result values will be stored in ImageMatch.

The following matching criteria (Mode) are available:

'norm_correlation'

$$\texttt{ImageMatch}[i][j] = 255 \cdot \frac{\sum_{u,v} (\texttt{Image}[i-u][j-v] \cdot \texttt{ImageTemplate}[l-u][c-v])}{\sqrt{\sum_{u,v} (\texttt{Image}[i-u][j-v]^2) \cdot \sum_{u,v} (\texttt{ImageTemplate}[l-u][c-v]^2)}}$$

whereby $X[i][j]$ indicates the gray value in the i-th column and j-th row of the image X. (l, c) is the center of the region of ImageTemplate. u and v are chosen so that all points of the template will be reached, i, j run across the RegionOfInterest. At the image frame only those parts of ImageTemplate will be considered which lie inside the image (i.e. u and v will be restricted correspondingly). Higher values returned in ImageMatch, represent better matches.

Range of values: [*0 ... 255* (best fit)].

'dfd' Calculating the average "displaced frame difference":

$$\texttt{ImageMatch}[i][j] = \frac{\sum_{u,v} |\texttt{Image}[i-u][j-v] - \texttt{ImageTemplate}[l-u][c-v]|}{AREA(ImageTemplate)}$$

The terms are the same as in *'norm_correlation'*. $AREA(X)$ means the area of the region X. Lower values returned in ImageMatch, represent better matches.

Range of values: [*0* (best fit) *... 255*].

To calculate the normalized correlation as well as the "displaced frame difference" is (with regard to the area of ImageTemplate) very time consuming. Therefore it is important to restrict the input region (RegionOfInterest if possible, i.e. to apply the filter only in a very confined "region of interest".

As far as quality is concerned, both modes return comparable results, whereby the mode *'dfd'* is faster by a factor of about 3.5.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
Input image.

▷ **RegionOfInterest** (input_object) ..... region ⇝ *object*
Area to be searched in the input image.

▷ **ImageTemplate** (input_object) . . . . . singlechannelimage ⇝ *object :* byte
This area will be "matched" by Image within the RegionOfInterest.

▷ **ImageMatch** (output_object) . . . . . image(-array) ⇝ *object :* byte
Result image: values of the matching criterion.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Desired matching criterion.
Default: 'dfd'
List of values: Mode ∈ {'norm_correlation', 'dfd'}

Example

```
read_image(Image,'monkey')
dev_display(Image)
draw_rectangle2(WindowHandle,Row,Column,Phi,Length1,Length2)
gen_rectangle2(Rectangle,Row,Column,Phi,Length1,Length2)
reduce_domain(Image,Rectangle,Template)
exhaustive_match(Image,Image,Template,ImageMatch,'dfd')
invert_image(ImageMatch,ImageInvert)
local_max(Image,Maxima)
union1(Maxima,AllMaxima)
add_channels(AllMaxima,ImageInvert,FitMaxima)
threshold(FitMaxima,BestFit,230.0,255.0)
dev_display(BestFit)
```

Result

If the parameter values are correct, the operator exhaustive_match returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

draw_region, draw_rectangle1

Possible Successors

local_max, threshold

Alternatives

exhaustive_match_mg

Module

Foundation

exhaustive_match_mg (Image, ImageTemplate : ImageMatch : Mode, Level, Threshold :)

Matching a template and an image in a resolution pyramid.

The operator exhaustive_match_mg is an additional option for the operator exhaustive_match performing a matching of the image Image and the template ImageTemplate. Hereby ImageTemplate will be moved over all points of the region of Image, a matching criterion will be calculated with regard to the parameter Mode and the result values will be stored in ImageMatch.

Of images having been filtered this way, normally only those areas with good matching results are of interest. The size of the area to be searched, i.e. the region of the input image Image, determines decisively the runtime of

the matching filter. Therefore it is recommendable to use at first exhaustive_match_mg with reduced image resolution in order to determine a "region of interest" in which good matching results can be expected; then in this restricted area only the real matching (see also exhaustive_match) will be executed with normal resolution.

Hereby the Gauss-pyramids of Image and ImageTemplate will be composed (in particular the corresponding regions will be transformed as well). Then on each level of the resolution pyramids - starting with the startlevel Level - the matching inside the current "region of interest" will be executed. Whereby the "region of interest" on the startlevel is equivalent to the region of the input image Image. After the filtering, a new "region of interest" is determined with the help of a threshold operation and will be transformed on the next resolution level:

threshold(..0,Threshold..), if Mode = 'dfd'
threshold(..Threshold,255..), if Mode = 'norm_correlation'

The final matching in the determined "region of interest" will then be calculated with the highest resolution (Level 0). The output image ImageMatch includes the corresponding filter result and the final "region of interest", which is determined on the result image with the help of a threshold operation.

The operator exhaustive_match_mg therefore is not simply a filter, but can also be considered as a member of the class of region transformations.

Parameters

- ▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
  Input image.
- ▷ **ImageTemplate** (input_object) . . . . . singlechannelimage ⇝ *object :* byte
  The domain of this image will be matched with Image.
- ▷ **ImageMatch** (output_object) . . . . . image(-array) ⇝ *object :* byte
  Result image and result region: values of the matching criterion within the determined "region of interest".
  Number of elements: ImageMatch == Image
- ▷ **Mode** (input_control) . . . . . string ⇝ *string*
  Desired matching criterion.
  Default: 'dfd'
  List of values: Mode ∈ {'norm_correlation', 'dfd'}
- ▷ **Level** (input_control) . . . . . integer ⇝ *integer*
  Startlevel in the resolution pyramid (highest resolution: Level 0).
  Default: 1
  Suggested values: Level ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8}
  Restriction: Level <= ld(width(Image)) && Level <= ld(height(Image)) && Level <= ld(width(ImageTemplate)) && Level <= ld(height(ImageTemplate))
- ▷ **Threshold** (input_control) . . . . . integer ⇝ *integer*
  Threshold to determine the "region of interest".
  Default: 30
  Suggested values: Threshold ∈ {5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120, 125, 130, 135, 140, 145, 150, 155, 160, 165, 170, 175, 180, 185, 190, 195, 200, 205, 210, 215, 220, 225, 230, 235, 240, 245, 250}
  Value range: 0 ≤ Threshold ≤ 255
  Minimum increment: 1
  Recommended increment: 5

Example

```
read_image(&Image,"monkey");
disp_image(Image,WindowHandle);
draw_rectangle2(WindowHandle,&Row,&Column,&Phi,&Length1,&Length2);
gen_rectangle2(&Rectangle,Row,Column,Phi,Length1,Length2);
reduce_domain(Image,Rectangle,&Template);
exhaustive_match_mg(Image,Template,&ImageMatch,'dfd'1,30);
invert_image(ImageMatch,&ImageInvert);
local_max(ImageInvert,&BestFit);
disp_region(BestFit,WindowHandle);
```

Result

If the parameter values are correct, the operator exhaustive_match_mg returns the value 2 (H_MSG_TRUE).

If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

draw_region, draw_rectangle1

Possible Successors

threshold, local_max

Alternatives

exhaustive_match

See also

gen_gauss_pyramid

Module

Foundation

gen_gauss_pyramid (Image : ImagePyramid : Mode, Scale :)

Calculating a Gauss pyramid.

The operator gen_gauss_pyramid calculates a pyramid of scaled down images. The scale by which the next image will be reduced is determined by the parameter Scale. For instance, a value of 0.5 for Scale will shorten the edge length of Image by 50%. This is exactly equivalent to the "normal" pyramid.

The parameter Mode determines the way of averaging. For a more detailed description concerning this parameter see also affine_trans_image. In the case that Scale is equal 0.5 there are the additional modes *'min'* and *'max'* available. In this case the minimum or the maximum of the four neighboring pixels is selected.

Please note that each level will be returned as an individual image, i.e., as one iconic object, with one matrix and its own domain. Single or multiple levels can be selected by using select_obj or copy_obj, respectively.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2 / real
Input image.

▷ **ImagePyramid** (output_object) . . . . . (multichannel-)image-array ⇝ *object :* byte / uint2 / real
Output images.
Number of elements: ImagePyramid > Image

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Kind of filter mask.
Default: 'weighted'
List of values: Mode ∈ {'nearest_neighbor', 'constant', 'weighted', 'min', 'max'}

▷ **Scale** (input_control) . . . . . real ⇝ *real*
Factor for scaling down.
Default: 0.5
Suggested values: Scale ∈ {0.2, 0.3, 0.4, 0.5, 0.6}
Value range: 0.001 ≤ Scale ≤ 0.9
Recommended increment: 0.1

Example

```
gen_gauss_pyramid(Image,Pyramid,"weighted",0.5);
count_obj(Pyramid,&num);
for (i=1; i<=num; i++)
```

```
{
  select_obj(Pyramid,&Single,i);
  disp_image(Single,WindowHandle);
  clear(Single);
}
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on channel level.

Possible Successors

image_to_channels, count_obj, select_obj, copy_obj

Alternatives

zoom_image_size, zoom_image_factor

See also

affine_trans_image

Module

Foundation

monotony (Image : ImageMonotony : :)

Calculating the monotony operation.

The operator monotony calculates the monotony operator. Thereby the points which are strictly smaller than the current gray value will be counted in the 8 neighborhood. This number will be entered into the output imaged.

If there is a strict maximum, the value 8 is returned; in case of a minimum or a plateau, the value 0 will be returned. A ridge or a slope will return the corresponding intermediate values.

The monotony operator is often used to prepare matching operations as it is invariant with regard to lightness modifications.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Input image.

▷ **ImageMonotony** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
Result of the monotony operator.
Number of elements: ImageMonotony == Image

Example

```
/* searching the strict maximums */
gauss_filter(Image,&Gauss,5);
monotony(Gauss,&Monotony);
threshold(Monotony,Maxima,8.0,8.0);
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

binomial_filter, gauss_filter, median_image, mean_image, smooth_image, invert_image

Possible Successors

threshold, exhaustive_match, disp_image

Alternatives

local_max, topographic_sketch, corner_response

Module

Foundation

12.11 Misc

convol_image (Image : ImageResult : FilterMask, Margin :)

Calculate the correlation between an image and an arbitrary filter mask

convol_image calculates the correlation between the input image Image and an arbitrary linear filter mask. The used filter mask, which is given in FilterMask, can be either loaded from a file or passed as a tuple. Several options for the treatment at the image's borders can be chosen (Margin):

gray value Pixels outside of the image border are assumed to be constant (with the indicated gray value).

'continued' Continuation of the gray values at the image border.

'cyclic' Cyclic continuation at the image borders.

'mirrored' Reflection of pixels at the image borders.

At all image positions, the correlation between the image and the filter mask is calculated. If an overflow or underflow occurs, the resulting gray value is clipped. Hence, if filters that result in negative output values are used (e.g., derivative filters), the input image should be of type int2 or real.

The reference pixel of the mask, i.e., the pixel that lies at the current image position for which the correlation is calculated, is determined as follows: First, the region of all mask pixels with a weight other than 0 is computed. Then, the center of this region is computed and rounded. This is the reference point of the mask.

If a file name is given in FilterMask, the filter mask is read from a text file with the following structure:

⟨Mask size⟩
⟨Inverse weight of the mask⟩
⟨Matrix⟩

The first line contains the size of the filter mask, given as two numbers separated by white space (e.g., 3 3 for 3×3). Here, the first number defines the height (rows) of the filter mask, while the second number defines its width (columns). The next line contains the inverse weight of the mask, i.e., the number by which the correlation at a particular image position is divided. The remaining lines contain the filter mask as integer or floating point numbers (separated by white space), one line of the mask per line in the file. The default HALCON file extension for the filter mask is 'fil'. It is not necessary to pass this extension to the operator. If the filter mask is to be computed from a tuple, the tuple given in FilterMask must also satisfy the structure described above. However, in this case the line feed is omitted.

For example, lets assume we want to use the following filter mask:

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

If the filter mask should be generated from a file, then the file should look like this:

3 3
16
1 2 1

2 4 2
1 2 1

In contrast, if the filter mask should be generated from a tuple, then the following tuple must be passed in `FilterMask`:

[3,3,16,1,2,1,2,4,2,1,2,1]

For convenience, it is possible to pass two vectors instead of a matrix in `FilterMask`:

[MaskHeight,MaskWidth,Weight,V1,V2]

The outer product of those two vectors forms the filter matrix:

$Mask := V1 * V1^T$

E.g., the matrix above can also be passed as:

[3,3,16,1,2,1,1,2,1]

If `FilterMask` is separable (which is detected automatically), `convol_image` uses a special implementation that is significantly faster than the filtering with non separable masks.

If *'sse2_enable'* is set to *'true'* (and the SIMD instruction set is available), the internal calculations for `byte` and `real` images are performed using SIMD technology. If *'sse41_enable'* is set to *'true'* (and the SIMD instruction set is available), the internal calculations for `int2` and `uint2` images are performed using SIMD technology. If *'avx_enable'* is set to *'true'* (and the SIMD instruction set is available), the internal calculations for `real` images are performed using AVX SIMD technology.

Attention

Note that `convol_image` does not compute a convolution of the image with the given filter mask but a correlation, i.e., it uses the given filter mask directly, not a mirrored version of the filter mask.

When using a 3x3 or 5x5 rectangular filter mask and the border treatment *'mirrored'*, `convol_image` can be executed on OpenCL devices.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
  Images for which the correlation will be calculated.
- ▷ **`ImageResult`** (output_object) ..... multichannel-image(-array) ⇝ *object :* byte / int2 / uint2 / real
  Result of the correlation.
- ▷ **`FilterMask`** (input_control) ..... filename.read(-array) ⇝ *string* / integer / real
  Filter mask as file name or tuple.
  Default: 'sobel'
  Suggested values: `FilterMask` ∈ {'sobel', 'laplace4', 'lowpas_3_3'}
  File extension: `.fil`
- ▷ **`Margin`** (input_control) ..... string ⇝ *string* / integer / real
  Border treatment.
  Default: 'mirrored'
  Suggested values: `Margin` ∈ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Module

Foundation

deviation_n (Image : ImageDeviation : :)

Calculate standard deviation over several channels.

deviation_n generates the pixel-by-pixel standard deviation of a multichannel gray value image. For each coordinate point p the expected value is defined as the arithmetic mean:

$$E = \frac{\sum g_c(p)}{n}$$

where $g\_c(p)$ denominates the gray value at p on channel c. n is the number of channels in the image.

The standard deviation itself is then calculated as:

$$s = 2 \cdot \sqrt{\frac{1}{n-1} \cdot \sum (g_c(p) - E)^2}$$

Note that this formula produces the *sample* standard deviation. The factor 2 is used to make better use of the range of values of the output image format. The output image has one channel.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Multichannel gray image.

▷ **ImageDeviation** (output_object) . . . . . image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Result of calculation.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on domain level.
- Automatically parallelized on tuple level.

Possible Predecessors

compose2, compose3, compose4, compose5, add_channels

See also

mean_n

Module

Foundation

expand_domain_gray (
InputImage : ExpandedImage : ExpansionRange :)

Expand the domain of an image and set the gray values in the expanded domain.

expand_domain_gray expands the border gray values of the domain outwards. The width of the expansion is set by the parameter ExpansionRange, which describes the expansion radius in pixels. All filters in HALCON which are applied to the image domain also include the gray values of a certain number of pixels, depending on the filter size, outside of the image domain in their calculation. This may lead to undesirable side effects especially in the border region of the domain. For example, if the foreground (domain) and the background of the image differ strongly in brightness, the result of a filter operation may lead to undesired darkening or brightening at the border of the domain. In order to avoid this drawback, the domain is artificially expanded by expand_domain_gray in a preliminary stage, copying the gray values of the border pixels to the outside of the domain. In addition, the domain itself is also expanded to reflect the newly set pixels. Therefore, in many cases it is reasonable to reduce the domain again (reduce_domain or change_domain) after using expand_domain_gray and call the filter operation afterwards. ExpansionRange should be set to a value high enough to make sure the whole region

including its dilatation through the filter mask has initialized values. Usually this leads to ExpansionRange values of half the filter length or greater. In combination with the filter mean_image on images of type real we recommend to determine the value for ExpansionRange as shown in the example below. For speed reasons this filter uses the values from pixels within the smallest rectangle around the dilated region.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **InputImage** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
Input image with domain to be expanded.

▷ **ExpandedImage** (output_object) .....image(-array) ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
Output image with new gray values in the expanded domain.

▷ **ExpansionRange** (input_control) ..... integer ⇝ *integer*
Radius of the gray value expansion, measured in pixels.
Default: 2
Suggested values: ExpansionRange ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16}
Restriction: ExpansionRange >= 1

Example

```
read_image(Fabrik, 'fabrik')
gen_rectangle2(Rectangle_Label,243,320,-1.55,62,28)
reduce_domain(Fabrik, Rectangle_Label, Fabrik_Label)
* Character extraction without gray value expansion:
MaskSize := 31
mean_image(Fabrik_Label,Label_Mean_normal,MaskSize, MaskSize)
dyn_threshold(Fabrik_Label,Label_Mean_normal,Characters_normal,10,'dark')
dev_display(Fabrik)
dev_display(Characters_normal)
* The characters in the border region are not extracted !
stop()
* Character extraction with gray value expansion:
get_domain (Fabrik_Label, Domain)
expand_domain_gray (Fabrik_Label, Expanded_Fabrik_Label, MaskSize)
get_domain (Expanded_Fabrik_Label, Expanded_Domain)
shape_trans (Expanded_Domain, RegionRect, 'rectangle1')
difference (RegionRect, Expanded_Domain, Region_Difference)
overpaint_region (Expanded_Fabrik_Label, Region_Difference, 0, 'fill')
reduce_domain (Expanded_Fabrik_Label, Domain, Reduced_Expanded_Fabrik_Label)
mean_image (Reduced_Expanded_Fabrik_Label, Label_Mean_Expanded,\
MaskSize, MaskSize)
dyn_threshold(Fabrik_Label,Label_Mean_Expanded,\
Characters_expanded,10,'dark')
dev_display(Fabrik)
dev_display(Characters_expanded)
```

Complexity

Let L the perimeter of the domain. Then the runtime complexity is approximately $O(L) * \text{ExpansionRange}$.

Result

expand_domain_gray returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.

Possible Predecessors

reduce_domain

Possible Successors

reduce_domain, mean_image, dyn_threshold

See also

reduce_domain, mean_image

Module

Foundation

gray_inside (Image : ImageDist : :)

Calculate the lowest possible gray value on an arbitrary path to the image border for each point in the image.

gray_inside determines the "cheapest" path to the image border for each point in the image, i.e., the path on which the lowest gray values have to be overcome. The resulting image contains the difference of the gray value of the particular point and the maximum gray value on the path. Bright areas in the result image therefore signify that these areas (which are typically dark in the original image) are surrounded by bright areas. Dark areas in the result image signify that there are only small gray value differences between them and the image border (which doesn't mean that they are surrounded by dark areas; a small "gap" of dark values suffices). The value 0 (black) in the result image signifies that only darker or equally bright pixels exist on the path to the image border.

The operator is implemented by first segmenting into basins and watersheds the image using the watersheds operator. If the image is regarded as a gray value mountain range, basins are the places where water accumulates and the mountain ridges are the watersheds. Then, the watersheds are distributed to adjacent basins, thus leaving only basins. The border of the domain (region) of the original image is now searched for the lowest gray value, and the region in which it resides is given its result values. If the lowest gray value resides on the image border, all result values can be calculated immediately using the gray value differences to the darkest point. If the smallest found gray value lies in the interior of a basin, the lowest possible gray value has to be determined from the already processed adjacent basins in order to compute the new values. An 8-neighborhood is used to determine adjacency. The found region is subtracted from the regions yet to process, and the whole process is repeated. Thus, the image is "stripped" form the outside.

Analogously to watersheds, it is advisable to apply a smoothing operation before calling watersheds, e.g., binomial_filter or gauss_filter, in order to reduce the amount of regions that result from the watershed algorithm, and thus to speed up the processing time.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Image being processed.

▷ **ImageDist** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* int2
Result image.

Example

```
read_image(Image,'fabrik')
gauss_filter (Image,GaussImage,11)
gray_inside(GaussImage,ImageOut)
dev_display(ImageOut)
```

Result

gray_inside always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image, mean_image, median_image

Possible Successors

select_shape, area_center, count_obj

See also

watersheds

Module

Foundation

gray_skeleton (Image : GraySkeleton : :)

Thinning of gray value images.

gray_skeleton applies a gray value thinning operation to the input image Image. Figuratively, the gray value "mountain range" is reduced to its ridge lines by setting the gray value of "hillsides" to the gray value at the corresponding valley bottom. The resulting ridge lines are at most two pixels wide. This operator is especially useful for thinning edge images, and is thus an alternative to nonmax_suppression_amp. In contrast to nonmax_suppression_amp, gray_skeleton preserves contours, but is much slower. In contrast to skeleton, this operator changes the gray values of an image while leaving its region unchanged.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image to be thinned.

▷ **GraySkeleton** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2
Thinned image.

Example

```
* Seeking leafs of a beech tree in an aerial picture:
read_image(Image,'forest')
gray_skeleton(Image,Skelett)
mean_image(Skelett,MeanSkelett,7,7)
dyn_threshold(Skelett,MeanSkelett,Leafs,3,'light')
```

Result

gray_skeleton returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Successors

mean_image

Alternatives

nonmax_suppression_amp, nonmax_suppression_dir, local_max

See also

skeleton, gray_dilation_rect

Module

Foundation

lut_trans (Image : ImageResult : Lut :)

Transform an image with a gray-value look-up-table

lut_trans transforms an image Image by using a gray value look-up-table Lut. This table acts as a transformation function. In the case of byte-images, Lut has to be a tuple of length 256. In the case of int2-images, Lut has to be a tuple of length 256 <= length <= 65536. If the length of the Lut is <= 32768, the transformation is applied to the positive gray values only, i.e., the first element of the Lut specifies the new gray value for the gray value 0. If the Lut is longer than 32768, exactly 65536 must be passed. In this case, the positive and negative gray values are transformed. In this case, the first element indicates the new gray value for the gray value -32768 of the input image, while the last element of the tuple indicates the new gray value for the gray value 32767. In all cases, the gray values of values outside the range of Lut are set to 0. In the case of uint2-images, Lut has to be a tuple of length 256 <= length <= 65536. Gray values outside the range of Lut are set to 0.

Attention

lut_trans can be executed on OpenCL devices.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Image whose gray values are to be transformed.
- ▷ **ImageResult** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Transformed image.
- ▷ **Lut** (input_control) ..... integer-array ⇝ *integer*
  Table containing the transformation.

Example

```
* Apply a gamma correction to an image.
Gamma := 2.2
GammaLUT := []
for G := 0 to 255 by 1
    GammaLUT := [GammaLUT,round(255.0*pow(G/255.0,1.0/Gamma))]
endfor
read_image (Image, 'mreut')
lut_trans (Image, ImageGamma, GammaLUT)
```

Result

The operator lut_trans returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

- Automatically parallelized on domain level.

Module

Foundation

symmetry (Image : ImageSymmetry : MaskSize, Direction, Exponent :)

Symmetry of gray values along a row.

symmetry calculates the symmetry along a line. For each pixel the gray values of both sides of the line are compared: The absolute value of the differences of gray values with same distance to the pixel is computed. Each of these differences is weighted by the exponent (after division by 255) and then summed up.

$$sym := 255 - \frac{255}{\texttt{MaskSize}} \sum_{i=1}^{\texttt{MaskSize}} \left(\frac{|g(i) - g(-i)|}{255} \right)^{\texttt{Exponent}}$$

Pixels with a high symmetry have large gray values.

Attention

Currently only horizontal search lines are implemented. Note that the parameter Direction exists for future extensions and can currently only have the value 0.0.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Input image.

▷ **ImageSymmetry** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Symmetry image.

▷ **MaskSize** (input_control) ..... number ⇝ *integer*
Extension of search area.
Default: 40
Suggested values: MaskSize ∈ {3, 5, 7, 10, 15, 20, 25, 30, 40, 50, 60, 70, 80, 100, 120, 140, 180}
Value range: 3 ≤ MaskSize ≤ 1000
Minimum increment: 1
Recommended increment: 2

▷ **Direction** (input_control) ..... number ⇝ *real*
Angle of test direction.
Default: 0.0
Suggested values: Direction ∈ {0.0}
Value range: 0.0 ≤ Direction ≤ 0.0

▷ **Exponent** (input_control) ..... number ⇝ *real*
Exponent for weighting.
Default: 0.5
Suggested values: Exponent ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: 0 < Exponent && Exponent <= 100

Example

```
read_image(Image,'monkey')
symmetry(Image,ImageSymmetry,70,0.0,0.5)
threshold(ImageSymmetry,SymmPoints,170,255)
```

Result

If the parameter values are correct the operator symmetry returns the value 2 (H_MSG_TRUE) The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

`threshold`

Module

Foundation

topographic_sketch (Image : Sketch : :)

Compute the topographic primal sketch of an image.

`topographic_sketch` computes the topographic primal sketch of the input image `Image`. This is done by approximating the image locally by a bicubic polynomial ("facet model"). It serves to calculate the first and second partial derivatives of the image, and thus to classify the image into 11 classes. These classes are coded in the output image `Sketch` as numbers from 1 to 11. The classes are as follows:

Peak	1
Pit	2
Ridge	3
Ravine	4
Saddle	5
Flat	6
Hillside Slope	7
Hillside Convex	8
Hillside Concave	9
Hillside Saddle	10
Hillside Inflection	11

In order to obtain the separate classes as regions, a threshold operation has to be applied to the result image with the appropriate thresholds.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Image for which the topographic primal sketch is to be computed.

▷ **Sketch** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Label image containing the 11 classes.

Example

```
* To extract the Ridges from a Image
read_image(Image,'sinus')
topographic_sketch(Image,Sketch)
threshold(Sketch,Ridges,3,3)
```

Complexity

Let n be the number of pixels in the image. Then $O(n)$ operations are performed.

Result

`topographic_sketch` returns 2 (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Successors

`threshold`

References

R. Haralick, L. Shapiro: "Computer and Robot Vision, Volume I"; Reading, Massachusetts, Addison-Wesley; 1992; Kapitel 8.13.

Module

Foundation

12.12 Noise

add_noise_distribution (Image : ImageNoise : Distribution :)

Add noise to an image.

`add_noise_distribution` adds noise distributed according to `Distribution` to the `Image`. The resulting gray values are clipped to the range of the corresponding pixel type.

The `Distribution` is stored in a tuple of length 513. The individual values of this tuple define the frequency of noise with a specific amplitude defined by the position within the tuple. The central value, i.e., the value at the position 256 in the tuple defines the frequency of pixels that are not changed. The value at the position 255 defines the frequency of pixels for which the grayvalue is decreased by 1. The value at the position 254 defines the respective frequency for a grayvalue decrease of 2, and so on. Analogously, the value at position 257 defines the frequency of pixels for which the grayvalue is increased by 1.

The `Distribution` represents salt and pepper noise if at most one value at a position smaller than 256 is not equal to zero and at most one value at a position larger than 256 is not equal to zero. In case of salt and pepper noise, the added noisy pixels are set to the minimum (pepper) and maximum (salt) values that can be represented by `ImageNoise` if the amount of pepper is indicated by the value at position 0 and the amount of salt is indicated by the value at position 512 in the tuple.

The random noise is generated using the C function "drand48()". See the parameter *'seed_rand'* of `set_system` for information on the used random seed.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2
Input image.

▷ **ImageNoise** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2
Noisy image.
Number of elements: ImageNoise == Image

▷ **Distribution** (input_control) . . . . . distribution.values-array ⇝ *real*
Noise distribution.
Number of elements: 513

Example

```
read_image(Image,'mreut')
dev_display (Image)
sp_distribution(30,30,Dist)
add_noise_distribution(Image,ImageNoise,Dist)
dev_display (ImageNoise)
```

Result

`add_noise_distribution` returns 2 (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Predecessors

`gauss_distribution`, `sp_distribution`, `noise_distribution_mean`

Alternatives

`add_noise_white`

See also

`sp_distribution`, `gauss_distribution`, `noise_distribution_mean`, `add_noise_white`

Module

Foundation

add_noise_white (Image : ImageNoise : Amp :)

Add noise to an image.

`add_noise_white` adds noise to the image `Image`. The noise is white noise, equally distributed in the interval [-`Amp`,`Amp`]. The resulting gray values are clipped to the range of the corresponding pixel type.

The random noise is generated using the C function “drand48”. See the parameter *’seed_rand’* of `set_system` for information on the used random seed.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Input image.

▷ **ImageNoise** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Noisy image.
Number of elements: ImageNoise == Image

▷ **Amp** (input_control) . . . . . real ⇝ *real*
Maximum noise amplitude.
Default: 60.0
Suggested values: Amp ∈ {1.0, 2.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Value range: 0.0 ≤ Amp
Minimum increment: 0.001
Recommended increment: 10.0

Example

```
read_image(Image,'fabrik')
dev_display (Image)
add_noise_white(Image,ImageNoise,90)
dev_display (ImageNoise)
```

Result

add_noise_white returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Alternatives

add_noise_distribution

See also

add_noise_distribution, noise_distribution_mean, gauss_distribution, sp_distribution

Module

Foundation

gauss_distribution (: : Sigma : Distribution)

Generate a Gaussian noise distribution.

gauss_distribution generates a Gaussian noise distribution. The parameter Sigma determines the noise's standard deviation. Usually, the result Distribution is used as input for the operator add_noise_distribution.

Parameters

▷ **Sigma** (input_control) ..... real ⇝ *real*
Standard deviation of the Gaussian noise distribution.
Default: 2.0
Suggested values: Sigma ∈ {1.5, 2.0, 3.0, 5.0, 10.0}
Value range: 0.0 ≤ Sigma ≤ 100.0
Minimum increment: 0.1
Recommended increment: 1.0

▷ **Distribution** (output_control) ..... distribution.values-array ⇝ *real*
Resulting Gaussian noise distribution.
Number of elements: 513

Example

```
read_image(Image,'fabrik')
dev_display (Image)
gauss_distribution(30,Dist)
add_noise_distribution(Image,ImageNoise,Dist)
dev_display (ImageNoise)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Successors

add_noise_distribution

Alternatives

sp_distribution, noise_distribution_mean

See also

sp_distribution, add_noise_white, noise_distribution_mean

Module

Foundation

noise_distribution_mean (ConstRegion, Image : : FilterSize : Distribution)

Determine the noise distribution of an image.

noise_distribution_mean calculates the noise distribution in a region of the image Image. The parameter ConstRegion determines a region of the image with approximately constant gray values. Ideally, the changes in gray values should only be caused by noise in this region. From this region the noise distribution is determined by using the mean_image operator to smooth the image, and to use the gray value differences in this area as an estimate for the noise distribution, which is returned in Distribution.

Attention

It is important to ensure that the region ConstRegion is not too close to a large gradient in the image, because the gradient values are then used for calculating the mean. This means the distance of ConstRegion must be at least as large as the filter size FilterSize used for calculating the mean.

Parameters

▷ **ConstRegion** (input_object) . . . . . region(-array) ⇝ *object*
Region from which the noise distribution is to be estimated.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte
Corresponding image.

▷ **FilterSize** (input_control) . . . . . integer ⇝ *integer*
Size of the mean filter.
Default: 21
Suggested values: FilterSize ∈ {5, 11, 15, 21, 31, 51, 101}
Value range: 3 ≤ FilterSize ≤ 501 (lin)
Minimum increment: 2
Recommended increment: 2

▷ **Distribution** (output_control) . . . . . distribution.values-array ⇝ *real*
Noise distribution of all input regions.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

draw_region, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, threshold, erosion_circle, binomial_filter, gauss_filter, smooth_image, sub_image

Possible Successors

add_noise_distribution

See also

mean_image, gauss_distribution

Module

Foundation

sp_distribution (: : PercentSalt, PercentPepper : Distribution)

Generate a salt-and-pepper noise distribution.

sp_distribution generates a noise distribution with the values 0 and 255. The parameters PercentSalt and PercentPepper determine the percentage of white and black noise pixels, respectively. The sum of these parameters must be smaller than 100. Usually, the result Distribution is used as input for the operator add_noise_distribution.

Parameters

▷ **PercentSalt** (input_control) . . . . . number ⇝ *real* / integer
Percentage of salt (white noise pixels).
Default: 5.0
Suggested values: PercentSalt ∈ {1.0, 2.0, 5.0, 7.0, 10.0, 15.0, 20.0, 30.0}
Value range: 0.0 ≤ PercentSalt ≤ 100.0
Minimum increment: 0.1
Recommended increment: 1.0
Restriction: PercentSalt + PercentPepper <= 100

▷ **PercentPepper** (input_control) . . . . . number ⇝ *real* / integer
Percentage of pepper (black noise pixels).
Default: 5.0
Suggested values: PercentPepper ∈ {1.0, 2.0, 5.0, 7.0, 10.0, 15.0, 20.0, 30.0}
Value range: 0.0 ≤ PercentPepper ≤ 100.0
Minimum increment: 0.1
Recommended increment: 1.0
Restriction: PercentSalt + PercentPepper <= 100

▷ **Distribution** (output_control) . . . . . distribution.values-array ⇝ *real*
Resulting noise distribution.
Number of elements: 513

Example

```
read_image(Image,'fabrik')
dev_display (Image)
sp_distribution(30,30,Dist)
add_noise_distribution(Image,ImageNoise,Dist)
dev_display (ImageNoise)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

add_noise_distribution

Alternatives

gauss_distribution, noise_distribution_mean

See also

gauss_distribution, noise_distribution_mean, add_noise_white

Module

Foundation

12.13 Optical Flow

derivate_vector_field (VectorField : Result : Sigma, Component :)

Convolve a vector field with derivatives of the Gaussian.

derivate_vector_field convolves the components of a vector field with the derivatives of a Gaussian and calculates various features derived therefrom. derivate_vector_field only accepts vector fields of the semantic type vector_field_relative. The VectorField $F(r,c) = (u(r,c), v(r,c))$ is defined as in optical_flow_mg. Sigma is the parameter of the Gaussian (i.e., the amount of smoothing). If a single value is passed in Sigma, the amount of smoothing in the column and row direction is identical. If two values are passed in Sigma, the first value specifies the amount of smoothing in the column direction, while the second value specifies the amount of smoothing in the row direction. The possible values for Component are:

'curl': The curl of the vector field. One application of using *'curl'* is to analyse optical flow fields. Metaphorically speaking, the curl is how much a small boat would rotate if the vector field was a fluid.

$$curl = \frac{\partial u(r,c)}{\partial c} - \frac{\partial v(r,c)}{\partial r}$$

'divergence': The divergence of the vector field. One application of using *'divergence'* is to analyze optical flow fields. Metaphorically speaking, the divergence is where the source and sink would be if the vector field was a fluid.

$$div = \frac{\partial u(r,c)}{\partial r} + \frac{\partial v(r,c)}{\partial c}$$

When used in context of photometric stereo, the operator derivate_vector_field offers two more parameters, which are especially designed to process the gradient field that is returned by photometric_stereo. In this case, we interpret the input vector field as gradient of the underlying surface.

In the following formulas, the input vector field is therefore noted as $G(r,c) = \nabla f = (\frac{\partial f(r,c)}{\partial r}, \frac{\partial f(r,c)}{\partial c})$

where the first and second component of the input is the gradient field of the surface $f(r,c)$. In the formulas below f_rc denotes the first derivative in column direction of the first component of the gradient field.

'mean_curvature': Mean curvature H of the underlying surface when the input vector field VectorField is interpreted as gradient field. One application of using *'mean_curvature'* is to process the vector field that is returned by photometric_stereo. After filtering the vector field, even tiny scratches or bumps can be segmented.

$$\begin{aligned}
A &= (1 + \frac{\partial f(r,c)^2}{\partial r}) \frac{\partial^2 f(r,c)}{\partial c^2} \\
B &= \frac{\partial f(r,c)}{\partial r} \frac{\partial f(r,c)}{\partial c} (\frac{\partial^2 f(r,c)}{\partial r \partial c} + \frac{\partial^2 f(r,c)}{\partial c \partial r}) \\
C &= (1 + \frac{\partial f(r,c)^2}{\partial c}) \frac{\partial^2 f(r,c)}{\partial r^2} \\
D &= (1 + \frac{\partial f(r,c)^2}{\partial r} + \frac{\partial f(r,c)^2}{\partial c})^{\frac{3}{2}} \\
H &= \frac{A - B + C}{D}
\end{aligned}$$

'gauss_curvature': Gaussian curvature K of the underlying surface when the input vector field VectorField is interpreted as gradient field. One application of using *'gauss_curvature'* is to process the vector field that is returned by photometric_stereo. After filtering the vector field, even tiny scratches or bumps can be segmented. If the underlying surface of the vector field is developable, the Gaussian curvature is zero.

$$K = \frac{\frac{\partial f(r,c)^2}{\partial r \partial r} * \frac{\partial f(r,c)^2}{\partial c \partial c} - \frac{\partial f(r,c)^2}{\partial r \partial c} * \frac{\partial f(r,c)^2}{\partial c \partial r}}{(1 + \frac{\partial f(r,c)^2}{\partial r} + \frac{\partial f(r,c)^2}{\partial c})^2}$$

Parameters

▷ **VectorField** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* vector_field
Input vector field.

▷ **Result** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
Filtered result images.

▷ **Sigma** (input_control) . . . . . real(-array) ⇝ *real*
Sigma of the Gaussian.
Default: 1.0
Suggested values: Sigma ∈ {0.7, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0}
Value range: 0.01 ≤ Sigma ≤ 50.0

▷ **Component** (input_control) . . . . . string ⇝ *string*
Component to be calculated.
Default: 'mean_curvature'
List of values: Component ∈ {'curl', 'divergence', 'mean_curvature', 'gauss_curvature'}

Result

If the parameters are valid, the operator derivate_vector_field returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on domain level.
- Automatically parallelized on tuple level.

Possible Predecessors

optical_flow_mg, photometric_stereo

Possible Successors

threshold

Module

Foundation

optical_flow_mg (ImageT1, ImageT2 : VectorField : Algorithm, SmoothingSigma, IntegrationSigma, FlowSmoothness, GradientConstancy, MGParamName, MGParamValue :)

Compute the optical flow between two images.

optical_flow_mg computes the optical flow between two images. The optical flow represents information about the movement between two consecutive images of a monocular image sequence. The movement in the images can be caused by objects that move in the world or by a movement of the camera (or both) between the acquisition of the two images. The projection of these 3D movements into the 2D image plane is called the optical flow.

The two consecutive images of the image sequence are passed in ImageT1 and ImageT2. The computed optical flow is returned in VectorField. The vectors in the vector field VectorField represent the movement in the image plane between ImageT1 and ImageT2. The point in ImageT2 that corresponds to the point (r, c) in ImageT1 is given by $(r', c') = (r + u(r, c), c + v(r, c))$, where $u(r, c)$ and $v(r, c)$ denote the value of the row and column components of the vector field image VectorField at the point (r, c).

The parameter Algorithm allows the selection of three different algorithms for computing the optical flow. All three algorithms are implemented by using multigrid solvers to ensure an efficient solution of the underlying partial differential equations.

For Algorithm = *'fdrig'*, the method proposed by Brox, Bruhn, Papenberg, and Weickert is used. This approach is **f**low-**d**riven, **r**obust, **i**sotropic, and uses a **g**radient constancy term.

For Algorithm = *'ddraw'*, a robust variant of the method proposed by Nagel and Enkelmann is used. This approach is **d**ata-**d**riven, **r**obust, **a**nisotropic, and uses **w**arping (in contrast to the original approach).

For Algorithm = *'clg'* the **c**ombined **l**ocal-**g**lobal method proposed by Bruhn, Weickert, Feddern, Kohlberger, and Schnörr is used.

In all three algorithms, the input images can first be smoothed by a Gaussian filter with a standard deviation of SmoothingSigma (see derivate_gauss).

All three approaches are variational approaches that compute the optical flow as the minimizer of a suitable energy functional. In general, the energy functionals have the following form:

$$E(\mathbf{w}) = E_{\mathrm{D}}(\mathbf{w}) + \alpha E_{\mathrm{S}}(\mathbf{w}),$$

where $\mathbf{w} = (u, v, 1)$ is the optical flow vector field to be determined (with a time step of 1 in the third coordinate). The image sequence is regarded as a continuous function $f(\mathbf{x})$, where $\mathbf{x} = (r, c, t)$ and (r, c) denotes the position and t the time. Furthermore, $E_{\mathrm{D}}(\mathbf{w})$ denotes the data term, while $E_{\mathrm{S}}(\mathbf{w})$ denotes the smoothness term, and α is a regularization parameter that determines the smoothness of the solution. The regularization parameter α is passed in FlowSmoothness. While the data term encodes assumptions about the constancy of the object features in consecutive images, e.g., the constancy of the gray values or the constancy of the first spatial derivative of the gray values, the smoothness term encodes assumptions about the (piecewise) smoothness of the solution, i.e., the smoothness of the vector field to be determined.

The **FDRIG algorithm** is based on the minimization of an energy functional that contains the following assumptions:

Constancy of the gray values: It is assumed that corresponding pixels in consecutive images of an image sequence have the same gray value, i.e., that $f(r + u, c + v, t + 1) = f(r, c, t)$. This can be written more compactly as $f(\mathbf{x} + \mathbf{w}) = f(\mathbf{x})$ using vector notation.

Constancy of the spatial gray value derivatives: It is assumed that corresponding pixels in consecutive images of an image sequence additionally have have the same spatial gray value derivatives, i.e, that $\nabla_2 f(x+u, y+v, t+1) = \nabla_2 f(x, y, t)$ also holds, where $\nabla_2 f = (\partial_x f, \partial_y f)$. This can be written more compactly as $\nabla_2 f(\mathbf{x}+\mathbf{w}) = \nabla_2 f(\mathbf{x})$. In contrast to the gray value constancy, the gradient constancy has the advantage that it is invariant to additive global illumination changes.

Large displacements: It is assumed that large displacements, i.e., displacements larger than one pixel, occur. Under this assumption, it makes sense to consciously abstain from using the linearization of the constancy assumptions in the model that is typically proposed in the literature.

Statistical robustness in the data term: To reduce the influence of outliers, i.e., points that violate the constancy assumptions, they are penalized in a statistically robust manner, i.e., the customary non-robust square penalization $\Psi_{\mathrm{D}}(s^2) = s^2$ is replaced by a linear penalization via $\Psi_{\mathrm{S}}(s^2) = \sqrt{s^2 + \epsilon^2}$, where $\epsilon = 0.001$ is a fixed regularization constant.

Preservation of discontinuities in the flow field I: The solution is assumed to be piecewise smooth. While the actual smoothness is achieved by penalizing the first derivatives of the flow $|\nabla_2 u|^2 + |\nabla_2 v|^2$, the use of a statistically robust (linear) penalty function $\Psi_{\mathrm{S}}(s^2) = \sqrt{s^2 + \epsilon^2}$ with $\epsilon = 0.001$ provides the desired preservation of edges in the movement in the flow field to be determined. This type of smoothness term is called flow-driven and isotropic.

Taking into account all of the above assumptions, the energy functional of the FDRIG algorithm can be written as

$$\begin{aligned} E_{\mathrm{FDRIG}}(\mathbf{w}) \;=\; & \int \Psi_{\mathrm{S}}\Big(\underbrace{|f(\mathbf{x}+\mathbf{w}) - f(\mathbf{x})|^2}_{\text{gray value constancy}} + \gamma \underbrace{|\nabla_2 f(\mathbf{x}+\mathbf{w}) - \nabla_2 f(\mathbf{x})|^2}_{\text{gradient constancy}} \Big) dr dc \\ & + \alpha \int \Psi_{\mathrm{S}} \Big(\underbrace{|\nabla_2 u(\mathbf{x})|^2 + |\nabla_2 v(\mathbf{x})|^2}_{\text{smoothness assumption}} \Big) dr dc \end{aligned}$$

Here, α is the regularization parameter passed in FlowSmoothness, while γ is the gradient constancy weight passed in GradientConstancy. These two parameters, which constitute the model parameters of the FDRIG algorithm, are described in more detail below.

The **DDRAW algorithm** is based on the minimization of an energy functional that contains the following assumptions:

Constancy of the gray values: It is assumed that corresponding pixels in consecutive images of an image sequence have the same gray value, i.e., that $f(\mathbf{x}+\mathbf{w}) = f(\mathbf{x})$.

Large displacements: It is assumed that large displacements, i.e., displacements larger than one pixel, occur. Under this assumption, it makes sense to consciously abstain from using the linearization of the constancy assumptions in the model that is typically proposed in the literature.

Statistical robustness in the data term: To reduce the influence of outliers, i.e., points that violate the constancy assumptions, they are penalized in a statistically robust manner, i.e., the customary non-robust square penalization $\Psi_{\mathrm{D}}(s^2) = s^2$ is replaced by a linear penalization via $\Psi_{\mathrm{S}}(s^2) = \sqrt{s^2+\epsilon^2}$, where $\epsilon = 0.001$ is a fixed regularization constant.

Preservation of discontinuities in the flow field II: The solution is assumed to be piecewise smooth. In contrast to the FDRIG algorithm, which allows discontinuities everywhere, the DDRAW algorithm only allows discontinuities at the edges in the original image. Here, the local smoothness is controlled in such a way that the flow field is sharp across image edges, while it is smooth along the image edges. This type of smoothness term is called data-driven and anisotropic.

All assumptions of the DDRAW algorithm can be combined into the following energy functional:

$$\begin{aligned} E_{\mathrm{DDRAW}}(\mathbf{w}) \;=\; & \int \Psi_{\mathrm{S}}\Big(\underbrace{|f(\mathbf{x}+\mathbf{w}) - f(\mathbf{x})|^2}_{\text{gray value constancy}} \Big) dr dc \\ & + \alpha \int \big(\underbrace{\nabla_2 u(\mathbf{x})^\top P_{\mathrm{NE}}(\nabla_2 f(\mathbf{x}))\, \nabla_2 u(\mathbf{x}) + \nabla_2 v(\mathbf{x})^\top P_{\mathrm{NE}}(\nabla_2 f(\mathbf{x}))\, \nabla_2 v(\mathbf{x})}_{\text{smoothness assumption}} \big) dr dc\,, \end{aligned}$$

where $P_{\mathrm{NE}}(\nabla_2 f(\mathbf{x}))$ is a normalized projection matrix orthogonal to $\nabla_2 f(\mathbf{x})$, for which

$$P_{\mathrm{NE}}(\nabla_2 f(\mathbf{x})) = \frac{1}{|\nabla_2 f(\mathbf{x})|^2 + 2\epsilon_{\mathrm{S}}^2} \begin{pmatrix} f_c^2(\mathbf{x}) + \epsilon_{\mathrm{S}}^2 & -f_r(\mathbf{x}) f_c(\mathbf{x}) \\ -f_r(\mathbf{x}) f_c(\mathbf{x}) & f_r^2(\mathbf{x}) + \epsilon_{\mathrm{S}}^2 \end{pmatrix} .$$

holds. This matrix ensures that the smoothness of the flow field is only assumed along the image edges. In contrast, no assumption is made with respect to the smoothness across the image edges, resulting in the fact that discontinuities in the solution may occur across the image edges. In this respect, $\epsilon_{\mathrm{S}} = 0.001$ serves as a regularization parameter that prevents the projection matrix $P_{\mathrm{NE}}(\nabla_2 f(\mathbf{x}))$ from becoming singular. In contrast to the FDRIG algorithm, there is only one model parameter for the DDRAW algorithm: the regularization parameter α. As mentioned above, α is described in more detail below.

As for the two approaches described above, the **CLG algorithm** uses certain assumptions:

Constancy of the gray values: It is assumed that corresponding pixels in consecutive images of an image sequence have the same gray value, i.e., that $f(\mathbf{x}+\mathbf{w}) = f(\mathbf{x})$.

Small displacements: In contrast to the two approaches above, it is assumed that only small displacements can occur, i.e., displacements in the order of a few pixels. This facilitates a linearization of the constancy assumptions in the model, and leads to the approximation $f(\mathbf{x}) + \nabla_3 f(\mathbf{x})^\top \mathbf{w}(\mathbf{x}) = f(\mathbf{x})$, i.e., $\nabla_3 f(\mathbf{x})^\top \mathbf{w}(\mathbf{x}) = 0$ should hold. Here, $\nabla_3 f(\mathbf{x})$ denotes the gradient in the spatial as well as the temporal domain.

Local constancy of the solution: Furthermore, it is assumed that the flow field to be computed is locally constant. This facilitates the integration of the image data in the data term over the respective neighborhood of each pixel. This, in turn, increases the robustness of the algorithm against noise. Mathematically, this can be achieved by reformulating the quadratic data term as $(\nabla_3 f(\mathbf{x})^\top \mathbf{w}(\mathbf{x}))^2 = \mathbf{w}(\mathbf{x})^\top \nabla_3 f(\mathbf{x}) \nabla_3 f(\mathbf{x})^\top \mathbf{w}(\mathbf{x})$. By performing a local Gaussian-weighted integration over a neighborhood specified by the ρ (passed in `IntegrationSigma`), the following data term is obtained: $\mathbf{w}(\mathbf{x})^\top G_\rho * (\nabla_3 f(\mathbf{x}) \nabla_3 f(\mathbf{x})^\top)\, \mathbf{w}(\mathbf{x})$. Here, $G_\rho * \ldots$ denotes a convolution of the 3×3 matrix $\nabla_3 f(\mathbf{x}) \nabla_3 f(\mathbf{x})^\top$ with a Gaussian filter with a standard deviation of ρ (see `derivate_gauss`).

General smoothness of the flow field: Finally, the solution is assumed to be smooth everywhere in the image. This particular type of smoothness term is called homogeneous.

All of the above assumptions can be combined into the following energy functional:

$$E_{\mathrm{CLG}}(\mathbf{w}) = \int \underbrace{\big(\mathbf{w}(\mathbf{x})^\top G_\rho * (\nabla_3 f(\mathbf{x}) \nabla_3 f(\mathbf{x})^\top)\, \mathbf{w}(\mathbf{x})\big)}_{\text{gray value constancy}} dr dc + \alpha \int \big(\underbrace{|\nabla_2 u(\mathbf{x})|^2 + |\nabla_2 v(\mathbf{x})|^2}_{\text{smoothness assumption}} \big) dr dc\,,$$

The corresponding model parameters are the regularization parameter α as well as the integration scale ρ (passed in `IntegrationSigma`), which determines the size of the neighborhood over which to integrate the data term. These two parameters are described in more detail below.

To compute the optical flow vector field for two consecutive images of an image sequence with the FDRIG, DDRAW, or CLG algorithm, the solution that best fulfills the assumptions of the respective algorithm must be determined. From a mathematical point of view, this means that a minimization of the above energy functionals should be performed. For the FDRIG and DDRAW algorithms, so called coarse-to-fine warping strategies play an important role in this minimization, because they enable the calculation of large displacements. Thus, they are a suitable means to handle the omission of the linearization of the constancy assumptions numerically in these two approaches.

To calculate large displacements, coarse-to-fine warping strategies use two concepts that are closely interlocked: The successive refinement of the problem (coarse-to-fine) and the successive compensation of the current image pair by already computed displacements (warping). Algorithmically, such coarse-to-fine warping strategies can be described as follows:

1. First, both images of the current image pair are zoomed down to a very coarse resolution level.

2. Then, the optical flow vector field is computed on this coarse resolution.

3. The vector field is required on the next resolution level: It is applied there to the second image of the image sequence, i.e., the problem on the finer resolution level is compensated by the already computed optical flow field. This step is also known as warping.

4. The modified problem (difference problem) is now solved on the finer resolution level, i.e., the optical flow vector field is computed there.

5. The steps 3-4 are repeated until the finest resolution level is reached.

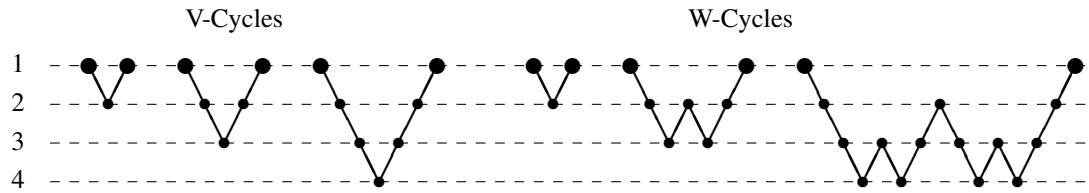
6. The final result is computed by adding up the vector fields from all resolution levels.

This incremental computation of the optical flow vector field has the following advantage: While the coarse-to-fine strategy ensures that the displacements on the finest resolution level are very small, the warping strategy ensures that the displacements remain small for the incremental displacements (optical flow vector fields of the difference problems). Since small displacements can be computed much more accurately than larger displacements, the accuracy of the results typically increases significantly by using such a coarse-to-fine warping strategy. However, instead of having to solve a single correspondence problem, an entire hierarchy of these problems must now be solved. For the CLG algorithm, such a coarse-to-fine warping strategy is unnecessary since the model already assumes small displacements.

The maximum number of resolution levels (warping levels), the resolution ratio between two consecutive resolution levels, as well as the finest resolution level can be specified for the FDRIG as well as the DDRAW algorithm. Details can be found below.

The minimization of functionals is mathematically very closely related to the minimization of functions: Like the fact that the zero crossing of the first derivative is a necessary condition for the minimum of a function, the fulfillment of the so called Euler-Lagrange equations is a necessary condition for the minimizing function of a functional (the minimizing function corresponds to the desired optical flow vector field in this case). The Euler-Lagrange equations are partial differential equations. By discretizing these Euler-Lagrange equations using finite differences, large sparse nonlinear equation systems result for the FDRIG and DDRAW algorithms. Because coarse-to-fine warping strategies are used, such an equation system must be solved for each resolution level, i.e., for each warping level. For the CLG algorithm, a single sparse linear equation system must be solved.

To ensure that the above nonlinear equation systems can be solved efficiently, the FDRIG and DDRAW use bidirectional multigrid methods. From a numerical point of view, these strategies are among the fastest methods for solving large linear and nonlinear equation systems. In contrast to conventional non-hierarchical iterative methods, e.g., the different linear and nonlinear Gauss-Seidel variants, the multigrid methods have the advantage that corrections to the solution can be determined efficiently on coarser resolution levels. This, in turn, leads to a significantly faster convergence. The basic idea of multigrid methods additionally consists of hierarchically computing these correction steps, i.e., the computation of the error on a coarser resolution level itself uses the same strategy and efficiently computes its error (i.e., the error of the error) by correction steps on an even coarser resolution level. Depending on whether one or two error correction steps are performed per cycle, a so called V or W cycle is obtained. The corresponding strategies for stepping through the resolution hierarchy are as follows for two to four resolution levels (where 1 is a fine and 4 is a coarse resolution level):



Here, iterations on the original problem are denoted by large markers, while small markers denote iterations on error correction problems.

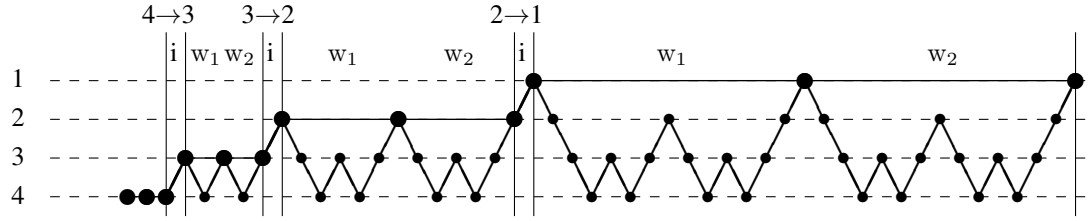
Algorithmically, a correction cycle can be described as follows:

1. In the first step, several (few) iterations using an interactive linear or nonlinear basic solver are performed (e.g., a variant of the Gauss-Seidel solver). This step is called pre-relaxation step.

2. In the second step, the current error is computed to correct the current solution (the solution after step 1). For efficiency reasons, the error is calculated on a coarser resolution level. This step, which can be performed iteratively several times, is called coarse grid correction step.

3. In a final step, again several (few) iterations using the interactive linear or nonlinear basic solver of step 1 are performed. This step is called post-relaxation step.

In addition, the solution can be initialized in a hierarchical manner. Starting from a very coarse variant of the original (non)linear equation system (level 4), the solution is successively refined. To do so, interpolated solutions of coarser variants of the equation system are used as the initialization of the next finer variant. On each resolution level itself, the V or W cycles described above are used to efficiently solve the (non)linear equation system on that resolution level. The corresponding multigrid methods are called full multigrid methods in the literature. The full multigrid algorithm can be visualized as follows:



This example represents a full multigrid algorithm that uses two W correction cycles per resolution level of the hierarchical initialization. The interpolation steps of the solution from one resolution level to the next are denoted by i and the two W correction cycles by w_1 and w_2. Iterations on the original problem are denoted by large markers, while small markers denote iterations on error correction problems.

In the multigrid implementation of the FDRIG, DDRAW, and CLG algorithm, the following parameters can be set: whether a hierarchical initialization is performed; the number of coarse grid correction steps; the maximum number of correction levels (resolution levels); the number of pre-relaxation steps; the number of post-relaxation steps. These parameters are described in more detail below.

The basic solver for the FDRIG algorithm is a point-coupled fixed-point variant of the linear Gauss-Seidel algorithm. The basic solver for the DDRAW algorithm is an alternating line-coupled fixed-point variant of the same type. The number of fixed-point steps can be specified for both algorithms with a further parameter. The basic solver for the CLG algorithm is a point-coupled linear Gauss-Seidel algorithm. The transfer of the data between the different resolution levels is performed by area-based interpolation and area-based averaging, respectively.

After the algorithms have been described, the effects of the individual parameters are discussed in the following.

The input images, along with their domains (regions of interest) are passed in `ImageT1` and `ImageT2`. The computation of the optical flow vector field `VectorField` is performed on the smallest surrounding rectangle

of the intersection of the domains of ImageT1 and ImageT2. The domain of VectorField is the intersection of the two domains. Hence, by specifying reduced domains for ImageT1 and ImageT2, the processing can be focused and runtime can potentially be saved. It should be noted, however, that all methods compute a global solution of the optical flow. In particular, it follows that the solution on a reduced domain need not (and cannot) be identical to the resolution on the full domain restricted to the reduced domain.

SmoothingSigma specifies the standard deviation of the Gaussian kernel that is used to smooth both input images. The larger the value of SmoothingSigma, the larger the low-pass effect of the Gaussian kernel, i.e., the smoother the preprocessed image. Usually, SmoothingSigma = 0.8 is a suitable choice. However, other values in the interval $[0, 2]$ are also possible. Larger standard deviations should only be considered if the input images are very noisy. It should be noted that larger values of SmoothingSigma lead to slightly longer execution times.

IntegrationSigma specifies the standard deviation ρ of the Gaussian kernel G_ρ that is used for the local integration of the neighborhood information of the data term. This parameter is used only in the CLG algorithm and has no effect on the other two algorithms. Usually, IntegrationSigma = 1.0 is a suitable choice. However, other values in the interval $[0, 3]$ are also possible. Larger standard deviations should only be considered if the input images are very noisy. It should be noted that larger values of IntegrationSigma lead to slightly longer execution times.

FlowSmoothness specifies the weight α of the smoothness term with respect to the data term. The larger the value of FlowSmoothness, the smoother the computed optical flow field. It should be noted that choosing FlowSmoothness too small can lead to unusable results, even though statistically robust penalty functions are used, in particular if the warping strategy needs to predict too much information outside of the image. For byte images with a gray value range of $[0, 255]$, values of FlowSmoothness around 20 for the flow-driven FDRIG algorithm and around 1000 for the data-driven DDRAW algorithm and the homogeneous CLG algorithm typically yield good results.

GradientConstancy specifies the weight γ of the gradient constancy with respect to the gray value constancy. This parameter is used only in the FDRIG algorithm. For the other two algorithms, it does not influence the results. For byte images with a gray value range of $[0, 255]$, a value of GradientConstancy = 5 is typically a good choice, since then both constancy assumptions are used to the same extent. For large changes in illumination, however, significantly larger values of GradientConstancy may be necessary to achieve good results. It should be noted that for large values of the gradient constancy weight the smoothness parameter FlowSmoothness must also be chosen larger.

The parameters of the multigrid solver and for the coarse-to-fine warping strategy can be specified with the generic parameters MGParamName and MGParamValue. Usually, it suffices to use one of the four default parameter sets via MGParamName = *'default_parameters'* and MGParamValue = *'very_accurate'*, *'accurate'*, *'fast_accurate'*, or *'fast'*. The default parameter sets are described below. If the parameters should be specified individually, MGParamName and MGParamValue must be set to tuples of the same length. The values corresponding to the parameters specified in MGParamName must be specified at the corresponding position in MGParamValue.

MGParamName = *'warp_zoom_factor'* can be used to specify the resolution ratio between two consecutive warping levels in the coarse-to-fine warping hierarchy. *'warp_zoom_factor'* must be selected from the open interval $(0, 1)$. For performance reasons, *'warp_zoom_factor'* is typically set to 0.5, i.e., the number of pixels is halved in each direction for each coarser warping level. This leads to an increase of 33% in the calculations that need to be performed with respect to an algorithm that does not use warping. Values for *'warp_zoom_factor'* close to 1 can lead to slightly better results. However, they require a disproportionately larger computation time, e.g., 426% for *'warp_zoom_factor'* = 0.9.

MGParamName = *'warp_levels'* can be used to restrict the warping hierarchy to a maximum number of levels. For *'warp_levels'* = 0, the largest possible number of levels is used. If the image size does not allow to use the specified number of levels (taking the resolution ratio *'warp_zoom_factor'* into account), the largest possible number of levels is used. Usually, *'warp_levels'* should be set to 0.

MGParamName = *'warp_last_level'* can be used to specify the number of warping levels for which the flow increment should *no longer* be computed. Usually, *'warp_last_level'* is set to 1 or 2, i.e., a flow increment is computed for each warping level, or the finest warping level is skipped in the computation. Since in the latter case the computation is performed on an image of half the resolution of the original image, the gained computation time can be used to compute a more accurate solution, e.g., by using a full multigrid algorithm with additional iterations. The more accurate solution is then interpolated to the full resolution.

The three parameters that specify the coarse-to-fine warping strategy are only used in the FDRIG and DDRAW algorithms. They are ignored for the CLG algorithm.

MGParamName = *'mg_solver'* can be used to specify the general multigrid strategy for solving the (non)linear equation system (in each warping level). For *'mg_solver'* = *'multigrid'*, a normal multigrid algorithm (*without* coarse-to-fine initialization) is used, while for *'mg_solver'* = *'full_multigrid'* a full multigrid algorithm (*with* coarse-to-fine initialization) is used. Since a resolution reduction of 0.5 is used between two consecutive levels of the coarse-to-fine initialization (in contrast to the resolution reduction in the warping strategy, this value is hard-coded into the algorithm), the use of a full multigrid algorithm results in an increase of the computation time by approximately 33% with respect to the normal multigrid algorithm. Using *'mg_solver'* to *'full_multigrid'* typically yields numerically more accurate results than *'mg_solver'* = *'multigrid'*.

MGParamName = *'mg_cycle_type'* can be used to specify whether a V or W correction cycle is used per multigrid level. Since a resolution reduction of 0.5 is used between two consecutive levels of the respective correction cycle, using a W cycle instead of a V cycle increases the computation time by approximately 50%. Using *'mg_cycle_type'* = *'w'* typically yields numerically more accurate results than *'mg_cycle_type'* = *'v'*.

MGParamName = *'mg_levels'* can be used to restrict the multigrid hierarchy for the coarse-to-fine initialization as well as for the actual V or W correction cycles. For *'mg_levels'* = 0, the largest possible number of levels is used. If the image size does not allow to use the specified number of levels, the largest possible number of levels is used. Usually, *'mg_levels'* should be set to 0.

MGParamName = *'mg_cycles'* can be used to specify the total number of V or W correction cycles that are being performed. If a full multigrid algorithm is used, *'mg_cycles'* refers to *each* level of the coarse-to-fine initialization. Usually, one or two cycles are sufficient to yield a sufficiently accurate solution of the equation system. Typically, the larger *'mg_cycles'*, the more accurate the numerical results. This parameter enters almost linearly into the computation time, i.e., doubling the number of cycles leads approximately to twice the computation time.

MGParamName = *'mg_pre_relax'* can be used to specify the number of iterations that are performed on each level of the V or W correction cycles using the iterative basic solver *before* the actual error correction is performed. Usually, one or two pre-relaxation steps are sufficient. Typically, the larger *'mg_pre_relax'*, the more accurate the numerical results.

MGParamName = *'mg_post_relax'* can be used to specify the number of iterations that are performed on each level of the V or W correction cycles using the iterative basic solver *after* the actual error correction is performed. Usually, one or two post-relaxation steps are sufficient. Typically, the larger *'mg_post_relax'*, the more accurate the numerical results.

Like when increasing the number of correction cycles, increasing the number of pre- and post-relaxation steps increases the computation time asymptotically linearly. However, no additional restriction and prolongation operations (zooming down and up of the error correction images) are performed. Consequently, a moderate increase in the number of relaxation steps only leads to a slight increase in the computation times.

MGParamName = *'mg_inner_iter'* can be used to specify the number of iterations to solve the linear equation systems in each fixed-point iteration of the nonlinear basic solver. Usually, one iteration is sufficient to achieve a sufficient convergence speed of the multigrid algorithm. The increase in computation time is slightly smaller than for the increase in the relaxation steps. This parameter only influences the FDRIG and DDRAW algorithms since for the CLG algorithm no nonlinear equation system needs to be solved.

MGParamName = *'parallel_solver'* can be used to enable a parallel processing of the FDRIG algorithm. This changes the results of the operator slightly, but can allow for a faster processing. The value can be set to *'true'* or *'false'* (default).

As described above, usually it is sufficient to use one of the default parameter sets for the parameters described above by using MGParamName = *'default_parameters'* and MGParamValue = *'very_accurate'*, *'accurate'*, *'fast_accurate'*, or *'fast'*. If necessary, individual parameters can be modified after the default parameter set has been chosen by specifying a subset of the above parameters and corresponding values after *'default_parameters'* in MGParamName and MGParamValue (e.g., MGParamName = *['default_parameters','warp_zoom_factor']* and MGParamValue = *['accurate',0.6]*).

The default parameter sets use the following values for the above parameters:

'default_parameters' = *'very_accurate'*: *'warp_zoom_factor'* = 0.5, *'warp_levels'* = 0, *'warp_last_level'* = 1, *'mg_solver'* = *'full_multigrid'*, *'mg_cycle_type'* = *'w'*, *'mg_levels'* = 0, *'mg_cycles'* = 1, *'mg_pre_relax'* = 2, *'mg_post_relax'* = 2, *'mg_inner_iter'* = 1.

'default_parameters' = *'accurate'*: *'warp_zoom_factor'* = 0.5, *'warp_levels'* = 0, *'warp_last_level'* = 1, *'mg_solver'* = *'multigrid'*, *'mg_cycle_type'* = *'v'*, *'mg_levels'* = 0, *'mg_cycles'* = 1, *'mg_pre_relax'* = 1, *'mg_post_relax'* = 1, *'mg_inner_iter'* = 1.

'default_parameters' = 'fast_accurate': *'warp_zoom_factor'* = 0.5, *'warp_levels'* = 0, *'warp_last_level'* = 2, *'mg_solver' = 'full_multigrid'*, *'mg_cycle_type' = 'w'*, *'mg_levels'* = 0, *'mg_cycles'* = 1, *'mg_pre_relax'* = 2, *'mg_post_relax'* = 2, *'mg_inner_iter'* = 1.

'default_parameters' = 'fast': *'warp_zoom_factor'* = 0.5, *'warp_levels'* = 0, *'warp_last_level'* = 2, *'mg_solver' = 'multigrid'*, *'mg_cycle_type' = 'v'*, *'mg_levels'* = 0, *'mg_cycles'* = 1, *'mg_pre_relax'* = 1, *'mg_post_relax'* = 1, *'mg_inner_iter'* = 1.

It should be noted that for the CLG algorithm the two modes *'fast_accurate'* and *'fast'* are identical to the modes *'very_accurate'* and *'accurate'* since the CLG algorithm does not use a coarse-to-fine warping strategy.

Parameters

- ▷ **ImageT1** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Input image 1.
- ▷ **ImageT2** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Input image 2.
- ▷ **VectorField** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* vector_field
  Optical flow.
- ▷ **Algorithm** (input_control) . . . . . string ⇝ *string*
  Algorithm for computing the optical flow.
  Default: 'fdrig'
  List of values: `Algorithm` ∈ {'fdrig', 'ddraw', 'clg'}
- ▷ **SmoothingSigma** (input_control) . . . . . real ⇝ *real*
  Standard deviation for initial Gaussian smoothing.
  Default: 0.8
  Suggested values: `SmoothingSigma` ∈ {0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0}
  Restriction: SmoothingSigma >= 0.0
- ▷ **IntegrationSigma** (input_control) . . . . . real ⇝ *real*
  Standard deviation of the integration filter.
  Default: 1.0
  Suggested values: `IntegrationSigma` ∈ {0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.2, 2.4, 2.6, 2.8, 3.0}
  Restriction: IntegrationSigma >= 0.0
- ▷ **FlowSmoothness** (input_control) . . . . . real ⇝ *real*
  Weight of the smoothing term relative to the data term.
  Default: 20.0
  Suggested values: `FlowSmoothness` ∈ {10.0, 20.0, 30.0, 50.0, 70.0, 100.0, 200.0, 300.0, 500.0, 700.0, 1000.0, 1500.0, 2000.0}
  Restriction: FlowSmoothness >= 0.0
- ▷ **GradientConstancy** (input_control) . . . . . real ⇝ *real*
  Weight of the gradient constancy relative to the gray value constancy.
  Default: 5.0
  Suggested values: `GradientConstancy` ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 15.0, 20.0, 30.0, 40.0, 50.0, 70.0, 100.0}
  Restriction: GradientConstancy >= 0.0
- ▷ **MGParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
  Parameter name(s) for the multigrid algorithm.
  Default: 'default_parameters'
  List of values: `MGParamName` ∈ {'default_parameters', 'mg_solver', 'mg_cycle_type', 'mg_levels', 'mg_cycles', 'mg_pre_relax', 'mg_post_relax', 'mg_inner_iter', 'warp_levels', 'warp_zoom_factor', 'warp_last_level', 'parallel_solver'}
- ▷ **MGParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer
  Parameter value(s) for the multigrid algorithm.
  Default: 'accurate'
  Suggested values: `MGParamValue` ∈ {'very_accurate', 'accurate', 'fast_accurate', 'fast', 'multigrid', 'full_multigrid', 'v', 'w', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 'true', 'false'}

Example

```
grab_image (ImageT1, AcqHandle)
while (true)
  grab_image (ImageT2, AcqHandle)
  optical_flow_mg (ImageT1, ImageT2, VectorField, 'fdrig', 0.8, 1, 10, \
                   5, 'default_parameters', 'accurate')
  threshold (VectorField, Region, 1, 10000)
  copy_obj (ImageT2, ImageT1, 1, 1)
endwhile
```

Result

If the parameter values are correct, the operator `optical_flow_mg` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input images are available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Successors

`threshold`, `vector_field_length`

See also

`unwarp_image_vector_field`

References

T. Brox, A. Bruhn, N. Papenberg, and J. Weickert: High accuracy optical flow estimation based on a theory for warping. In T. Pajdla and J. Matas, editors, Computer Vision - ECCV 2004, volume 3024 of Lecture Notes in Computer Science, pages 25–36. Springer, Berlin, 2004.

A. Bruhn, J. Weickert, C. Feddern, T. Kohlberger, and C. Schnörr: Variational optical flow computation in real-time. IEEE Transactions on Image Processing, 14(5):608-615, May 2005.

H.-H. Nagel and W. Enkelmann: An investigation of smoothness constraints for the estimation of displacement vector fields from image sequences. IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(5):565-593, September 1986.

Ulrich Trottenberg, Cornelis Oosterlee, Anton Schüller: Multigrid. Academic Press, Inc., San Diego, 2000.

Module

Foundation

unwarp_image_vector_field (Image, VectorField : ImageUnwarped : :)

Unwarp an image using a vector field.

`unwarp_image_vector_field` unwarps the image `Image` using the vector field `VectorField` and returns the unwarped image in `ImageUnwarped`. The vector field must be of the semantic type `vector_field_relative` and is typically determined with `optical_flow_mg`. Hence, `unwarp_image_vector_field` can be used to unwarp the second input image of `optical_flow_mg` to the first input image. It should be noted that because of the above semantics the vector field image represents an inverse transformation from the destination image of the vector field to the source image.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **VectorField** (input_object) ..... singlechannelimage(-array) ⇝ *object :* vector_field
  Input vector field.
- ▷ **ImageUnwarped** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Unwarped image.

Example

```
optical_flow_mg (Image1, Image2, VectorField, 'fdrig', 0.8, 1, 20, \
                 5, 'default_parameters', 'accurate')
unwarp_image_vector_field (Image2, VectorField, ImageUnwarped)
```

Result

If the parameter values are correct, the operator `unwarp_image_vector_field` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input images are available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on domain level.
- Automatically parallelized on tuple level.

Possible Predecessors

`optical_flow_mg`

Module

Foundation

vector_field_length (VectorField : Length : Mode :)

Compute the length of the vectors of a vector field.

`vector_field_length` compute the length of the vectors of the vector field `VectorField` and returns them in `Length`. `vector_field_length` only accepts vector fields of the semantic type `vector_field_relative`. The parameter `Mode` can be used to specify how the lengths are computed. For `Mode` = *'length'*, the Euclidean length of the vectors is computed. For `Mode` = *'squared_length'*, the square of the length of the vectors is computed. This avoids having to compute a square root internally, which is a costly operation on many processors, and hence saves runtime on these processors. Note that the `VectorField` must be in relative coordinates as returned by `optical_flow_mg`.

Parameters

▷ **VectorField** (input_object) ..... singlechannelimage(-array) ⇝ *object :* vector_field
Input vector field

▷ **Length** (output_object) ..... singlechannelimage(-array) ⇝ *object :* real
Length of the vectors of the vector field.

▷ **Mode** (input_control) ..... string ⇝ *string*
Mode for computing the length of the vectors.
Default: 'length'
List of values: `Mode` ∈ {'length', 'squared_length'}

Result

If the parameter values are correct, the operator `vector_field_length` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input images are available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on domain level.
- Automatically parallelized on tuple level.

Possible Predecessors

optical_flow_mg

Possible Successors

threshold

Module

Foundation

12.14 Points

corner_response (Image : ImageCorner : Size, Weight :)

Searching corners in images.

The operator corner_response extracts gray value corners in an image. The formula for the calculation of the response is:

$$\begin{aligned} R(x,y) &= A(x,y) \cdot B(x,y) - C^2(x,y) - Weight \cdot (A(x,y) + B(x,y))^2 \\ A(x,y) &= W(u,v) * (\nabla_x I(x,y))^2 \\ B(x,y) &= W(u,v) * (\nabla_y I(x,y))^2 \\ C(c,y) &= W(u,v) * (\nabla_x I(x,y) \nabla_y I(x,y)) \end{aligned}$$

with I: input image, R: output image of the filter, A, and B: smoothed directional derivatives, C: smoothing of the products of the directional derivatives. The operator gauss_image is used for smoothing (W), the operator sobel_amp is used for calculating the derivative (∇).

The corner response function is invariant with regard to rotation. In order to achieve a suitable dependency of the function $R(x,y)$ on the local gradient, the parameter Weight must be set to 0.04. With this, only gray value corners will return positive values for $R(x,y)$, while straight edges will receive negative values. The output image type is identical to the input image type. Therefore, the negative output values are set to 0 if byte images are used as input images. If this is not desired, the input image should be converted into a real or int2 image with convert_image_type.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / int2 / real
  Input image.
- ▷ **ImageCorner** (output_object) ..... multichannel-image(-array) ⇝ *object :* byte / uint2 / int2 / real
  Result of the filtering.
  Number of elements: ImageCorner == Image
- ▷ **Size** (input_control) ..... integer ⇝ *integer*
  Desired filtersize of the graymask.
  Default: 3
  Suggested values: Size ∈ {3, 5, 7, 9, 11}
- ▷ **Weight** (input_control) ..... real ⇝ *real*
  Weighting.
  Default: 0.04
  Value range: 0.0 ≤ Weight ≤ 0.3
  Minimum increment: 0.001
  Recommended increment: 0.01

Example

```
read_image(&Fabrik,"fabrik");
```

```
corner_response(Fabrik,&CornerResponse,3,0.04);
local_max(CornerResponse,&LocalMax);
disp_image(Fabrik,WindowHandle);
set_color(WindowHandle,"red");
disp_region(LocalMax,WindowHandle);
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

local_max, threshold

See also

gauss_filter, sobel_amp, convert_image_type

References

C.G. Harris, M.J. Stephens, "A combined corner and edge detector"'; Proc. of the 4th Alvey Vision Conference; August 1988; pp. 147-152.

H. Breit, "Bestimmung der Kameraeigenbewegung und Gewinnung von Tiefendaten aus monokularen Bildfolgen"; Diplomarbeit am Lehrstuhl für Nachrichtentechnik der TU München; 30. September 1990.

Module

Foundation

dots_image (Image : DotImage : Diameter, FilterType, PixelShift :)

Enhance circular dots in an image.

dots_image enhances circular dots of diameter Diameter in the input image Image. Hence, dots_image is especially suited for the segmentation of dot prints, e.g., in OCR applications. The enhancement is performed by using matched filters with filter masks that are tuned for a particular dot size. For example, for Diameter $= 5$ the filter mask is given by:

$$\frac{1}{336}\begin{pmatrix} & & -21 & -21 & -21 & & \\ & -21 & 16 & 16 & 16 & -21 & \\ -21 & 16 & 16 & 16 & 16 & 16 & -21 \\ -21 & 16 & 16 & 16 & 16 & 16 & -21 \\ -21 & 16 & 16 & 16 & 16 & 16 & -21 \\ & -21 & 16 & 16 & 16 & -21 & \\ & & -21 & -21 & -21 & & \end{pmatrix}$$

The parameter FilterType selects whether *'dark'*, *'light'*, or *'all'* dots in the image should be enhanced. The PixelShift can be used either to increase the contrast of the output image (PixelShift > 0) or to dampen the values in extremely bright areas that would be cut off otherwise (PixelShift $= -1$).

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Input image.

▷ **DotImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Output image.

▷ **Diameter** (input_control) . . . . . integer ⇝ *integer*
Diameter of the dots to be enhanced.
Default: 5
List of values: `Diameter` ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23}

▷ **FilterType** (input_control) . . . . . string ⇝ *string*
Enhance dark, light, or all dots.
Default: 'light'
List of values: `FilterType` ∈ {'dark', 'light', 'all'}

▷ **PixelShift** (input_control) . . . . . integer ⇝ *integer*
Shift of the filter response.
Default: 0
List of values: `PixelShift` ∈ {-1, 0, 1, 2}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

`threshold`

Alternatives

`laplace`, `laplace_of_gauss`, `diff_of_gauss`, `derivate_gauss`, `convol_image`

Module

Foundation

points_foerstner (Image : : SigmaGrad, SigmaInt, SigmaPoints, ThreshInhom, ThreshShape, Smoothing, EliminateDoublets : RowJunctions, ColumnJunctions, CoRRJunctions, CoRCJunctions, CoCCJunctions, RowArea, ColumnArea, CoRRArea, CoRCArea, CoCCArea)

Detect points of interest using the Förstner operator.

`points_foerstner` extracts significant points from an image. Significant points are points that differ from their neighborhood, i.e., points where the image function changes in two dimensions. These changes occur on the one hand at the intersection of image edges (called junction points), and on the other hand at places where color or brightness differs from the surrounding neighborhood (called area points).

The point extraction takes place in two steps: In the first step the point regions, i.e., the inhomogeneous, isotropic regions, are extracted from the image. To do so, the smoothed matrix

$$M = S * \begin{pmatrix} \sum_{c=1}^{n} I_{x,c}^2 & \sum_{c=1}^{n} I_{x,c} I_{y,c} \\ \sum_{c=1}^{n} I_{x,c} I_{y,c} & \sum_{c=1}^{n} I_{y,c}^2 \end{pmatrix}$$

is calculated, where $I_{x,c}$ and $I_{y,c}$ are the first derivatives of each image channel and S stands for a smoothing. If `Smoothing` is *'gauss'*, the derivatives are computed with Gaussian derivatives of size `SigmaGrad` and the

smoothing is performed by a Gaussian of size SigmaInt. If Smoothing is *'mean'*, the derivatives are computed with a 3×3 Sobel filter (and hence SigmaGrad is ignored) and the smoothing is performed by a SigmaInt $\times$ SigmaInt mean filter. Then

$$\textit{inhomogeneity} = \mathrm{Trace}M$$

is the degree of inhomogeneity in the image and

$$\textit{isotropy} = 4 \cdot \frac{\mathrm{Det}M}{(\mathrm{Trace}M)^2}$$

is the degree of the isotropy of the texture in the image. Image points that have an inhomogeneity greater or equal to ThreshInhom and at the same time an isotropy greater or equal to ThreshShape are subsequently examined further.

In the second step, two optimization functions are calculated for the resulting points. Essentially, these optimization functions average for each point the distances to the edge directions (for junction points) and the gradient directions (for area points) within an observation window around the point. If Smoothing is *'gauss'*, the averaging is performed by a Gaussian of size SigmaPoints, if Smoothing is *'mean'*, the averaging is performed by a SigmaPoints $\times$ SigmaPoints mean filter. The local minima of the optimization functions determine the extracted points. Their subpixel precise position is returned in (RowJunctions, ColumnJunctions) and (RowArea, ColumnArea).

In addition to their position, for each extracted point the elements CoRRJunctions, CoRCJunctions, and CoCCJunctions (and CoRRArea, CoRCArea, and CoCCArea, respectively) of the corresponding covariance matrix are returned. This matrix facilitates conclusions about the precision of the calculated point position. To obtain the actual values, it is necessary to estimate the amount of noise in the input image and to multiply all components of the covariance matrix with the variance of the noise. (To estimate the amount of noise, apply intensity to homogeneous image regions or plane_deviation to image regions, where the gray values form a plane. In both cases the amount of noise is returned in the parameter Deviation.) This is illustrated by the example program points_foerstner_ellipses.hdev.

It lies in the nature of this operator that corners often result in two distinct points: One junction point, where the edges of the corner actually meet, and one area point inside the corner. Such doublets will be eliminated automatically, if EliminateDoublets is *'true'*. To do so, each pair of one junction point and one area point is examined. If the points lie within each others' observation window of the optimization function, for both points the precision of the point position is calculated and the point with the lower precision is rejected. If EliminateDoublets is *'false'*, every detected point is returned.

Attention

Note that only odd values for SigmaInt and SigmaPoints are allowed, if Smoothing is *'mean'*. Even values automatically will be replaced by the next larger odd value.

points_foerstner with Smoothing = *'gauss'* uses a special implementation that is optimized using SSE2 instructions if the system parameter *'sse2_enable'* is set to *'true'* (which is default if SSE2 is available on your machine). This implementation is slightly inaccurate compared to the pure C version due to numerical issues (for 'byte' images the difference in RowJunctions and ColumnJunctions is in order of magnitude of $1.0e-5$). If you prefer accuracy over performance you can set *'sse2_enable'* to *'false'* (using set_system) before you call points_foerstner. This way points_foerstner does not use SSE2 accelerations. Don't forget to set *'sse2_enable'* back to *'true'* afterwards.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / uint2 / real
Input image.

▷ **SigmaGrad** (input_control) ..... number ⇝ *real* / integer
Amount of smoothing used for the calculation of the gradient. If Smoothing is 'mean', SigmaGrad is ignored.
Default: 1.0
Suggested values: SigmaGrad ∈ {0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0}
Value range: 0.0 ≤ SigmaGrad
Recommended increment: 0.1

▷ **SigmaInt** (input_control) . . . . . number ⇝ *real* / integer
Amount of smoothing used for the integration of the gradients.
Default: 2.0
Suggested values: SigmaInt ∈ {0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0}
Recommended increment: 0.1
Restriction: SigmaInt > 0

▷ **SigmaPoints** (input_control) . . . . . number ⇝ *real* / integer
Amount of smoothing used in the optimization functions.
Default: 3.0
Suggested values: SigmaPoints ∈ {0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0}
Recommended increment: 0.1
Restriction: SigmaPoints >= SigmaInt && SigmaPoints > 0.6

▷ **ThreshInhom** (input_control) . . . . . number ⇝ *real* / integer
Threshold for the segmentation of inhomogeneous image areas.
Default: 200
Suggested values: ThreshInhom ∈ {50, 100, 200, 500, 1000}
Value range: 0.0 ≤ ThreshInhom

▷ **ThreshShape** (input_control) . . . . . real ⇝ *real*
Threshold for the segmentation of point areas.
Default: 0.3
Suggested values: ThreshShape ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.7}
Value range: 0.01 ≤ ThreshShape ≤ 1
Minimum increment: 0.01
Recommended increment: 0.1

▷ **Smoothing** (input_control) . . . . . string ⇝ *string*
Used smoothing method.
Default: 'gauss'
List of values: Smoothing ∈ {'gauss', 'mean'}

▷ **EliminateDoublets** (input_control) . . . . . string ⇝ *string*
Elimination of multiply detected points.
Default: 'false'
List of values: EliminateDoublets ∈ {'false', 'true'}

▷ **RowJunctions** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinates of the detected junction points.

▷ **ColumnJunctions** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinates of the detected junction points.

▷ **CoRRJunctions** (output_control) . . . . . number-array ⇝ *real*
Row part of the covariance matrix of the detected junction points.

▷ **CoRCJunctions** (output_control) . . . . . number-array ⇝ *real*
Mixed part of the covariance matrix of the detected junction points.

▷ **CoCCJunctions** (output_control) . . . . . number-array ⇝ *real*
Column part of the covariance matrix of the detected junction points.

▷ **RowArea** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinates of the detected area points.

▷ **ColumnArea** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinates of the detected area points.

▷ **CoRRArea** (output_control) . . . . . number-array ⇝ *real*
Row part of the covariance matrix of the detected area points.

▷ **CoRCArea** (output_control) . . . . . number-array ⇝ *real*
Mixed part of the covariance matrix of the detected area points.

▷ **CoCCArea** (output_control) . . . . . number-array ⇝ *real*
Column part of the covariance matrix of the detected area points.

Result

points_foerstner returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during the execution. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

gen_cross_contour_xld, disp_cross

Alternatives

points_harris, points_lepetit, points_harris_binomial

References

W. Förstner, E. Gülch: "A Fast Operator for Detection and Precise Location of Distinct Points, Corners and Circular features". In Proceedings of the Intercommission Conference on Fast Processing of Photogrametric Data, Interlaken, pp. 281-305, 1987.

W. Förstner: "Statistische Verfahren für die automatische Bildanalyse und ihre Bewertung bei der Objekterkennung und -vermessung". Volume 370, Series C, Deutsche Geodätische Kommission, München, 1991.

W. Förstner: "A Framework for Low Level Feature Extraction". European Conference on Computer Vision, LNCS 802, pp. 383-394, Springer Verlag, 1994.

C. Fuchs: "Extraktion polymorpher Bildstrukturen und ihre topologische und geometrische Gruppierung". Volume 502, Series C, Deutsche Geodätische Kommission, München, 1998.

Module

Foundation

points_harris (Image : : SigmaGrad, SigmaSmooth, Alpha, Threshold : Row, Column)

Detect points of interest using the Harris operator.

points_harris extracts points of interest from an image. The Harris operator is based upon the smoothed matrix

$$M = G_\sigma * \begin{pmatrix} \sum_{c=1}^{n} I_{x,c}^2 & \sum_{c=1}^{n} I_{x,c} I_{y,c} \\ \sum_{c=1}^{n} I_{x,c} I_{y,c} & \sum_{c=1}^{n} I_{y,c}^2 \end{pmatrix},$$

where G_σ stands for a Gaussian smoothing of size SigmaSmooth and $I_{x,c}$ and $I_{y,c}$ are the first derivatives of each image channel, computed with Gaussian derivatives of size SigmaGrad. The resulting points are the positive local extrema of

$$\mathrm{Det}M - \mathtt{Alpha} * (\mathrm{Trace}M)^2.$$

If necessary, they can be restricted to points with a minimum filter response of Threshold. The coordinates of the points are calculated with subpixel accuracy.

Attention

points_harris uses a special implementation that is optimized using SSE2 instructions if the system parameter *'sse2_enable'* is set to *'true'* (which is default if SSE2 is available on your machine). This implementation is slightly inaccurate compared to the pure C version due to numerical issues (for 'byte' images the difference in Row and Column is in order of magnitude of $1.0e-5$). If you prefer accuracy over performance you can set *'sse2_enable'* to *'false'* (using set_system) before you call points_harris. This way points_harris does not use SSE2 accelerations. Don't forget to set *'sse2_enable'* back to *'true'* afterwards.

points_harris can be executed on an OpenCL device if both SigmaGrad and SigmaSmooth induce a filter size of no more than 129 pixels. This corresponds to a value of less than 20.7 for both parameters. As with

the SSE2 version, the results of the OpenCL implementation may diverge slightly from that of pure C version due to numerical issues.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2 / real
Input image.

▷ **SigmaGrad** (input_control) . . . . . real ⇝ *real*
Amount of smoothing used for the calculation of the gradient.
Default: 0.7
Suggested values: SigmaGrad ∈ {0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0}
Recommended increment: 0.1
Restriction: SigmaGrad > 0.0

▷ **SigmaSmooth** (input_control) . . . . . real ⇝ *real*
Amount of smoothing used for the integration of the gradients.
Default: 2.0
Suggested values: SigmaSmooth ∈ {0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0}
Recommended increment: 0.1
Restriction: SigmaSmooth > 0.0

▷ **Alpha** (input_control) . . . . . real ⇝ *real*
Weight of the squared trace of the squared gradient matrix.
Default: 0.08
Suggested values: Alpha ∈ {0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08}
Value range: 0.0 ≤ Alpha
Minimum increment: 0.001
Recommended increment: 0.01

▷ **Threshold** (input_control) . . . . . number ⇝ *real* / integer
Minimum filter response for the points.
Default: 1000.0
Restriction: Threshold >= 0.0

▷ **Row** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinates of the detected points.

▷ **Column** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinates of the detected points.

Result

points_harris returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during the execution. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

gen_cross_contour_xld

Alternatives

points_foerstner, points_lepetit, points_harris_binomial

References

C. Harris, M. Stephens: "A combined corner and edge detector". Proceedings of the 4th Alvey Vision Conference, pp. 147-151, 1988.

V. Gouet, N.Boujemaa: "Object-based queries using color points of interest". IEEE Workshop on Content-Based Access of Image and Video Libraries, CVPR/CBAIVL 2001, Hawaii, USA, 2001.

Module

Foundation

points_harris_binomial (Image : : MaskSizeGrad, MaskSizeSmooth, Alpha, Threshold, Subpix : Row, Column)

Detect points of interest using the binomial approximation of the Harris operator.

points_harris extracts points of interest from an image Image. The Harris operator is based upon the smoothed matrix

$$M = G_\sigma * \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} ,$$

where G_σ stands for a Binomial smoothing of size MaskSizeSmooth and I_x and I_y are the first derivatives of the image, computed with a Sobel filter of size MaskSizeGrad. The resulting points are the positive local extrema of

$$\mathrm{Det}M - \texttt{Alpha} * (\mathrm{Trace}M)^2.$$

If necessary, they can be restricted to points with a minimum filter response of Threshold. By default the coordinates of the points are calculated with subpixel accuracy. This can be turned off by setting the parameter Subpix to 'off'.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image.

▷ **MaskSizeGrad** (input_control) . . . . . integer ⇝ *integer*
Amount of binomial smoothing used for the calculation of the gradient.
Default: 5
Suggested values: MaskSizeGrad ∈ {3, 5, 7, 9, 11, 15, 21, 31}
Value range: 1 ≤ MaskSizeGrad
Recommended increment: 2

▷ **MaskSizeSmooth** (input_control) . . . . . integer ⇝ *integer*
Amount of smoothing used for the integration of the gradients.
Default: 15
Suggested values: MaskSizeSmooth ∈ {3, 5, 7, 9, 11, 15, 21, 31}
Value range: 1 ≤ MaskSizeSmooth
Recommended increment: 2

▷ **Alpha** (input_control) . . . . . real ⇝ *real*
Weight of the squared trace of the squared gradient matrix.
Default: 0.08
Suggested values: Alpha ∈ {0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08}
Value range: 0.0 ≤ Alpha
Minimum increment: 0.001
Recommended increment: 0.01

▷ **Threshold** (input_control) . . . . . number ⇝ *real* / integer
Minimum filter response for the points.
Default: 1000.0
Value range: 0.0 ≤ Threshold

▷ **Subpix** (input_control) . . . . . string ⇝ *string*
Turn on or off subpixel refinement.
Default: 'on'
List of values: Subpix ∈ {'on', 'off'}

▷ **Row** (output_control) ..... point.y-array ⇝ *real*
Row coordinates of the detected points.

▷ **Column** (output_control) ..... point.x-array ⇝ *real*
Column coordinates of the detected points.

Result

points_harris returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during the execution. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

gen_cross_contour_xld

Alternatives

points_foerstner, points_harris, points_lepetit, points_sojka

References

C. Harris, M. Stephens: "A combined corner and edge detector". Proceedings of the 4th Alvey Vision Conference, pp. 147-151, 1988.

V. Gouet, N.Boujemaa: "Object-based queries using color points of interest". IEEE Workshop on Content-Based Access of Image and Video Libraries, CVPR/CBAIVL 2001, Hawaii, USA, 2001.

Module

Foundation

points_lepetit (Image : : Radius, CheckNeighbor, MinCheckNeighborDiff, MinScore, Subpix : Row, Column)

Detect points of interest using the Lepetit operator.

points_lepetit extracts points of interest like corners or blob-like structures from Image. The Image is first smoothed with a median of size 3x3. Then, all the gray values on a circle with radius Radius around an interest point candidate (*m*) are examined. The absolute differences of two diagonally opposed gray values (*m1*,*m2*) on the circle to the central pixel *m* is computed. At least one of these differences has to be larger than MinCheckNeighborDiff. All diagonally opposed pixels on the circle must fulfill that condition. To suppress detection of points at edges that have a small curvature (aliasing), it is possible to compute CheckNeighbor further differences of circle point neighbors of *m1* and *m2* to the center, that as well fulfill the above criteria. By computing all gray value differences of the circle points to the center, a mean gray value difference is determined. That value has to be larger than MinScore and allows to restrict the results to points with high contrast. By computing the score of all eight neighbors of m, it is possible to fit a quadratic equation to that. The maxima of that equation determines a subpixel accurate interest point position. By setting the parameter Subpix to 'interpolation' (default) or 'none', it is possible to turn that refinement step on or off. The resulting points are returned in Row and Column. The operator points_lepetit can especially be used for very fast interest point extraction. The results are however less robust than points extracted by points_harris for example.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
Input image.

▷ **Radius** (input_control) ..... integer ⇝ *integer*
Radius of the circle.
Default: 3
Suggested values: Radius ∈ {3, 5, 6, 7, 8, 9, 10, 15}

▷ **CheckNeighbor** (input_control) ..... integer ⇝ *integer*
Number of checked neighbors on the circle.
Default: 1
Suggested values: CheckNeighbor ∈ {1, 2, 3, 5}

▷ **MinCheckNeighborDiff** (input_control) ..... integer ⇝ *integer*
Threshold of gray value difference to each circle point.
Default: 15
Suggested values: MinCheckNeighborDiff ∈ {10, 15, 20, 25, 30, 35, 40, 45, 60, 80}

▷ **MinScore** (input_control) ..... integer ⇝ *integer*
Threshold of gray value difference to all circle points.
Default: 30
Suggested values: MinScore ∈ {5, 10, 15, 20, 25, 30}

▷ **Subpix** (input_control) ..... string ⇝ *string*
Subpixel accuracy of point coordinates.
Default: 'interpolation'
List of values: Subpix ∈ {'none', 'interpolation'}

▷ **Row** (output_control) ..... point.y-array ⇝ *integer* / real
Row-coordinates of the detected points.

▷ **Column** (output_control) ..... point.x-array ⇝ *integer* / real
Column-coordinates of the detected points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on domain level.

Possible Predecessors

gauss_filter

Alternatives

points_foerstner, points_harris, points_harris_binomial, points_sojka

Module

Foundation

points_sojka (Image : : MaskSize, SigmaW, SigmaD, MinGrad, MinApparentness, MinAngle, Subpix : Row, Column)

Find corners using the Sojka operator.

points_sojka defines a corner as the point of intersection of two straight, non-collinear gray value edges. To decide whether a point of the input image Image is a corner or not, a neighborhood of MaskSize × MaskSize points is inspected. Only those image regions that are relevant for the decision are considered. Pixels with a magnitude of the gradient of less than MinGrad are ignored from the outset.

Furthermore, only those of the remaining points are used that belong to one of the two gray value edges that form the corner. For this, the so called *Apparentness* is calculated, which is an indicator of the probability that the examined point actually is a corner point. Essentially, it is determined by the number of relevant points and their gradients. A point can only be accepted as a corner when its *Apparentness* is at least MinApparentness. Typical values of MinApparentness should range in the region of a few multiples of MinGrad.

To calculate the *Apparentness*, each mask point is weighted according to two criteria: First, the influence of a mask point is weighted with a Gaussian of size SigmaW according to its distance from the possible corner point. SigmaW should be roughly between quarter and half of MaskSize to obtain a reasonable proportion of the size of the weighting function to the mask size. Secondly, the distance of the point from the (assumed) ideal gray value edge is estimated and the point is weighted with a Gaussian of size SigmaD according to that distance. I.e., pixels that (due to the discretization of the input image) lie farther from the ideal gray value edge have less influence on the result than pixels with a smaller distance. Typically, it is not necessary to modify the default value *0.75* of SigmaD .

As a further criterion, the angle is calculated, by which the gray value edges change their direction in the corner point. A point can only be accepted as a corner when this angle is greater than MinAngle.

The position of the detected corner points is returned in (Row, Column). Row and Column are calculated with subpixel accuracy if Subpix is *'true'*. They are calculated only with pixel accuracy if Subpix is *'false'*.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
  Input image.
- ▷ **MaskSize** (input_control) ..... integer ⇝ *integer*
  Required filter size.
  Default: 9
  List of values: MaskSize ∈ {5, 7, 9, 11, 13}
- ▷ **SigmaW** (input_control) ..... number ⇝ *real* / integer
  Sigma of the weight function according to the distance to the corner candidate.
  Default: 2.5
  Suggested values: SigmaW ∈ {2.0, 2.2, 2.4, 2.5, 2.6, 2.8, 3.0}
  Restriction: 2.0 <= SigmaW && SigmaW <= 3.0
- ▷ **SigmaD** (input_control) ..... number ⇝ *real* / integer
  Sigma of the weight function for the distance to the ideal gray value edge.
  Default: 0.75
  Suggested values: SigmaD ∈ {0.6, 0.7, 0.75, 0.8, 0.9, 1.0}
  Restriction: 0.6 <= SigmaD && SigmaD <= 1.0
- ▷ **MinGrad** (input_control) ..... number ⇝ *real* / integer
  Threshold for the magnitude of the gradient.
  Default: 30.0
  Suggested values: MinGrad ∈ {20.0, 15.0, 30.0, 35.0, 40.0}
- ▷ **MinApparentness** (input_control) ..... number ⇝ *real* / integer
  Threshold for $Apparentness$.
  Default: 90.0
  Suggested values: MinApparentness ∈ {30.0, 60.0, 90.0, 150.0, 300.0, 600.0, 1500.0}
- ▷ **MinAngle** (input_control) ..... angle.rad ⇝ *real*
  Threshold for the direction change in a corner point (radians).
  Default: 0.5
  Restriction: 0.0 <= MinAngle && MinAngle <= pi
- ▷ **Subpix** (input_control) ..... string ⇝ *string*
  Subpixel precise calculation of the corner points.
  Default: 'false'
  List of values: Subpix ∈ {'false', 'true'}
- ▷ **Row** (output_control) ..... point.y-array ⇝ *real*
  Row coordinates of the detected corner points.
- ▷ **Column** (output_control) ..... point.x-array ⇝ *real*
  Column coordinates of the detected corner points.

Result

points_sojka returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during the execution. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

References

Eduard Sojka: "A New and Efficient Algorithm for Detecting the Corners in Digital Images". Pattern Recognition, Luc Van Gool (Editor), LNCS 2449, pp. 125-132, Springer Verlag, 2002.

Module

Foundation

12.15 Scene Flow

```
scene_flow_calib ( ImageRect1T1, ImageRect2T1, ImageRect1T2,
    ImageRect2T2, Disparity : : SmoothingFlow, SmoothingDisparity,
    GenParamName, GenParamValue, CamParamRect1, CamParamRect2,
    RelPoseRect : ObjectModel3D )
```

Compute the calibrated scene flow between two stereo image pairs.

`scene_flow_calib` computes the calibrated scene flow between two consecutive rectified stereo image pairs. The scene flow is the three-dimensional position and motion of surface points in a dynamic scene. The movement in the images can be caused by objects that move in the world or by a movement of the camera (or both) between the acquisition of the two image pairs.

The scene flow is returned as the 3D object model `ObjectModel3D`. The 3D object model contains the coordinates of the reconstructed 3D points. Furthermore, the 3D flow is encoded in the 3D object model by the attributes *'&flow_x'*, *'&flow_y'*, and *'&flow_z'*.

The two consecutive stereo image pairs of the image sequence are passed in `ImageRect1T1`, `ImageRect2T1`, `ImageRect1T2`, and `ImageRect2T2`. Each stereo image pair must be rectified. Note that the images can be rectified by using the operators `calibrate_cameras`, `gen_binocular_rectification_map`, and `map_image`.

The camera geometry of the rectified binocular camera system is specified by its internal camera parameters `CamParamRect1` of the rectified camera 1 and `CamParamRect2` of the rectified camera 2, and the pose `RelPoseRect` that defines the pose of the rectified camera 2 in relation to the rectified camera 1. These camera parameters can be obtained from the operators `calibrate_cameras` and `gen_binocular_rectification_map`. The focal length and scale factor of the rectified camera system 1 and 2 must be equal.

Furthermore, a single-channel `Disparity` image is required, which specifies for each pixel $(r, c1)$ of the image `ImageRect1T1` a matching pixel $(r, c2)$ of `ImageRect2T1` according to the equation $c2 = c1 + d(r, c1)$, where $d(r, c)$ is the `Disparity` at pixel (r, c). The disparity image can be computed using `binocular_disparity` or `binocular_disparity_mg`.

To calculate the calibrated scene flow, internally `scene_flow_uncalib` is first executed. The results are then converted to 3D points and 3D flow vectors using the stereo camera geometry parameters described above.

For a description of the remaining parameters of `scene_flow_calib`, please refer to `scene_flow_uncalib`.

Parameters

- ▷ **ImageRect1T1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Input image 1 at time t_1.
- ▷ **ImageRect2T1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Input image 2 at time t_1.
- ▷ **ImageRect1T2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Input image 1 at time t_2.
- ▷ **ImageRect2T2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Input image 2 at time t_2.
- ▷ **Disparity** (input_object) ..... singlechannelimage(-array) ⇝ *object :* real
  Disparity between input images 1 and 2 at time t_1.

▷ **SmoothingFlow** (input_control) . . . . . number ⇝ *real* / integer
Weight of the regularization term relative to the data term (derivatives of the optical flow).
Default: 40.0
Suggested values: SmoothingFlow ∈ {10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0}
Restriction: SmoothingFlow > 0.0

▷ **SmoothingDisparity** (input_control) . . . . . number ⇝ *real* / integer
Weight of the regularization term relative to the data term (derivatives of the disparity change).
Default: 40.0
Suggested values: SmoothingDisparity ∈ {10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0}
Restriction: SmoothingDisparity > 0.0

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Parameter name(s) for the algorithm.
Default: 'default_parameters'
Suggested values: GenParamName ∈ {'default_parameters', 'warp_levels', 'warp_zoom_factor', 'warp_last_level', 'outer_iter', 'inner_iter', 'sor_iter', 'omega'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Parameter value(s) for the algorithm.
Default: 'accurate'
Suggested values: GenParamValue ∈ {'very_accurate', 'accurate', 'fast', 'very_fast', 0, 1, 2, 3, 4, 5, 6, 0.5, 0.6, 0.7, 0.75, 3, 5, 7, 2, 3, 1.9}

▷ **CamParamRect1** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters of the rectified camera 1.

▷ **CamParamRect2** (input_control) . . . . . campar ⇝ *real* / integer / string
Internal camera parameters of the rectified camera 2.

▷ **RelPoseRect** (input_control) . . . . . pose ⇝ *real* / integer
Pose of the rectified camera 2 in relation to the rectified camera 1.
Number of elements: 7

▷ **ObjectModel3D** (output_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

Result

If the parameter values are correct, the operator scene_flow_calib returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

binocular_disparity, binocular_disparity_mg

Alternatives

scene_flow_uncalib, optical_flow_mg

References

A. Wedel, C. Rabe, T. Vaudrey, T. Brox, U. Franke and D. Cremers: "Efficient dense scene flow from sparse or dense stereo data"; In: Proceedings of the 10th European Conference on Computer Vision: Part I, pages 739-751. Springer-Verlag, 2008.

Module

Foundation

scene_flow_uncalib (ImageRect1T1, ImageRect2T1, ImageRect1T2, ImageRect2T2, Disparity : OpticalFlow, DisparityChange : SmoothingFlow, SmoothingDisparity, GenParamName, GenParamValue :)

Compute the uncalibrated scene flow between two stereo image pairs.

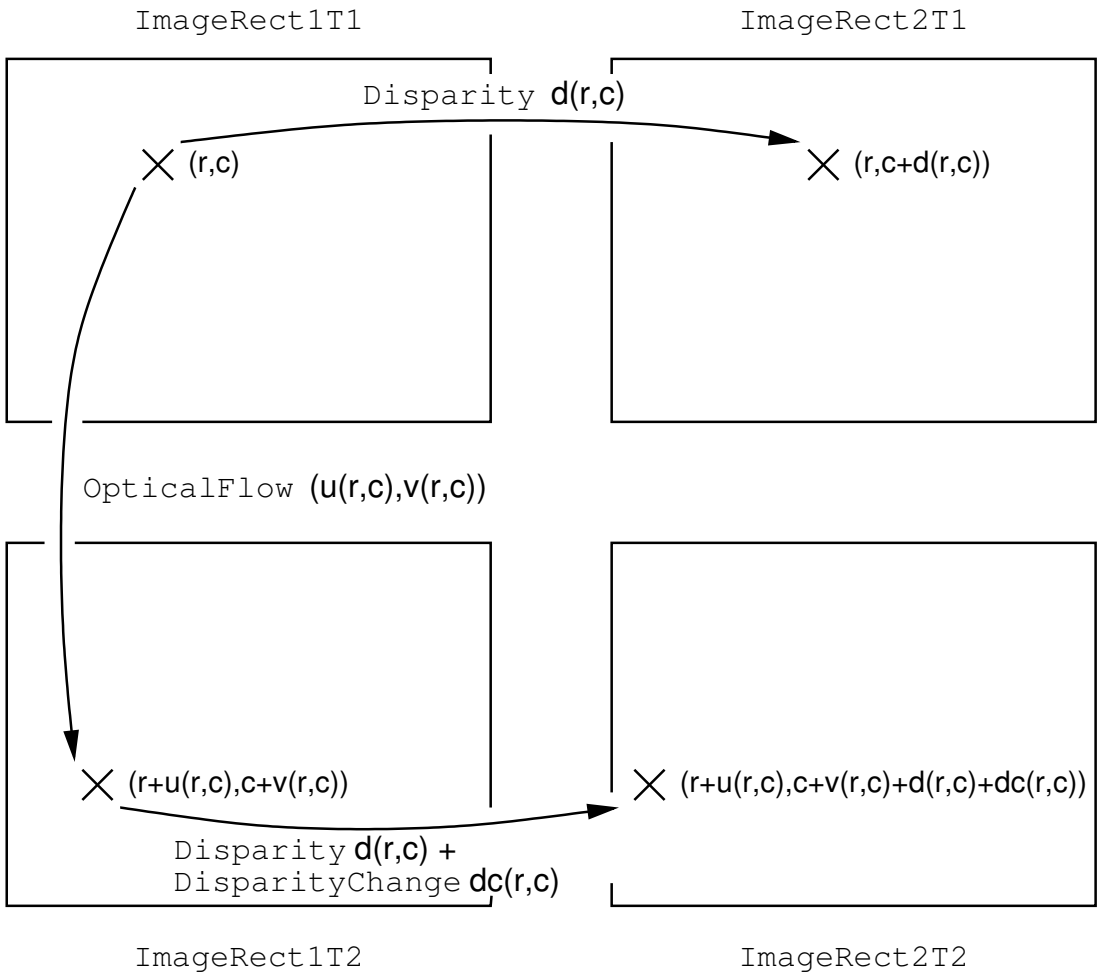
scene_flow_uncalib computes the uncalibrated scene flow between two consecutive rectified stereo image pairs. The scene flow is the three-dimensional position and motion of surface points in a dynamic scene. The movement in the images can be caused by objects that move in the world or by a movement of the camera (or both) between the acquisition of the two image pairs. To calculate the calibrated scene flow, scene_flow_calib can be used.

The two consecutive stereo image pairs of the image sequence are passed in ImageRect1T1, ImageRect2T1, ImageRect1T2, and ImageRect2T2. Each stereo image pair must be rectified. Note that the images can be rectified by using the operators calibrate_cameras, gen_binocular_rectification_map, and map_image. Furthermore, a single-channel Disparity image is required, which specifies for each pixel $(r, c1)$ of the image ImageRect1T1 a matching pixel $(r, c2)$ of ImageRect2T1 according to the equation $c2 = c1 + d(r, c1)$, where $d(r, c)$ is the Disparity at pixel (r, c). The disparity image can be computed using binocular_disparity or binocular_disparity_mg.

The computed uncalibrated scene flow is returned in OpticalFlow and DisparityChange. The vectors in the vector field OpticalFlow represent the movement in the image plane between ImageRect1T1 and ImageRect1T2. The single-channel image DisparityChange describes the change in disparity between ImageRect2T1 and ImageRect2T2. A world point is projected into ImageRect1T1 at position (r, c). The same point is projected into

- ImageRect2T1 at position $(r, c + d(r, c))$,
- ImageRect1T2 at position $(r + u(r, c), c + v(r, c))$,
- ImageRect2T2 at position $(r + u(r, c), c + v(r, c) + d(r, c) + dc(r, c))$,

where $u(r, c)$ and $v(r, c)$ denote the values of the row and column components of the vector field image OpticalFlow, $d(r, c)$ denotes the Disparity, and $dc(r, c)$ the DisparityChange at the pixel (r, c).



Relations between the four images and the optical flow as well as the disparities in the disparity images.

Parameter Description

The rectified input images are passed in `ImageRect1T1`, `ImageRect2T1`, `ImageRect1T2`, and `ImageRect2T2`. The computation of the scene flow is performed on the domain of `ImageRect1T1`, which is also the domain of the scene flow in `OpticalFlow` and `DisparityChange`. `Disparity` describes the disparity between the rectified images `ImageRect1T1` and `ImageRect2T1`.

`SmoothingFlow` and `SmoothingDisparity` specify the regularization weights λ and γ with respect to the data term. The larger the value of these parameters, the smoother the computed scene flow is. For byte images with a gray value range of $[0, 255]$, values around 40 typically yield good results.

The parameters of the iteration scheme and for the coarse-to-fine warping strategy can be specified with the generic parameters `GenParamName` and `GenParamValue`.

Usually, it is sufficient to use one of the default parameter sets for the parameters by using `GenParamName` = *'default_parameters'* and `GenParamValue` = *'very_accurate'*, *'accurate'*, *'fast'*, or *'very_fast'*. If necessary, individual parameters can be modified after the default parameter set has been chosen by specifying a subset of the parameters and corresponding values after *'default_parameters'* in `GenParamName` and `GenParamValue` (e.g., `GenParamName` = *['default_parameters','warp_zoom_factor']* and `GenParamValue` = *['accurate',0.6]*). The meaning of the individual parameters is described in detail below. The default parameter sets are given by:

'default_parameters'	*'very_accurate'*	*'accurate'*	*'fast'*	*'very_fast'*
'warp_zoom_factor'	0.75	0.5	0.5	0.5
'warp_levels'	0	0	0	0
'warp_last_level'	1	1	1	2
'outer_iter'	10	7	5	4
'inner_iter'	2	2	2	2
'sor_iter'	3	3	3	3
'omega'	1.9	1.9	1.9	1.9

If the parameters should be specified individually, `GenParamName` and `GenParamValue` must be set to tuples of the same length. The values corresponding to the parameters specified in `GenParamName` must be specified at the corresponding position in `GenParamValue`. For a deeper understanding of the following parameters, please refer to the section *Algorithm* below.

- `GenParamName` = *'warp_zoom_factor'* can be used to specify the resolution ratio between two consecutive warping levels in the coarse-to-fine warping hierarchy. *'warp_zoom_factor'* must be selected from the open interval $(0, 1)$. For performance reasons, *'warp_zoom_factor'* is typically set to 0.5, i.e., the number of pixels is halved in each direction for each coarser warping level. Values for *'warp_zoom_factor'* close to 1 can lead to slightly better results. However, they require a disproportionately larger computation time.
- `GenParamName` = *'warp_levels'* can be used to restrict the warping hierarchy to a maximum number of levels. For *'warp_levels'* = 0, the largest possible number of levels is used. If the image size does not allow to use the specified number of levels (taking the resolution ratio *'warp_zoom_factor'* into account), the largest possible number of levels is used. Usually, *'warp_levels'* should be set to 0.
- `GenParamName` = *'warp_last_level'* can be used to specify the number of warping levels for which the flow increment should no longer be computed. Usually, *'warp_last_level'* is set to 1 or 2, i.e., a flow increment is computed for each warping level, or the finest warping level is skipped in the computation. In the latter case, the computation is performed on an image of half the resolution of the original image and then interpolated to the full resolution.
- `GenParamName` = *'outer_iter'* can be used to specify the number of outer iterations in the minimization scheme. Typically, the larger *'outer_iter'*, the more accurate the numerical results are. Higher values for this parameter lead to an increase in the computation time. Typically, *'outer_iter'* is set to values between 5 and 10.
- `GenParamName` = *'inner_iter'* can be used to specify the number of inner iterations in the minimization scheme. Typically, the larger *'inner_iter'*, the more accurate the numerical results are. Higher values for this parameter lead to an increase in the computation time. Usually, two inner iterations are sufficient.
- `GenParamName` = *'sor_iter'* can be used to specify the number of SOR iterations for solving the linear system of equations. Typically, the larger *'sor_iter'*, the more accurate the numerical results are. Higher values for this parameter lead to an increase in the computation time. Usually, three SOR iterations are sufficient.
- `GenParamName` = *'omega'* can be used to specify the relaxation factor ω of the SOR method. *'omega'* must be selected from the open interval $(1, 2)$. Typically, *'omega'* is set to 1.9.

Algorithm

The scene flow is estimated by minimizing a suitable energy functional:

$$E(f) = \int E_D(f) + E_S(f)\,\mathrm{d}r\mathrm{d}c$$

where $f = (u, v, dc)$ is the optical flow field and the disparity change. $E_D(f)$ denotes the data term and $E_S(f)$ the smoothness (regularization) term. The algorithm is based on the following assumptions, which lead to the data and smoothness terms:

Brightness Constancy It is assumed that the gray value of a point remains constant in all four input images, resulting in the following four constraints:

$$
\begin{aligned}
C_{D1} &= I_{1,t1}(r,c) - I_{2,t1}(r, c + d(r,c)) = 0 \\
C_{F1} &= I_{1,t2}(r + u(r,c), c + v(r,c)) - I_{1,t1}(r,c) = 0 \\
C_{F2} &= I_{2,t2}(r + u(r,c), c + v(r,c) + d(r,c) + dc(r,c)) - I_{2,t1}(r, c + d(r,c)) = 0 \\
C_{D2} &= I_{2,t2}(r + u(r,c), c + v(r,c) + d(r,c) + dc(r,c)) - I_{1,t2}(r + u(r,c), c + v(r,c)) = 0
\end{aligned}
$$

Here, $I_{1,t1}$, $I_{2,t1}$, $I_{1,t2}$, and $I_{2,t2}$ denote `ImageRect1T1`, `ImageRect2T1`, `ImageRect1T2`, and `ImageRect2T2`, respectively.

Piecewise smoothness of the scene flow The solution is assumed to be piecewise smooth. This smoothness is achieved by penalizing the first derivatives of the flow $\nabla(u)^2, \nabla(v)^2, \nabla(dc)^2$. The use of a statistically robust (linear) penalty function $\Psi(s) = \sqrt{s^2 + \varepsilon}$ with $\varepsilon = 0.001$ provides the desired preservation of edges in the movement in the scene flow to be determined.

Because the disparity image d is given, the first constraint C_{D1} can be omitted. Taking into account all of the above assumptions, the energy functional can be written as

$$
E(f) = \int \Psi(C_{F1}^2) + \Psi(C_{F2}^2) + \Psi(C_{D2}^2) + \Psi(\lambda(\nabla(u)^2 + \nabla(v)^2) + \gamma \nabla(dc)^2) \, \mathrm{d}r\mathrm{d}c
$$

where λ and γ are the regularization parameters passed in `SmoothingFlow` and `SmoothingDisparity`.

To calculate large displacements, coarse-to-fine warping strategies use two concepts that are closely interlocked: The successive refinement of the problem (coarse-to-fine) and the successive compensation of the current image pair by already computed displacements (warping). Algorithmically, such coarse-to-fine warping strategies can be described as follows:

1. First, all images are zoomed down to a very coarse resolution level.
2. Then, the scene flow is computed on this coarse resolution.
3. The scene flow is required on the next resolution level: It is applied there to the second image pair of the image sequence, i.e., the problem on the finer resolution level is compensated by the already computed scene flow. This step is also known as warping.
4. The modified problem (difference problem) is now solved on the finer resolution level, i.e., the scene scene flow is computed there.
5. The steps 3-4 are repeated until the finest resolution level is reached.
6. The final result is computed by adding up the scene flow from all resolution levels.

This incremental computation of the scene flow has the following advantage: While the coarse-to-fine strategy ensures that the displacements on the finest resolution level are very small, the warping strategy ensures that the displacements remain small for the incremental displacements (scene flow of the difference problems). Since small displacements can be computed much more accurately than larger displacements, the accuracy of the results typically increases significantly by using such a coarse-to-fine warping strategy. However, instead of having to solve a single correspondence problem, an entire hierarchy of these problems must be solved.

The minimization of functionals is mathematically very closely related to the minimization of functions: Like the fact that the zero crossing of the first derivative is a necessary condition for the minimum of a function, the fulfillment of the so called Euler-Lagrange equations is a necessary condition for the minimizing function of a functional (the minimizing function corresponds to the desired scene flow in this case). The Euler-Lagrange equations are partial differential equations. By discretizing these Euler-Lagrange equations using finite differences, large sparse nonlinear equation systems have to be solved in this algorithm.

For each warping level a single equation system must be solved. The algorithm uses an iteration scheme consisting of two nested iterations (called the outer and inner iteration) and the SOR (Successive Over-Relaxation) method. The outer loop contains the linearization of the nonlinear terms resulting from the data constraints. The nonlinearity of Ψ' is removed by the inner fixed point iteration scheme. The resulting linear system of equations can be solved efficiently by the SOR method.

Parameters

▷ **ImageRect1T1** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Input image 1 at time t_1.

▷ **ImageRect2T1** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Input image 2 at time t_1.

▷ **ImageRect1T2** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Input image 1 at time t_2.

▷ **ImageRect2T2** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Input image 2 at time t_2.

▷ **Disparity** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
Disparity between input images 1 and 2 at time t_1.

▷ **OpticalFlow** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* vector_field
Estimated optical flow.

▷ **DisparityChange** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
Estimated change in disparity.

▷ **SmoothingFlow** (input_control) . . . . . number ⇝ *real* / integer
Weight of the regularization term relative to the data term (derivatives of the optical flow).
Default: 40.0
Suggested values: SmoothingFlow ∈ {10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0}
Restriction: SmoothingFlow > 0.0

▷ **SmoothingDisparity** (input_control) . . . . . number ⇝ *real* / integer
Weight of the regularization term relative to the data term (derivatives of the disparity change).
Default: 40.0
Suggested values: SmoothingDisparity ∈ {10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0}
Restriction: SmoothingDisparity > 0.0

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Parameter name(s) for the algorithm.
Default: 'default_parameters'
Suggested values: GenParamName ∈ {'default_parameters', 'warp_levels', 'warp_zoom_factor', 'warp_last_level', 'outer_iter', 'inner_iter', 'sor_iter', 'omega'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Parameter value(s) for the algorithm.
Default: 'accurate'
Suggested values: GenParamValue ∈ {'very_accurate', 'accurate', 'fast', 'very_fast', 0, 1, 2, 3, 4, 5, 6, 0.5, 0.6, 0.7, 0.75, 3, 5, 7, 2, 3, 1.9}

Result

If the parameter values are correct, the operator scene_flow_uncalib returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

binocular_disparity, binocular_disparity_mg

Possible Successors

threshold, vector_field_length

Alternatives

scene_flow_calib, optical_flow_mg

References

A. Wedel, C. Rabe, T. Vaudrey, T. Brox, U. Franke and D. Cremers: "Efficient dense scene flow from sparse or dense stereo data"; In: Proceedings of the 10th European Conference on Computer Vision: Part I, pages 739-751. Springer-Verlag, 2008.

Module

Foundation

12.16 Smoothing

This chapter contains operators for smoothing filters. Further information about filtering can be found at the introduction to the chapter Filters.

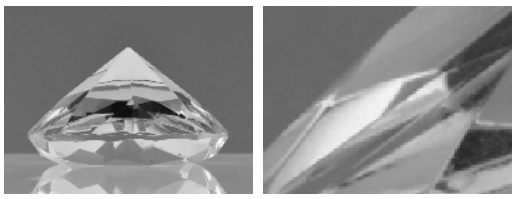
General information about smoothing filters

Smoothing operators are filters that help to suppress noise in an image. For this purpose it is assumed, that in the undisturbed or true image the gray value of a given data point does not completely differ from its surroundings, ideally even varies only little. Thus, to suppress noise, it can be useful to replace the measured gray value with an estimate based on surrounding data points. Such an estimate can be done in different ways, so HALCON provides different smoothing operators.

The operators differ in speed and suitability for different kinds of noise. Information like the complexity (runtime dependence on the image size) is, if available, given in the operator reference. While most operators treat a single image, some can process depending images (e.g., multichannel filters like `mean_n` and `rank_n`, or edge-preserving filters like `guided_filter` and `bilateral_filter`, which additionally use guidance images). Please note that some filters have both possibilities and more information is given in the specific operator reference.

Smoothing filters for single images with random noise

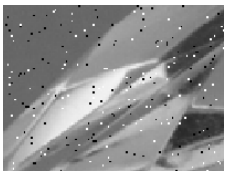
These smoothing filters apply their smoothing function on each channel of the input image separately and return a smoothed image with the same number of channels. In the following table we list implemented variants of smoothing filters for a single image with random noise and apply them for three different variants of random noise. The images in the table shall give an idea of the operators capability, but please note that the smoothed images highly depend on the input parameters and the individual image for every operator. For comparison, the different noisy images without filtering are given in the first row of the table. The undisturbed image without noise is shown in the following figure ((1) the full image as well as (2) its part by means of which possible effects on edges and remains from Salt & Pepper noise are visualized more clearly).



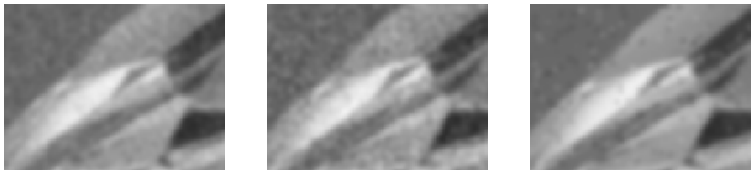
(1) (2)

(1) Undisturbed image, (2) part of the image chosen for the visualization of the filter capabilities

We marked filters recommended due to their special suitability concerning speed (S), edge-preservation (E), or a compromise between these two (C). The numbers in square brackets refer to further information that is given in a list below the table.

White Noise	Gaussian Noise	Salt & Pepper Noise	Time[1]	Alternatives
noisy image				
				

binomial_filter(S)



1|2

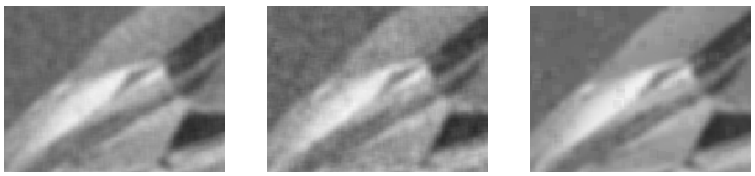
gauss_filter, smooth_image, derivate_gauss, isotropic_diffusion

smooth_image

2 | 9

binomial_filter, gauss_filter, mean_image, derivate_gauss, isotropic_diffusion

mean_image(S)



1 | 1

binomial_filter, gauss_filter, smooth_image, mean_image_shape

anisotropic_diffusion(E)[2]

805|2568

bilateral_filter, guided_filter

guided_filter(E)[2,3]

13 | 62

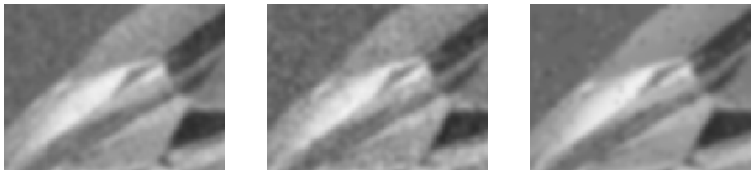
bilateral_filter, anisotropic_diffusion, median_image

bilateral_filter(E)[3]

16 | 54

guided_filter, anisotropic_diffusion, median_image

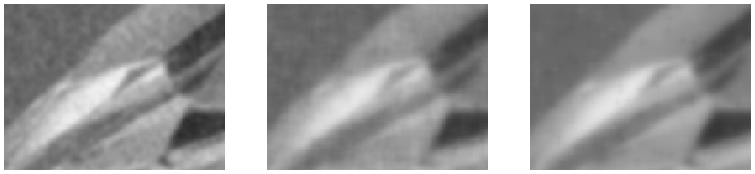
gauss_filter



1 | 4

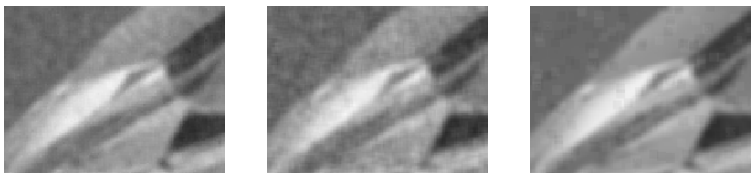
binomial_filter, smooth_image, derivate_gauss, isotropic_diffusion

isotropic_diffusion(E)[2]



11 | 51

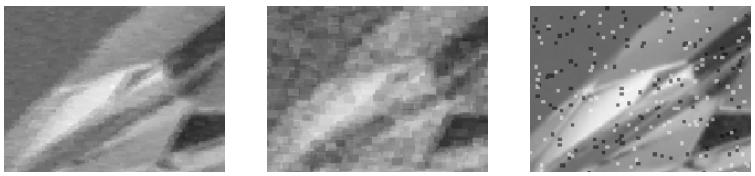
sigma_image



10 | 33

anisotropic_diffusion, rank_image

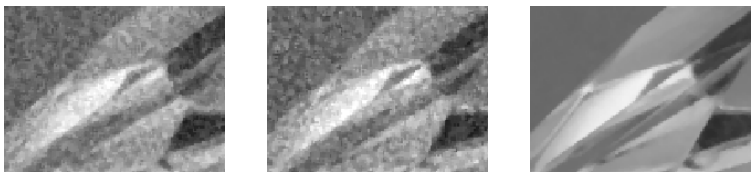
midrange_image



3 | 11

sigma_image

median_image(E)



3 | 4

median_rect, rank_image, rank_rect

median_rect(C)

2 | 3

median_image, rank_rect, rank_image

median_separate(C)

7 | 24 median_image

median_weighted

14 | 47 median_image, trimmed_mean, sigma_image

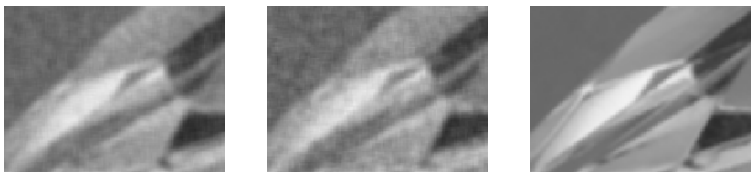
rank_rect(E)

2 | 8 rank_image, median_rect, median_image

rank_image(E)

3 | 15 rank_rect, median_image, median_rect

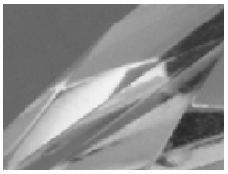
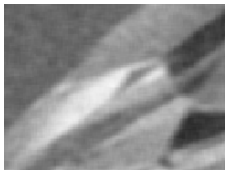
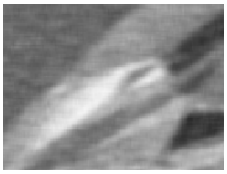
mean_sp



4 | 9 mean_image, median_image, median_separate, eliminate_min_max

eliminate_min_max

11 | 42

eliminate_sp				1 | 10	mean_sp, mean_image, median_image, eliminate_min_max
trimmed_mean				7 | 29	sigma_image, median_weighted, median_image

Further information related to the numbers in square brackets used in the table above:

1. The numbers in the column 'Time' are indications about the time the operator uses to process an image. The numbers are obtained from averaging over multiple runs on the three different noise images and given in arbitrary units. Note also that the runtime of an operator depends on many factors, not at least on the used parameters and the image size. For each filter, the first number is for an image of size 800x600, the second for an image of size 1497x1160. The shown images are parts of the smaller image with parameters reasonable to us (we do not claim to have found the parameters resulting neither necessarily in the best image smoothing nor the smallest runtime). Even if two filters have the same parameter name as input, we did not necessarily use the same parameter value for both of them. Therefore, these numbers are to be understood as an indicator only.
2. This operator can be used iteratively.
3. This operator uses a guidance image (which can be the Image itself).

Smoothing filters for single images with systematic noise

Video images composed of two half images can have systematic errors. In such a case, the operator fill_interlace can help.

Operators designed for smoothing over multiple channels

These smoothing filters take an image with multiple channels as input and return a single channel (gray value) image. In HALCON, the following filters of this group are implemented:

- mean_n
- rank_n

Further operators

In addition to the smoothing filters, this chapter contains the following operator: info_smooth, which returns information related to the different filters used by the operator smooth_image.

Glossary

In the following, the most important terms that are used in the context of smoothing filters are described:

smoothing Smoothing means to apply a filter function on the given data to capture the main data patterns while removing noise.

random noise Random noise is a stationary variation of brightness or color information by a small random amount for every pixel with an assumed mean of 0 over the total image.

systematic noise Systematic noise is predictable noise, caused, e.g., by the specific setup used to acquire the images.

anisotropic_diffusion (Image : ImageAniso : Mode, Contrast, Theta, Iterations :)

Perform an anisotropic diffusion of an image.

The operator anisotropic_diffusion performs an anisotropic diffusion on the input image Image according to the model of Perona and Malik. This procedure is also referred to as nonlinear isotropic diffusion. Considering the image as a gray value function u, the algorithm is a discretization of the partial differential equation

$$u_t = div(g(|\nabla u|^2, c)\nabla u)$$

with the initial value $u = u_0$ defined by Image at a time t_0. The equation is iterated Iterations times in time steps of length Theta, so that the output image ImageAniso contains the gray value function at the time $t_0 + \text{Iterations} \cdot \text{Theta}$.

The goal of the anisotropic diffusion is the elimination of image noise in constant image patches while preserving the edges in the image. The distinction between edges and constant patches is achieved using the threshold Contrast on the size of the gray value differences between adjacent pixels. Contrast is referred to as the contrast parameter and abbreviated with the letter c.

The variable diffusion coefficient g can be chosen to follow different monotonically decreasing functions with values between *0* and *1* and determines the response of the diffusion process to an edge. With the parameter Mode, the following functions can be selected:

$$g_1(x, c) = \frac{1}{\sqrt{1 + 2\frac{x}{c^2}}}$$

Choosing the function g_1 by setting Mode to *'parabolic'* guarantees that the associated differential equation is parabolic, so that a well-posedness theory exists for the problem and the procedure is stable for an arbitrary step size Theta. In this case however, there remains a slight diffusion even across edges of a height larger than c.

$$g_2(x, c) = \frac{1}{1 + \frac{x}{c^2}}$$

The choice of *'perona-malik'* for Mode, as used in the publication of Perona and Malik, does not possess the theoretical properties of g_1, but in practice it has proved to be sufficiently stable and is thus widely used. The theoretical instability results in a slight sharpening of strong edges.

$$g_3(x, c) = 1 - \exp(-C\frac{c^8}{x^4})$$

The function g_3 with the constant $C = 3.31488$, proposed by Weickert, and selectable by setting Mode to *'weickert'* is an improvement of g_2 with respect to edge sharpening. The transition between smoothing and sharpening happens very abruptly at $x = c^2$.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **ImageAniso** (output_object) ..... image(-array) ⇝ *object :* byte / uint2 / real
Output image.

▷ **Mode** (input_control) ..... string ⇝ *string*
Diffusion coefficient as a function of the edge amplitude.
Default: 'weickert'
List of values: `Mode` ∈ {'weickert', 'perona-malik', 'parabolic'}

▷ **Contrast** (input_control) ..... real ⇝ *real*
Contrast parameter.
Default: 5.0
Suggested values: `Contrast` ∈ {2.0, 5.0, 10.0, 20.0, 50.0, 100.0}
Restriction: Contrast > 0

▷ **Theta** (input_control) ..... real ⇝ *real*
Time step.
Default: 1.0
Suggested values: `Theta` ∈ {0.5, 1.0, 3.0}
Restriction: Theta > 0

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 10
Suggested values: `Iterations` ∈ {1, 3, 10, 100, 500}
Restriction: Iterations >= 1

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Alternatives

`bilateral_filter`, `guided_filter`

References

J. Weickert; "'Anisotropic Diffusion in Image Processing'; PhD Thesis; Fachbereich Mathematik, Universität Kaiserslautern; 1996.

P. Perona, J. Malik; "Scale-space and edge detection using anisotropic diffusion"; Transactions on Pattern Analysis and Machine Intelligence 12(7), pp. 629-639; IEEE; 1990.

G. Aubert, P. Kornprobst; "Mathematical Problems in Image Processing"; Applied Mathematical Sciences 147; Springer, New York; 2002.

Module

Foundation

```
bilateral_filter ( Image,
    ImageJoint :  ImageBilateral :  SigmaSpatial,  SigmaRange,
    GenParamName,  GenParamValue :  )
```

bilateral filtering of an image.

`bilateral_filter` performs a joint bilateral filtering on the input `Image` using the guidance image `ImageJoint` and returns the result in `ImageBilateral`. `Image` and `ImageJoint` must be of the same size and type.

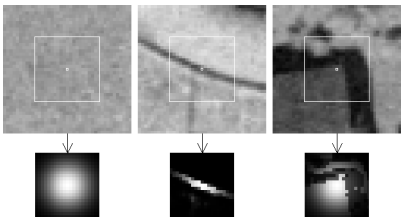
`SigmaSpatial` defines the size of the filter mask and corresponds to the standard deviation of a conventional Gauss filter. Bigger values increase the area of influence of the filter and less detail is preserved.

SigmaRange is used to modify the filter mask depending on the pixels of ImageJoint around the current pixel. Only pixels in areas with weak edges that have a contrast lower than SigmaRange contribute to the smoothing. Please note that the contrast in uint2 or real images may differ significantly from the default values of SigmaRange and adjust the parameter accordingly.

GenParamName and GenParamValue currently can be used to control the trade-off between accuracy and speed (see below).

Influence of the Joint Image

Each pixel of Image is filtered with a filter mask that depends on ImageJoint. The filter mask combines a Gaussian closeness function depending on SigmaSpatial and a Gaussian similarity function that weights gray value differences depending on SigmaRange.



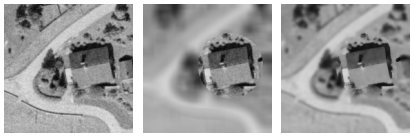
Examples for local filter masks depending on ImageJoint: Left: In homogeneous areas the filter mask is nearly Gaussian. Center: The filter mask follows the line. That means, only the dark pixels are smoothed and the edge is preserved. Right: The filter mask resembles the corner. Note that the filter mask extends across the shadow into areas with similar gray values.

If Image and ImageJoint are identical, bilateral_filter behaves like an edge-preserving smoothing where SigmaSpatial defines the size of the filter mask. Pixels at edges that have a contrast significantly greater than SigmaRange are preserved, while pixels in homogeneous areas are smoothed.

(1) (2) (3)

(1) Image and (2) ImageJoint are identical. That leads to edge-preserving smoothing in (3) ImageBilateral.

If Image and ImageJoint are different, each pixel of Image is smoothed with a filter mask that is influenced by ImageJoint. Pixels at positions where ImageJoint has strong edges with a contrast significantly greater than SigmaRange are smoothed less than pixels in homogeneous areas.



(1) (2) (3)

(1) Image and (2) ImageJoint are different. (3) ImageBilateral: Only edges are preserved where ImageJoint has edges.

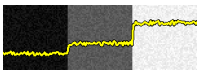
If ImageJoint is constant, bilateral_filter is equivalent to a Gaussian smoothing with SigmaSpatial (see gauss_filter or smooth_image).

(1) (2) (3)

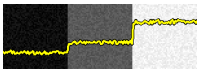
(1) Image. (2) ImageJoint is constant. (3) ImageBilateral is equal to Image after Gaussian smoothing.

Influence of the smoothing parameters

The following examples show the influence of SigmaRange on an artificial image. In this image, the noise level is 10 gray values, the left edge has a contrast of 50 gray values, the right edge has a contrast of 100 gray values. The yellow line shows a gray-value profile of a horizontal cross section.



Original image with overlaid gray profile, used as Image and ImageJoint.



Filter result with SigmaRange = 1: No effect because SigmaRange is below noise level. Therefore noise is treated as edge and preserved.

Filter result with SigmaRange = 25: Noise is smoothed, edges are preserved.

Filter result with SigmaRange = 50: The weaker edge is smoothed, the stronger edge is preserved.

Filter result with SigmaRange = 100: Both edges are smoothed.

Generic Parameters

The following values for GenParamName are supported:

'sampling_method' Per default, bilateral_filter uses an approximation of the bilateral filter that only uses a subset of sampled points for the calculation of the local filter masks.

By setting *'sampling_method'*, the used approximation can be selected. Possible values are:

'grid' **(default)** Uses a regular grid for subsampling the filter masks.

'poisson_disk' Uses a Poisson disk sampling. This method is slower than *'grid'*, but may produce less artifacts.

'exact' Uses all available points. This method is slowest, but the most accurate. If *'exact'* is used, *'sampling_ratio'* is ignored.

'sampling_ratio' Controls how many points are used for the subsampling of the local filter masks.

By setting *'sampling_ratio'* to *1.0*, the exact method is used. Using a lower sampling ratio leads to faster filtering, but also to slightly less accurate results.

Suggested values: 0.25, 0.5, 0.75, 1.0

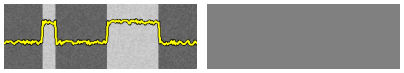
Default: 0.50

Rolling Bilateral Filter

bilateral_filter can be applied iteratively. In this case, the result of one iteration is used as guidance image for the next iteration. This can be useful, e.g., to remove small structures from the original image even if they have a high contrast.

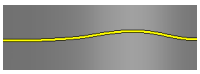
The following example shows the effect of a rolling filter on an artificial image. In this image, the noise level is 10 gray values, the contrast between dark and bright areas is 100 gray values, the left bright bar is 10 pixels wide, the right bar is 40 pixels wide. The yellow line shows a gray-value profile of a horizontal cross section. Used parameters: ImageJoint constant, SigmaSpatial = 25, SigmaRange = 15.

```
* Apply the rolling bilateral filter
* (use a constant guide for the first iteration).
gen_image_proto(Image, ImageJoint, 128)
for I := 1 to 6 by 1
   bilateral_filter(Image, ImageJoint, ImageJoint, 25, 15, [], [])
endfor
```



(1) (2)

(1) Input `Image` and (2) `ImageJoint` for first iteration of the rolling filter.



Result after first iteration: The smaller bar is removed.

Result after third iteration: The edges of the right bar are partly restored.

Result after sixth iteration: The edges of the right bar are restored completely.

Mathematical Background

The calculation of the filtered gray values I'_i is done on the basis of the following formula:

$$I'_i = \frac{1}{K_i} \sum_{k \in \omega_i} c_{ik} \cdot s_{ik} \cdot I_k$$

where c_{ik} is a closeness function

$$c_{ik} = exp\left(\frac{-\|i-k\|^2}{2 \cdot SigmaSpatial^2}\right)$$

s_{ik} is a similarity function

$$s_{ik} = exp\left(\frac{-(J_i - J_k)^2}{2 \cdot SigmaRange^2}\right)$$

K_i is a normalization factor

$$K_i = \sum_{k \in \omega_i} c_{ik} \cdot s_{ik}$$

where I'_i, I_i and J_i are the gray values of `Image` and `ImageJoint` at the pixel position i, and ω_i is the neighborhood around the pixel i.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Image to be filtered.
- ▷ **ImageJoint** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Joint image.
- ▷ **ImageBilateral** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Filtered output image.
- ▷ **SigmaSpatial** (input_control) . . . . . real ⇝ *real*
  Size of the Gaussian of the closeness function.
  Default: 3.0
  Suggested values: SigmaSpatial ∈ {1.0, 2.0, 3.0, 10.0}
  Restriction: SigmaSpatial > 0.6
- ▷ **SigmaRange** (input_control) . . . . . real ⇝ *real*
  Size of the Gaussian of the similarity function.
  Default: 20.0
  Suggested values: SigmaRange ∈ {3.0, 10.0, 20.0, 50.0, 100.0}
  Restriction: SigmaRange > 0.0001
- ▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
  Generic parameter name.
  Default: []
  List of values: GenParamName ∈ {'sampling_ratio', 'sampling_method'}
- ▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
  Generic parameter value.
  Default: []
  Suggested values: GenParamValue ∈ {'grid', 'poisson_disk', 'exact', 0.5, 0.25, 0.75, 1.0}

Example

```
read_image (Image, 'mreut')
* Edge-preserving smoothing
bilateral_filter (Image, Image, ImageBilateral, 5, 20, [], [])
* Rolling filter (5 iterations)
gen_image_proto (Image, ImageJoint, 0)
for I := 1 to 5 by 1
  bilateral_filter (Image, ImageJoint, ImageJoint, 5, 20, [], [])
endfor
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image

Possible Successors

threshold, dyn_threshold, var_threshold, regiongrowing

Alternatives

guided_filter, anisotropic_diffusion, median_image

References

C. Tomasi, R. Manduchi: "Bilateral filtering for gray and color images"; Sixth International Conference in Computer Vision; S. 839-846; January 1998.

F. Banterle, M. Corsini, P. Cignoni, R. Scopigno: "A Low-Memory, Straightforward and Fast Bilateral Filter Through Subsampling in Spatial Domain"; Computer Graphics Forum, no. 1, vol 31; S. 19-23; February 2012.

G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, K. Toyama: "Digital Photography with Flash and No-flash Image Pairs"; ACM Trans., no. 3, vol. 23; S. 9; August 2004.

R. Bridson: "Fast Poisson Disk Sampling in Arbitrary Dimensions"; ACM SIGGRAPH 2007 Sketches, no. 22; 2007.

Module

Foundation

binomial_filter (Image : ImageBinomial : MaskWidth, MaskHeight :)

Smooth an image using the binomial filter.

binomial_filter smooths the image Image using a binomial filter with a mask size of MaskWidth × MaskHeight pixels and returns the smoothed image in ImageBinomial. The binomial filter is a very good approximation of a Gaussian filter that can be implemented extremely efficiently using only integer operations. Hence, binomial_filter is very fast. Let m = MaskHeight and n = MaskWidth. Then, the filter coefficients b_{ij} are given by binomial coefficients

$$\binom{l}{k} = \frac{l!}{k!(l-k)!}$$

as follows:

$$b_{ij} = \frac{1}{2^{n+m-2}} \binom{m-1}{i} \binom{n-1}{j}$$

Here, $i = 0, \ldots, m-1$ and $j = 0, \ldots, n-1$. The binomial filter performs approximately the same smoothing as a Gaussian filter with $\sigma = \sqrt{n-1}/2$, where for simplicity it is assumed that $m = n$. In detail, the relationship between n and σ is:

n	σ
3	0.7523
5	1.0317
7	1.2505
9	1.4365
11	1.6010
13	1.7502
15	1.8876
17	2.0157
19	2.1361
21	2.2501
23	2.3586
25	2.4623
27	2.5618
29	2.6576
31	2.7500
33	2.8395
35	2.9262
37	3.0104

If different values are chosen for MaskHeight and MaskWidth, the above relation between n and σ still holds and refers to the amount of smoothing in the row and column directions.

binomial_filter can be executed on OpenCL devices for all supported image types.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **ImageBinomial** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Smoothed image.

▷ **MaskWidth** (input_control) ..... integer ⇝ *integer*
Filter width.
Default: 5
List of values: MaskWidth ∈ {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37}

▷ **MaskHeight** (input_control) ..... integer ⇝ *integer*
Filter height.
Default: 5
List of values: MaskHeight ∈ {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37}

Result

If the parameter values are correct the operator binomial_filter returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image, grab_image

Possible Successors

regiongrowing, threshold, sub_image, dyn_threshold, auto_threshold

Alternatives

gauss_filter, smooth_image, derivate_gauss, isotropic_diffusion

See also

mean_image, anisotropic_diffusion, sigma_image, gen_lowpass

Module

Foundation

eliminate_min_max (Image : FilteredImage : MaskWidth, MaskHeight, Gap, Mode :)

Smooth an image in the spatial domain to suppress noise.

eliminate_min_max smooths an image by replacing gray values with neighboring mean values, or local minima/maxima. In order to prevent edges and lines from being smoothed, only those gray values that represent local minima or maxima are replaced (if there is a line or edge within an image there will be at least one neighboring pixel with a comparable gray value). Gap controls the strictness of replacement: Only gray values that exceed all other values within their local neighborhood more than Gap and all values that fall below their neighboring more than Gap are replaced.

Thus, with $g(x, y)$ being the gray value at position (x, y) and $E(x, y)$ representing the gray values of a MaskWidth × MaskHeight sized rectangular neighborhood of a pixel at position (x, y), containing all pixels within the neighborhood except the pixel itself, a pixel is replaced

- if $g(x,y) \geq$ Gap $+ max(E(x,y))$, or
- else if $g(x,y) +$ Gap $<= min(E(x,y))$.
- Elsewise $g(x,y)$ is adopted without change.

Mode specifies how to perform the new value in case of a replacement.

- Mode = *1*: replace a local maximum with next minor local maximum and replace a local minimum with next bigger local minimum.
- Mode = *2*: replace with mean value of all pixels within the local neighborhood (including the replaced pixel).
- Mode = *3*: replace with median value of all pixels within the local neighborhood (including the replaced pixel (also used if Mode has got any other value than *1* or *2*).

MaskWidth and MaskHeight specify the width and height of the rectangular neighborhood. Border treatment: Pixels outside the image border are not considered (e.g., with a local 3×3-mask the neighborhood of a pixel at $(0,0)$ reduces to the pixels at $(1,0)$, $(0,1)$, and $(1,1)$).

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

If MaskWidth or MaskHeight is an even number, it is replaced by the next higher odd number (this allows the unique extraction of the center of the filter mask). Width/height of the mask may not exceed the image width/height.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image to smooth.

▷ **FilteredImage** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Smoothed image.

▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *integer*
Width of filter mask.
Default: 3
Suggested values: MaskWidth ∈ {3, 5, 7, 9}
Value range: 3 ≤ MaskWidth ≤ width(Image)
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth)

▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *integer*
Height of filter mask.
Default: 3
Suggested values: MaskHeight ∈ {3, 5, 7, 9}
Value range: 3 ≤ MaskHeight ≤ width(Image)
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth)

▷ **Gap** (input_control) . . . . . number ⇝ *real*
Gap between local maximum/minimum and all other gray values of the neighborhood.
Default: 1.0
Suggested values: Gap ∈ {1.0, 2.0, 5.0, 10.0}

▷ **Mode** (input_control) . . . . . integer ⇝ *integer*
Replacement rule.
Default: 3
List of values: Mode ∈ {1, 2, 3}

Result

eliminate_min_max returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty eliminate_min_max returns with an error message.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

wiener_filter, wiener_filter_ni

See also

mean_sp, mean_image, median_image, median_weighted, binomial_filter, gauss_filter, smooth_image

References

M. Imme:"A Noise Peak Elimination Filter"; S. 204-211 in CVGIP Graphical Models and Image Processing, Vol. 53, No. 2, March 1991

M. Lückenhaus:"Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse"; Diplomarbeit; Technische Universität München, Institut für Informatik; Lehrstuhl Prof. Radig; 1995.

Module

Foundation

eliminate_sp (Image : ImageFillSP : MaskWidth, MaskHeight, MinThresh, MaxThresh :)

Replace values outside of thresholds with average value.

The operator eliminate_sp replaces all gray values outside the indicated gray value intervals (MinThresh to MaxThresh) with the neighboring mean values. Only those neighboring pixels which also fall within the gray value interval are used for averaging. If no such pixel is present in the vicinity the original gray value is used. The gray values in the input image falling within the gray value interval are also adopted without change.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

If even values instead of odd values are given for MaskHeight or MaskWidth, the routine uses the next larger odd values instead (this way the center of the filter mask is always explicitly determined).

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Input image.

▷ **ImageFillSP** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Smoothed image.

▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *integer*
Width of filter mask.
Default: 3
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11}
Value range: 3 ≤ MaskWidth (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth) && MaskWidth < width(Image) * 2

▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *integer*
Height of filter mask.
Default: 3
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11}
Value range: 3 ≤ MaskHeight (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskHeight) && MaskHeight < height(Image) * 2

▷ **MinThresh** (input_control) . . . . . integer ⇝ *integer*
Minimum gray value.
Default: 1
Suggested values: MinThresh ∈ {1, 5, 7, 9, 11, 15, 23, 31, 43, 61, 101}

▷ **MaxThresh** (input_control) . . . . . integer ⇝ *integer*
Maximum gray value.
Default: 254
Suggested values: MaxThresh ∈ {5, 7, 9, 11, 15, 23, 31, 43, 61, 101, 200, 230, 250, 254}
Restriction: MinThresh <= MaxThresh

Example

```
read_image(Image,'mreut')
dev_display(Image)
eliminate_sp(Image,ImageFillSP,3,3,101,201)
dev_display(ImageFillSP)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

disp_image

Alternatives

mean_sp, mean_image, median_image, eliminate_min_max

See also

binomial_filter, gauss_filter, smooth_image, anisotropic_diffusion, sigma_image, eliminate_min_max

Module

Foundation

fill_interlace (ImageCamera : ImageFilled : Mode :)

Interpolate 2 video half images.

The operator fill_interlace calculates an interpolated full image or removes odd/even lines from a video image composed of two half images. If an image is recorded with a video camera it consists of two half images recorded at different times but stored in one image in the digital form. This can lead to several errors in further processing. In order to reduce these errors the video image is modified. Every second line is re-calculated or removed. The parameter Mode determines whether this must be done for even (*'even'*, *'rmeven'*) or odd (*'odd'*, *'rmodd'*) line numbers. If you choose *'even'* or *'odd'* the gray values in the generated lines are calculated as mean values from the direct neighbors above or below the current pixel, respectively. If you choose *'rmeven'* or *'rmodd'* the even or odd lines numbers are removed (in that case the resulting image is only half as high as the input image). The value *'switch'* for Mode cause the odd and even lines to be exchanged.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **ImageCamera** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Gray image consisting of two half images.

▷ **ImageFilled** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Full image with interpolated/removed lines.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Instruction whether even or odd lines should be replaced/removed.
Default: 'odd'
List of values: `Mode` ∈ {'odd', 'even', 'rmodd', 'rmeven', 'switch'}

Example

```
read_image(Image,'video_image')
fill_interlace(Image,New,'odd')
sobel_amp(New,Sobel,'sum_abs',3)
```

Complexity

For each pixel: $O(2)$.

Result

If the parameter values are correct the operator `fill_interlace` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system ('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

`read_image`, `grab_image`

Possible Successors

`sobel_amp`, `edges_image`, `regiongrowing`, `diff_of_gauss`, `threshold`, `dyn_threshold`, `auto_threshold`, `mean_image`, `binomial_filter`, `gauss_filter`, `anisotropic_diffusion`, `sigma_image`, `median_image`

See also

`median_image`, `binomial_filter`, `gauss_filter`, `crop_part`

Module

Foundation

gauss_filter (Image : ImageGauss : Size :)

Smooth using discrete Gauss functions.

The operator `gauss_filter` smoothes images using the discrete Gaussian, a discrete approximation of the Gaussian function,

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-x^2}{2\sigma^2}\right)$$

The smoothing effect increases with increasing filter size. The following filter sizes (`Size`) are supported (the sigma value of the Gauss function is indicated in brackets):

- *3* (0.600)
- *5* (1.075)
- *7* (1.550)
- *9* (2.025)
- *11* (2.550)

For border treatment the gray values of the images are reflected at the image borders. Notice that, contrary to the operator `gauss_image`, the relationship between the filter mask size and its respective value for the sigma parameter is linear.

The operator `binomial_filter` can be used as an alternative to `gauss_filter`. `binomial_filter` is significantly faster than `gauss_filter`. It should be noted that the mask size in `binomial_filter` does not lead to the same amount of smoothing as the mask size in `gauss_filter`. Corresponding mask sizes can be determined based on the respective values of the Gaussian smoothing parameter sigma.

`gauss_filter` can be executed on OpenCL devices for all supported image types. However, the OpenCL implementation can produce slightly different results from the scalar implementation.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

In order to be able to process `gauss_filter` on an OpenCL device, `Image` must be at least 64 pixels in both width and height.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **`Image`** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Image to be smoothed.

▷ **`ImageGauss`** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Filtered image.

▷ **`Size`** (input_control) . . . . . integer ⇝ *integer*
Required filter size.
Default: 5
List of values: `Size` ∈ {3, 5, 7, 9, 11}

Example

```
gauss_filter(Input,Gauss,7)
regiongrowing(Gauss,Segments,7,7,5,100)
```

Complexity

For each pixel: $O(Size * 2)$.

Result

If the parameter values are correct the operator `gauss_filter` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system ('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image, grab_image

Possible Successors

regiongrowing, threshold, sub_image, dyn_threshold, auto_threshold

Alternatives

binomial_filter, smooth_image, derivate_gauss, isotropic_diffusion

See also

mean_image, anisotropic_diffusion, sigma_image, gen_lowpass

Module

Foundation

guided_filter (Image, ImageGuide : ImageGuided : Radius, Amplitude :)

Guided filtering of an image.

guided_filter filters the input Image using the guidance image ImageGuide and returns the result in ImageGuided. Image and ImageGuide must be of the same size and type.

The Radius is the size of the filter mask. Bigger values increase the area of influence of the filter and less detail is preserved. The value of Radius does not influence the runtime of the operator.

Amplitude is used to decide what is an edge and what is a homogeneous area. Bigger values of Amplitude lead to stronger edges being smoothed. As a rule of thumb, Amplitude should be lower than the contrast of the edges that should be preserved. Please note that the contrast in uint2 or real images may differ significantly from the default values of Amplitude and adjust the parameter accordingly.

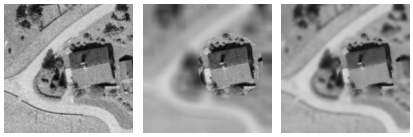
Influence of the Guidance Image

If Image and ImageGuide are identical, guided_filter behaves like an edge-preserving smoothing with a filter mask with Radius. Pixels at edges that have a contrast significantly greater than Amplitude are preserved, while pixels in homogeneous areas are smoothed. Hence, guided_filter is a fast alternative to anisotropic_diffusion or bilateral_filter.

(1) (2) (3)

(1) Image and (2) ImageGuide are identical. That leads to edge-preserving smoothing in (3) ImageGuided.

If Image and ImageGuide are different, Image is smoothed with a filter mask with Radius, except in areas where ImageGuide has edges with a contrast significantly greater than Amplitude.



(1) (2) (3)

(1) Image and (2) ImageGuide are different. (3) ImageGuided: Only edges are preserved where ImageGuide has edges.

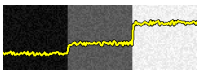
If ImageGuide is constant, guided_filter is equivalent to 2 consecutive calls of mean_image with mask size 2*Radius+1.

(1) (2) (3)

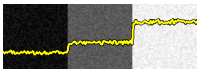
(2) ImageGuide is constant. This is equivalent to a double smoothing of (1) Image with mean_image. (3) ImageGuided

Influence of the smoothing parameters

The following examples show the influence of Amplitude on an artificial image. In this image, the noise level is 10 gray values, the left edge has a contrast of 50 gray values, the right edge has a contrast of 100 gray values. The yellow line shows a gray-value profile of a horizontal cross section.



Original image with overlaid gray profile, used as Image and ImageGuide.



Filter result with Amplitude = 1: No effect because Amplitude is below noise level. Therefore noise is treated as edge and preserved.

Filter result with Amplitude = 25: Noise is smoothed, edges are preserved.

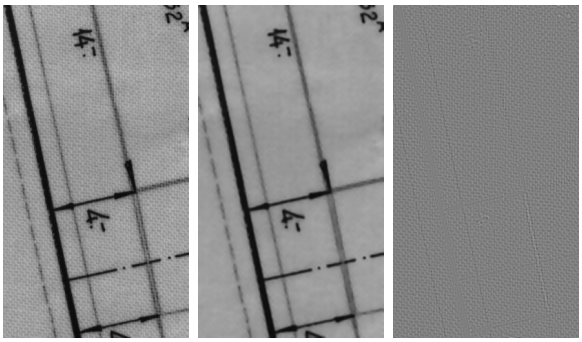
Filter result with Amplitude = 50: The weaker edge is smoothed, the stronger edge is preserved.

Filter result with Amplitude = 100: Both edges are smoothed.

Rolling Guided Filter

guided_filter can be applied iteratively. In this case, the result of one iteration is used as guidance image for the next iteration. This can be useful, e.g., to remove small structures from the original image even if they have a high contrast.

In the following example, the rolling guided filter is used to separate the texture from the original image.



(1) (2) (3)

Texture removal with the rolling guided filter: (1) Original, (2) separated structure, (3) separated texture.

```
* Apply the rolling guided filter
* (use a constant guide for the first iteration).
gen_image_proto(Image, ImageStructure, 0)
for I := 1 to 4 by 1
    guided_filter(Image, ImageStructure, ImageStructure, 1.5, 10)
endfor
* Separate texture by subtracting large structures from the
original.
sub_image(Image, ImageStructure, ImageTexture, 1, 128)
```

Since `guided_filter` with a constant `ImageGuide` is similar to `mean_image`, the first iteration could be replaced by a call of `mean_image` (or a similar smoothing filter), which is faster.

Mathematical Background

The calculation of the filtered gray value I'_i at the position i is done according to the following formula:

$$I'_i = \overline{a_i} G_i + \overline{b_i}$$

where

$$a_i = \frac{\frac{1}{|\omega_i|} \sum_{k \in \omega_i} G_k I_k - \overline{G_i} \overline{I_i}}{\sigma^2_{G_i} + Amplitude^2}$$

and

$$b_i = \overline{I_i} - a_i \overline{G_i}$$

where I_i and G_i are the gray values of `Image` and `ImageGuide` at the pixel position i, ω_i is the neighborhood with radius `Radius` around the pixel i, $\overline{a_i}$, $\overline{b_i}$, $\overline{I_i}$, and $\overline{G_i}$ are the mean of all a, b, I, or G in ω_i, σ_{G_i} is the standard deviation of all gray values of G in ω_i, and $|\omega_i|$ is the number of pixels in ω_i.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **ImageGuide** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Guidance image.
- ▷ **ImageGuided** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **Radius** (input_control) ..... integer ⇝ *integer*
  Radius of the filtering operation.
  Default: 3
  Suggested values: `Radius` ∈ {1, 2, 3, 5, 10}
  Restriction: Radius > 0
- ▷ **Amplitude** (input_control) ..... real ⇝ *real*
  Controls the influence of edges on the smoothing.
  Default: 20.0
  Suggested values: `Amplitude` ∈ {3.0, 10.0, 20.0, 50.0, 100.0}
  Restriction: Amplitude > 0

Example

```
read_image (Image, 'mreut')
* Edge-preserving smoothing
guided_filter (Image, Image, ImageGuided, 5, 20)
```

```
* Rolling filter (5 iterations)
gen_image_proto (Image, ImageGuide, 0)
for I := 1 to 5 by 1
  guided_filter (Image, ImageGuide, ImageGuide, 5, 20)
endfor
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

`read_image`

Possible Successors

`threshold`, `dyn_threshold`, `regiongrowing`

Alternatives

`bilateral_filter`, `anisotropic_diffusion`, `median_image`

References

Kaiming He, Jian Sun, Xiaoou Tang: "Guided Image Filtering"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-35, no. 6; S. 1397-1409; 2013.

Module

Foundation

info_smooth (: : Filter, Alpha : Size, Coeffs)

Information on smoothing filter `smooth_image`.

The operator `info_smooth` returns an estimation of the width of the smoothing filters used in routine `smooth_image`. For this purpose the underlying continuous impulse answers of `Filter` are scanned until a filter coefficient is smaller than five percent of the maximum coefficient (at the origin). `Alpha` is the filter parameter (see `smooth_image`). Currently four filters are supported (parameter `Filter`):

'deriche1', 'deriche2', 'shen' and 'gauss'.

The gauss filter was conventionally implemented with filter masks (the other three are recursive filters). In the case of the gauss filter the filter coefficients (of the one-dimensional impulse answer $f(n)$ with $n \geq 0$) are returned in `Coeffs` in addition to the filter size.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Filter** (input_control) . . . . . string ⇝ *string*
Name of required filter.
Default: 'deriche2'
List of values: `Filter` ∈ {'deriche1', 'deriche2', 'shen', 'gauss'}

▷ **Alpha** (input_control) . . . . . real ⇝ *real*
Filter parameter: small values effect strong smoothing (reversed in case of 'gauss').
Default: 0.5
Suggested values: `Alpha` ∈ {0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0, 5.0, 7.0, 10.0}
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Alpha > 0.0

▷ **Size** (output_control) . . . . . integer ⇝ *integer*
Width of filter is approx. size x size pixels.

▷ **Coeffs** (output_control) . . . . . integer-array ⇝ *integer*
In case of gauss filter: coefficients of the "positive" half of the 1D impulse answer.

Example

```
info_smooth('deriche2',0.5,Size,Coeffs)
smooth_image(Input,Smooth,'deriche2',7)
```

Result

If the parameter values are correct the operator `info_smooth` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_image`

Possible Successors

`smooth_image`

See also

`smooth_image`

Module

Foundation

isotropic_diffusion (Image : SmoothedImage : Sigma, Iterations :)

Perform an isotropic diffusion of an image.

The operator `isotropic_diffusion` performs an isotropic diffusion of the input image `Image`. This corresponds to a convolution of the image matrix with a Gaussian mask of standard deviation `Sigma`. If the parameter `Iterations` is set to *0*, such a convolution is performed explicitly. For input images with a full ROI, `isotropic_diffusion` returns the same results as the operator `derivate_gauss` when choosing *'none'* for its parameter `Component`. If the gray value matrix is larger than the ROI of `Image` the two operators differ since `derivate_gauss` takes the gray values outside of the ROI into account, while `isotropic_diffusion` mirrors the values at the boundary of the ROI in any case. The computational complexity increases linearly with the value of `Sigma`.

If `Iterations` has a positive value the smoothing process is considered as an application of the heat equation

$$u_t = \Delta u$$

on the gray value function u with the initial value $u = u_0$ defined by the gray values of `Image` at a time t_0. This equation is then solved up to a time $t_0 + \frac{1}{2}\texttt{Sigma}^2$, which is equivalent to the above convolution, using an iterative procedure for parabolic partial differential equations. The computational complexity is proportional to the value of `Iterations` and independent of `Sigma` in this case. For small values of `Iterations`, the computational accuracy is very low, however. For this reason, choosing `Iterations` < 3 is not recommended.

For smaller values of `Sigma`, the convolution implementation is typically the faster method. Since the runtime of the partial differential equation solver only depends on the number of iterations and not on the value of `Sigma`, it is typically faster for large values of `Sigma` if few iterations are chosen (e.g., `Iterations` $= 3$).

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **SmoothedImage** (output_object) ..... image(-array) ⇝ *object :* byte / uint2 / real
Output image.

▷ **Sigma** (input_control) ..... real ⇝ *real*
Standard deviation of the Gauss distribution.
Default: 1.0
Suggested values: Sigma ∈ {0.1, 0.5, 1.0, 3.0, 10.0, 20.0, 50.0}
Restriction: Sigma > 0

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 10
Suggested values: Iterations ∈ {0, 3, 10, 100, 500}
Restriction: Iterations >= 0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Module

Foundation

mean_image (Image : ImageMean : MaskWidth, MaskHeight :)

Smooth by averaging.

The operator mean_image carries out a linear smoothing with the gray values of all input images (Image). The filter matrix consists of ones (evaluated equally) and has the size MaskHeight × MaskWidth. The result of the convolution is divided by MaskHeight × MaskWidth. For border treatment the gray values are reflected at the image edges.

For mean_image special optimizations are implemented that use SIMD technology. The actual application of these special optimizations is controlled by the system parameter *'mmx_enable'* (see set_system). If *'mmx_enable'* is set to *'true'* (and the SIMD instruction set is available), the internal calculations are performed using SIMD technology. Note that SIMD technology performs best on large, compact input regions. Depending on the input region and the capabilities of the hardware the execution of mean_image might even take significantly more time with SIMD technology than without.

At any rate, it is advantageous for the performance of mean_image to choose the input region of Image such that any border treatment is avoided.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

If even values instead of odd values are given for MaskHeight or MaskWidth, the routine uses the next larger odd values instead (this way the center of the filter mask is always explicitly determined).

The mean filter value on real images is calculated internally using single precision floating point. This can lead to overflows (and thus incorrect results) if the full dynamic range is used.

mean_image can be executed on OpenCL devices for byte, int2, uint2, int4 and real images if MaskHeight is less than twice the height of Image. For OpenCL, the mean filter value is calculated internally using either 32 bit signed integers (for all integer image types) or single precision floating point (for real images). This can

lead to overflows (and thus incorrect results) if Image is either an int4 or real image and the full dynamic range is used. Additionally, to improve performance a full scan of each row of Image is calculated (again using either 32 bit integer or single precision floating point arithmetic) if MaskWidth is bigger than 9. This can also lead to overflows with very wide images even for byte, int2, or uint2 images. In these cases, the CPU version of mean_image should be used.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / int8 / real / vector_field

Image to be smoothed.

▷ **ImageMean** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / int8 / real / vector_field

Smoothed image.

▷ **MaskWidth** (input_control) ..... extent.x ⇝ *integer*

Width of filter mask.
Default: 9
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 15, 23, 31, 43, 61, 101}
Value range: 1 ≤ MaskWidth
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth) && MaskWidth < width(Image) * 2

▷ **MaskHeight** (input_control) ..... extent.y ⇝ *integer*

Height of filter mask.
Default: 9
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 15, 23, 31, 43, 61, 101}
Value range: 1 ≤ MaskHeight
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskHeight) && MaskHeight < height(Image) * 2

Example

```
read_image(Image,'fabrik')
mean_image(Image,Mean,3,3)
dev_display(Mean)
```

Complexity

For each pixel: O(15).

Result

If the parameter values are correct the operator mean_image returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

reduce_domain, rectangle1_domain

Possible Successors

dyn_threshold, regiongrowing

Alternatives

binomial_filter, gauss_filter, smooth_image, mean_image_shape

See also

anisotropic_diffusion, sigma_image, convol_image, gen_lowpass

Module

Foundation

mean_image_shape (Image, Mask : ImageMean : :)

Smooth image using a mean filter with arbitrary mask.

mean_image_shape performs a mean filter operation on the input image Image with a mask that is specified by the region Mask and returns the filtered image in ImageMean. The shape of the mask can be chosen arbitrarily and can, for example, be created with operators like gen_circle or draw_region. The position of the mask does not influence the result since the center of gravity of the mask region is used as the reference point of the mask. For border treatment the gray values are reflected at the image edges.

For mean_image_shape special optimizations are implemented that use SIMD technology. The actual application of these special optimizations is controlled by the system parameter *'avx2_enable'* (see set_system). If *'avx2_enable'* is set to *'true'* (and the SIMD instruction set is available), the internal calculations are performed using SIMD technology. Furthermore, these optimizations are only used for byte images and if the area of the filter mask provided by Mask is smaller than 129.

Note that mean_image_shape performs best on compact input regions.

At any rate, it is advantageous for the performance of mean_image_shape to choose the input region of Image such that any border treatment is avoided.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Image to be filtered.
- ▷ **Mask** (input_object) ..... region ⇝ *object*
  Filter mask.
- ▷ **ImageMean** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Filtered image.

Example

```
read_image (Image, 'fabrik')
gen_circle (Circle, 10, 10, 2)
mean_image_shape (Image, Circle, ImageMean)
dev_display (ImageMean)
```

Result

If the parameter values are correct the operator mean_image_shape returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

- Automatically parallelized on internal data level.

Possible Predecessors

read_image, reduce_domain, rectangle1_domain, gen_circle

Possible Successors

dyn_threshold, regiongrowing

Alternatives

binomial_filter, gauss_filter, smooth_image, mean_image

See also

anisotropic_diffusion, sigma_image, convol_image, gen_lowpass

Module

Foundation

mean_n (Image : ImageMean : :)

Average gray values over several channels.

The operator mean_n generates the pixel-by-pixel mean value of all channels . For each coordinate point the sum of all gray values at this coordinate is calculated. The result is the mean of the gray values (sum divided by the number of channels). The output image has one channel.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Multichannel gray image.

▷ **ImageMean** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Result of averaging.

Example

```
compose3(Channel1,Channel2,Channel3,&MultiChannel);
mean_n(MultiChannel,&Mean);
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

compose2, compose3, compose4, compose5, add_channels

Alternatives

rank_n

See also

count_channels, mean_image

Module

Foundation

mean_sp (Image : ImageSPMean : MaskWidth, MaskHeight, MinThresh, MaxThresh :)

Suppress salt and pepper noise.

The operator mean_sp carries out a smoothing by averaging the values. Only the gray values within the interval from MinThresh to MaxThresh are averaged. Gray values which are too light or too dark are ignored during summation. If no gray value lies within the default interval during summation the original gray value is adopted. If the thresholds are set at *0* or *255*, respectively, the operator mean_sp behaves like mean_image except for the running time.

The operator mean_sp is used to suppress extreme gray values (salt and pepper noise = white and black dots).

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

If even values instead of odd values are given for MaskHeight or MaskWidth, the routine uses the next larger odd values instead (this way the center of the filter mask is always explicitly determined).

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
  Input image.
- ▷ **ImageSPMean** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
  Smoothed image.
- ▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *integer*
  Width of filter mask.
  Default: 3
  Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11}
  (lin)
  Minimum increment: 2
  Recommended increment: 2
  Restriction: odd(MaskWidth) && MaskWidth < width(Image) * 2
- ▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *integer*
  Height of filter mask.
  Default: 3
  Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11}
  (lin)
  Minimum increment: 2
  Recommended increment: 2
  Restriction: odd(MaskHeight) && MaskHeight < height(Image) * 2
- ▷ **MinThresh** (input_control) . . . . . integer ⇝ *integer*
  Minimum gray value.
  Default: 1
  Suggested values: MinThresh ∈ {1, 5, 7, 9, 11, 15, 23, 31, 43, 61, 101}
  Value range: 0 ≤ MinThresh
- ▷ **MaxThresh** (input_control) . . . . . integer ⇝ *integer*
  Maximum gray value.
  Default: 254
  Suggested values: MaxThresh ∈ {5, 7, 9, 11, 15, 23, 31, 43, 61, 101, 200, 230, 250, 254}
  Restriction: MinThresh <= MaxThresh

Example

```
read_image(Image,'mreut')
dev_display(Image)
mean_sp(Image,ImageMeansp,3,3,101,201)
dev_display(ImageMeansp)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

disp_image

Alternatives

mean_image, median_image, median_separate, eliminate_min_max

See also

anisotropic_diffusion, sigma_image, binomial_filter, gauss_filter, smooth_image, eliminate_min_max

Module

Foundation

median_image (Image : ImageMedian : MaskType, Radius, Margin :)

Compute a median filter with various masks.

median_image performs a median filter on the input image Image with a square or circular mask and returns the filtered image in ImageMedian. The shape of the mask can be selected with MaskType. The radius of the mask can be selected with Radius.

Conceptually, the median filter sorts all gray values within the mask in ascending order and then selects the median of the gray values. The median is the "middle" one of the sorted gray values, i.e., the gray value with rank (position) $(N-1)/2+1$ of the sorted gray values, where N denotes the number of pixels covered by the filter mask. Here, the rank 1 corresponds to the smallest gray value and the rank N corresponds to the largest gray value within the mask (see also rank_image).

The filter mask is determined by Radius, defining the size, and MaskType, defining the shape of the mask. For latter one, the following options are available:

'circle' The mask consists of the pixel within a circle with Radius around the pixel of the mask center.

'square' The mask consists of the pixel within a square with an edge length of $2 \cdot$Radius $+1$ pixel.

median_image can be used, for example, to smooth images, to suppress unwanted objects (e.g., point-like or line-like structures) that are smaller than the mask, and can therefore be used to estimate the background illumination for a shading correction or as a preprocessing step for the dynamic threshold operation (see dyn_threshold).

Several border treatments can be chosen for filtering via the parameter Margin:

gray value Pixels outside of the image border are assumed to be constant (with the indicated gray value).

'continued' Continuation of the gray values at the image border.

'cyclic' Cyclic continuation at the image borders.

'mirrored' Reflection of pixels at the image borders.

When using the MaskType *'square'* with Radius 1 or 2 (resulting in a 3x3 or 5x5 pixel filter mask) and the border treatment 'mirrored', median_image can be executed on OpenCL devices.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

median_image uses an algorithm with a runtime per pixel that depends on the mask height $2 \cdot$ Radius $+1$. Therefore, median_image is slower than median_rect for square masks with a large mask height. The precise mask height for which median_rect will become faster than median_image depends on the computer architecture (processor type, availability of SIMD instructions like SSE2 or MMX, cache size and throughput,

memory throughput). Typically, this is the case for mask heights > 15, but can also be the case only for larger mask sizes, e.g., if SIMD instructions are unavailable and memory throughput is low.

Furthermore, it should be noted that median_rect uses a recursive implementation, which internally computes the filter response on the smallest enclosing rectangle of the domain of the input image. Therefore, if the domain of the input image only covers a small fraction of the smallest enclosing rectangle, it can happen that median_image is faster than median_rect even for larger mask heights.

Due to performance reasons, the input Image is not checked whether it contains NaNs. Using an input image with NaNs crashes HALCON.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Image to be filtered.

▷ **ImageMedian** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Filtered image.

▷ **MaskType** (input_control) . . . . . string ⇝ *string*
Filter mask type.
Default: 'circle'
List of values: MaskType ∈ {'circle', 'square'}

▷ **Radius** (input_control) . . . . . integer ⇝ *integer*
Radius of the filter mask.
Default: 1
Suggested values: Radius ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 19, 25, 31, 39, 47, 59}
Value range: $1 \leq$ Radius ≤ 4095
Minimum increment: 1
Recommended increment: 2

▷ **Margin** (input_control) . . . . . string ⇝ *string* / integer / real
Border treatment.
Default: 'mirrored'
Suggested values: Margin ∈ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Example

```
read_image (Image, 'fabrik')
median_image (Image, Median, 'circle', 3, 'continued')
dev_display(Median)
```

Complexity

For each pixel: $O(2 \cdot \text{Radius} + 1)$.

Result

If the parameter values are correct the operator median_image returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image

Possible Successors

threshold, dyn_threshold, regiongrowing

Alternatives

median_rect, rank_image, rank_rect

See also

gray_erosion_rect, gray_dilation_rect, gray_erosion_shape, gray_dilation_shape, gray_erosion, gray_dilation

References

T.S. Huang, G.J. Yang, G.Y. Tang; "A Fast Two-Dimensional Median Filtering Algorithm"; IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 27, no. 1, pp. 13-18, 1979.

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, pp. 318-320.

Module

Foundation

median_rect (Image : ImageMedian : MaskWidth, MaskHeight :)

Compute a median filter with rectangular masks.

median_rect performs a median filter on the input image Image with a rectangular mask of size MaskWidth × MaskHeight and returns the filtered image in ImageMedian.

Conceptually, the median filter sorts all gray values within the mask in ascending order and then selects the median of the gray values. The median is the "middle" one of the sorted gray values, i.e., the gray value with rank (position) $(\text{MaskWidth} \cdot \text{MaskHeight} - 1)/2 + 1$ of the sorted gray values, where the rank 1 corresponds to the smallest gray value and the rank $\text{MaskWidth} \cdot \text{MaskHeight}$ corresponds to the largest gray value within the mask (see also rank_rect).

median_rect can be used, for example, to smooth images, to suppress unwanted objects (e.g., point-like or line-like structures) that are smaller than the mask, and can therefore be used to estimate the background illumination for a shading correction or as a preprocessing step for the dynamic threshold operation (see dyn_threshold).

When using a 3x3 or 5x5 filter mask, median_rect can be executed on OpenCL devices.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

If even values instead of odd values are passed in MaskHeight or MaskWidth, median_rect uses the next larger odd values instead.

median_rect uses an algorithm with constant runtime per pixel, i.e., the runtime only depends on the size of the input image and not on the mask size. Therefore, for large mask sizes median_rect is the fastest implementation of the median filter in HALCON. Depending on the computer architecture (processor type, availability of SIMD instructions like SSE2 or MMX, cache size and throughput, memory throughput), for small mask sizes the implementation used in median_image and rank_image is faster than median_rect. Typically, this is the case for MaskHeight ≤ 15, but can also happen for larger mask sizes, e.g., if SIMD instructions are unavailable and memory throughput is low.

Furthermore, it should be noted that median_rect uses a recursive implementation, which internally computes the filter response on the smallest enclosing rectangle of the domain of the input image. Therefore, if the domain of the input image only covers a small fraction of the smallest enclosing rectangle, it can happen that median_image and rank_image are faster than median_rect even for larger values of MaskHeight.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
  Image to be filtered.
- ▷ **ImageMedian** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
  Filtered image.

▷ **MaskWidth** (input_control) . . . . . integer ⇝ *integer*
Width of the filter mask.
Default: 15
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 31, 49, 51, 61, 71, 81, 91, 101}
Value range: 3 ≤ MaskWidth ≤ 4095
Minimum increment: 2
Recommended increment: 2

▷ **MaskHeight** (input_control) . . . . . integer ⇝ *integer*
Height of the filter mask.
Default: 15
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 31, 49, 51, 61, 71, 81, 91, 101}
Value range: 3 ≤ MaskHeight ≤ 4095
Minimum increment: 2
Recommended increment: 2

Complexity

For each pixel: $O(1)$.

Result

If the parameter values are correct the operator median_rect returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image

Possible Successors

threshold, dyn_threshold, regiongrowing

Alternatives

median_image, rank_rect, rank_image

See also

gray_erosion_rect, gray_dilation_rect, gray_erosion_shape, gray_dilation_shape, gray_erosion, gray_dilation

References

S. Perreault, P. Hébert; "Median Filtering in Constant Time"; IEEE Transactions on Image Processing vol. 16, no. 9, pp. 2389-2394, 2007.

D. Cline, K.B. White, P.K. Egbert; "Fast 8-Bit Median Filtering Based On Separability"; International Conference on Image Processing, vol. V, pp. 281-284, 2007.

Module

Foundation

median_separate (Image : ImageSMedian : MaskWidth, MaskHeight, Margin :)

Separated median filtering with rectangle masks.

The operator median_separate carries out a variation of the median filtering: First two auxiliary images are created. The first one originates from a median filtering with a horizontal mask having a height of one pixel and the

width MaskWidth followed by filtering with a vertical mask having the height MaskHeight and width of one pixel. The second auxiliary image is created by filtering with the same masks, but with a reversed sequence of the operation: first the vertical, then the horizontal mask. The output image results from averaging the two auxiliary images pixel by pixel.

The operator median_separate is clearly faster than the normal operator median_image because both masks are one pixel wide, facilitating a very efficient processing. The runtime is practically *independent* of the size of the mask. For example, the operator median_separate can be well used after texture filters, where large masks are needed.

The filter can also be used several times in a row in order to enhance the smoothing.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Due to performance reasons, the input Image is not checked whether it contains NaNs. Using an input image with NaNs crashes HALCON.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
  Image to be filtered.
- ▷ **ImageSMedian** (output_object) . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
  Median filtered image.
- ▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *integer*
  Width of rank mask.
  Default: 25
  Suggested values: MaskWidth ∈ {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 27, 43, 51, 67, 91, 121, 151}
  Value range: 1 ≤ MaskWidth ≤ 401
  Minimum increment: 2
  Recommended increment: 2
- ▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *integer*
  Height of rank mask.
  Default: 25
  Suggested values: MaskHeight ∈ {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 27, 43, 51, 67, 91, 121, 151}
  Value range: 1 ≤ MaskHeight ≤ 401
  Minimum increment: 2
  Recommended increment: 2
- ▷ **Margin** (input_control) . . . . . string ⇝ *string* / integer / real
  Border treatment.
  Default: 'mirrored'
  Suggested values: Margin ∈ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Example

```
read_image(Image,'fabrik')
median_separate(Image,MedianSeparate,5,5,3)
dev_display(MedianSeparate)
```

Complexity

For each pixel: O(40).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

texture_laws, sobel_amp, deviation_image

Possible Successors

learn_ndim_norm, regiongrowing, auto_threshold

Alternatives

median_image

See also

rank_image

References

R. Haralick, L. Shapiro; “Computer and Robot Vision”; Addison-Wesley, 1992, Seite 319

Module

Foundation

median_weighted (Image : ImageWMedian : MaskType, MaskSize :)

Weighted median filtering with different rank masks.

The operator median_weighted calculates the median of the gray values within a local environment. In contrast to median_image, which uses all gray values within the environment exactly once, the operator median_weighted weights all gray values several times depending on their position. A gray value is received into the field to be sorted several times according to its weighting. The following masks are available:

’gauss’ (MaskSize = 3)

$$\begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix}$$

’inner’ (MaskSize = 3)

$$\begin{matrix} 1 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 1 \end{matrix}$$

The operator median_weighted means that, contrary to median_image, gray value corners remain.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Image to be filtered.
- ▷ **ImageWMedian** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Median filtered image.
- ▷ **MaskType** (input_control) . . . . . string ⇝ *string*
  Type of median mask.
  Default: ’inner’
  List of values: MaskType ∈ {’inner’, ’gauss’}
- ▷ **MaskSize** (input_control) . . . . . integer ⇝ *integer*
  mask size.
  Default: 3
  List of values: MaskSize ∈ {3}

Example

```
read_image(Image,'fabrik')
median_weighted(Image,MedianWeighted,'gauss',3)
dev_display(MedianWeighted)
```

Complexity

For each pixel: $O(F * log(F))$ with F = area of `MaskType`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

`read_image`

Possible Successors

`threshold`, `dyn_threshold`, `regiongrowing`

Alternatives

`median_image`, `trimmed_mean`, `sigma_image`

References

R. Haralick, L. Shapiro; “Computer and Robot Vision”; Addison-Wesley, 1992, Seite 319

Module

Foundation

midrange_image (Image, Mask : ImageMidrange : Margin :)

Calculate the average of maximum and minimum inside any mask.

The operator `midrange_image` forms the average of maximum and minimum inside the indicated mask in the whole image. Several border treatments (`Margin`) can be chosen for filtering:

gray value Pixels outside of the image border are assumed to be constant (with the indicated gray value).

'continued' Continuation of the gray values at the image border.

'cyclic' Cyclic continuation at the image borders.

'mirrored' Reflection of pixels at the image borders.

The indicated mask (= region of the mask image) is put over the image to be filtered in such a way that the center of the mask touches all pixels once.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Image to be filtered.

▷ **Mask** (input_object) . . . . . region ⇝ *object*
Filter mask.

▷ **ImageMidrange** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Filtered image.

▷ **Margin** (input_control) ..... string ⇝ *string* / integer / real
Border treatment.
Default: 'mirrored'
Suggested values: Margin ∈ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Example

```
read_image(Image,'fabrik')
draw_region(Region,WindowHandle)
midrange_image(Image,Region,Midrange,'mirrored')
dev_display(Midrange)
```

Complexity

For each pixel: $O(sqrt(F) * 5)$ with F = area of Mask.

Result

If the parameter values are correct the operator midrange_image returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image, draw_region, gen_circle, gen_rectangle1

Possible Successors

threshold, dyn_threshold, regiongrowing

Alternatives

sigma_image

See also

gen_circle, gen_rectangle1, gray_erosion_rect, gray_dilation_rect, gray_range_rect

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, Seite 319

Module

Foundation

rank_image (Image, Mask : ImageRank : Rank, Margin :)

Compute a rank filter with arbitrary masks.

rank_image performs a rank filter on the input image Image with a mask that is specified by the region Mask and returns the filtered image in ImageRank. The shape of the mask can be chosen arbitrarily and can, for example, be created with operators like gen_circle or draw_region. The position of the mask does not influence the result since the center of gravity of the mask region is used as the reference point of the mask.

Conceptually, the rank filter sorts all gray values within the mask in ascending order and then selects the gray value with rank Rank. The rank 1 corresponds to the smallest gray value and the rank A corresponds to the largest gray value within the mask. Here, A denotes the area of Mask (see area_center). For Rank $= (A-1)/2+1$,

rank_image returns the median gray value (see median_image). For Rank = 1, rank_image performs a gray value erosion (see gray_erosion_rect, gray_erosion_shape, and gray_erosion), while for Rank = A rank_image performs a gray value dilation (see gray_dilation_rect, gray_dilation_shape, and gray_dilation).

rank_image can be used, for example, to suppress noise or to suppress unwanted objects that are smaller than the mask. Furthermore, rank_image is less sensitive to noise than the corresponding gray value morphology operators. Therefore, to obtain a more robust version of the gray value morphology, instead of using 1 or A, slightly larger or smaller values should be selected for Rank.

Several border treatments can be chosen for filtering via the parameter Margin:

gray value Pixels outside of the image border are assumed to be constant (with the indicated gray value).

'continued' Continuation of the gray values at the image border.

'cyclic' Cyclic continuation at the image borders.

'mirrored' Reflection of pixels at the image borders.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

rank_image uses an algorithm with a runtime per pixel that depends on the number of runs in the mask Mask. Therefore, rank_image is slower than rank_rect for rectangular masks with a large mask height. The precise mask height for which rank_rect will become faster than rank_image depends on the computer architecture (processor type, availability of SIMD instructions like SSE2 or MMX, cache size and throughput, memory throughput). Typically, this is the case for mask heights > 15, but can also be the case only for larger mask sizes, e.g., if SIMD instructions are unavailable and memory throughput is low.

Furthermore, it should be noted that rank_rect uses a recursive implementation, which internally computes the filter response on the smallest enclosing rectangle of the domain of the input image. Therefore, if the domain of the input image only covers a small fraction of the smallest enclosing rectangle, it can happen that rank_image is faster than rank_rect even for larger mask heights.

rank_image should neither be used with Rank = 1 to perform a gray value erosion nor with Rank = A to perform a gray value dilation. In these cases, the operators gray_erosion_rect, gray_erosion_shape, or gray_erosion and gray_dilation_rect, gray_dilation_shape, or gray_dilation, respectively, are typically faster than rank_image.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Image to be filtered.

▷ **Mask** (input_object) . . . . . region ⇝ *object*
Filter mask.

▷ **ImageRank** (output_object) . . . . . multichannel-image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Filtered image.

▷ **Rank** (input_control) . . . . . integer ⇝ *integer*
Rank of the output gray value.
Default: 5
Suggested values: Rank ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 31, 49, 51, 61, 71, 81, 91, 101}
Value range: 1 ≤ Rank ≤ 4095
Minimum increment: 1
Recommended increment: 2

▷ **Margin** (input_control) . . . . . string ⇝ *string* / integer / real
Border treatment.
Default: 'mirrored'
Suggested values: Margin ∈ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Example

```
read_image (Image, 'fabrik')
```

```
draw_region (Region, WindowHandle)
rank_image (Image, Region, ImageRank, 5, 'mirrored')
dev_display(ImageRank)
```

Complexity

For each pixel: $O(N)$, where N is the number of runs of Mask (see runlength_features).

Result

If the parameter values are correct the operator rank_image returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image, draw_region, gen_circle, gen_rectangle1, gen_rectangle2

Possible Successors

threshold, dyn_threshold, regiongrowing

Alternatives

rank_rect, median_image, median_rect

See also

gray_erosion_rect, gray_dilation_rect, gray_erosion_shape, gray_dilation_shape, gray_erosion, gray_dilation

References

T.S. Huang, G.J. Yang, G.Y. Tang; "A Fast Two-Dimensional Median Filtering Algorithm"; IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 27, no. 1, pp. 13-18, 1979.

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, pp. 318-320.

Module

Foundation

rank_n (Image : RankImage : RankIndex :)

Return gray values with given rank from multiple channels.

The operator rank_n returns pixel-by-pixel the result of the rank-function over all channels.

For every pixel in the input image the following is being done: The gray values of all channels at this position are sorted in ascending order. Then the pixel with index RankIndex is selected and placed in the output image at the same position. The output image has one channel.

In the special cases RankIndex = *1* and RankIndex = *'(Number of channels)'* the minimum and maximum are returned. RankIndex = *'(Number of channels + 1) / 2'* returns the median (here, / denotes integer division). Hence, for a five-channel image, *3* returns the median.

The operator rank_n should not be confused with the operator rank_image which computes the rank within a certain mask.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / int8 / real
Multichannel gray image.

▷ **RankImage** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2 / int4 / int8 / real
Result of the rank function.

▷ **RankIndex** (input_control) . . . . . integer ⇝ *integer*
Rank of the gray value images to return.
Default: 2
Suggested values: RankIndex ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20}

Example

```
compose5 (Image1, Image2, Image3, Image4, Image5, MultiChannelImage)
rank_n (MultiChannelImage, ImageMin, 1)
rank_n (MultiChannelImage, ImageMax, 5)
rank_n (MultiChannelImage, ImageMedian, 3)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

compose2, compose3, compose4, compose5, add_channels

Alternatives

mean_n

See also

count_channels, rank_image

Module

Foundation

rank_rect (Image : ImageRank : MaskWidth, MaskHeight, Rank :)

Compute a rank filter with rectangular masks.

rank_rect performs a rank filter on the input image Image with a rectangular mask of size MaskWidth × MaskHeight and returns the filtered image in ImageRank.

Conceptually, the rank filter sorts all gray values within the mask in ascending order and then selects the gray value with rank Rank. The rank 1 corresponds to the smallest gray value and the rank MaskWidth · MaskHeight corresponds to the largest gray value within the mask. For Rank = (MaskWidth · MaskHeight − 1)/2 + 1, rank_rect returns the median gray value (see median_rect). For Rank = 1, rank_rect performs a gray value erosion (see gray_erosion_rect, gray_erosion_shape, and gray_erosion), while for Rank = MaskWidth · MaskHeight rank_rect performs a gray value dilation (see gray_dilation_rect, gray_dilation_shape, and gray_dilation).

rank_rect can be used, for example, to suppress noise or to suppress unwanted objects that are smaller than the mask. Furthermore, rank_rect is less sensitive to noise than the corresponding gray value morphology operators. Therefore, to obtain a more robust version of the gray value morphology, instead of using 1 or MaskWidth · MaskHeight, slightly larger or smaller values should be selected for Rank.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

If even values instead of odd values are passed in MaskHeight or MaskWidth, rank_rect uses the next larger odd values instead.

rank_rect uses an algorithm with constant runtime per pixel, i.e., the runtime only depends on the size of the input image and not on the mask size. Therefore, for large mask sizes rank_rect is the fastest implementation of the rank filter in HALCON. Depending on the computer architecture (processor type, availability of SIMD instructions like SSE2 or MMX, cache size and throughput, memory throughput), for small mask sizes the implementation used in rank_image is faster than rank_rect. Typically, this is the case for MaskHeight ≤ 15, but can also happen for larger mask sizes, e.g., if SIMD instructions are unavailable and memory throughput is low.

Furthermore, it should be noted that rank_rect uses a recursive implementation, which internally computes the filter response on the smallest enclosing rectangle of the domain of the input image. Therefore, if the domain of the input image only covers a small fraction of the smallest enclosing rectangle, it can happen that rank_image is faster than rank_rect even for larger values of MaskHeight.

rank_rect should neither be used with Rank $= 1$ to perform a gray value erosion nor with Rank $=$ MaskWidth $\cdot$ MaskHeight to perform a gray value dilation. In these cases, the operators gray_erosion_rect or gray_erosion_shape and gray_dilation_rect or gray_dilation_shape, respectively, are faster than rank_rect for almost all mask sizes.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Image to be filtered.

▷ **ImageRank** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Filtered image.

▷ **MaskWidth** (input_control) ..... integer ⇝ *integer*
Width of the filter mask.
Default: 15
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 31, 49, 51, 61, 71, 81, 91, 101}
Value range: 3 ≤ MaskWidth ≤ 4095
Minimum increment: 2
Recommended increment: 2

▷ **MaskHeight** (input_control) ..... integer ⇝ *integer*
Height of the filter mask.
Default: 15
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 31, 49, 51, 61, 71, 81, 91, 101}
Value range: 3 ≤ MaskHeight ≤ 4095
Minimum increment: 2
Recommended increment: 2

▷ **Rank** (input_control) ..... integer ⇝ *integer*
Rank of the output gray value.
Default: 5
Suggested values: Rank ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 31, 49, 51, 61, 71, 81, 91, 101}
Minimum increment: 1
Recommended increment: 2
Restriction: 1 <= Rank && Rank <= MaskWidth * MaskHeight

Complexity

For each pixel: $O(1)$.

Result

If the parameter values are correct the operator rank_rect returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image

Possible Successors

threshold, dyn_threshold, regiongrowing

Alternatives

rank_image, median_rect, median_image

See also

gray_erosion_rect, gray_dilation_rect, gray_erosion_shape, gray_dilation_shape, gray_erosion, gray_dilation

References

S. Perreault, P. Hébert; "Median Filtering in Constant Time"; IEEE Transactions on Image Processing, vol. 16, no. 9, pp. 2389-2394, 2007.

D. Cline, K.B. White, P.K. Egbert; "Fast 8-Bit Median Filtering Based On Separability"; International Conference on Image Processing, vol. V, pp. 281-284, 2007.

Module

Foundation

sigma_image (Image : ImageSigma : MaskHeight, MaskWidth, Sigma :)

Non-linear smoothing with the sigma filter.

The operator sigma_image carries out a non-linear smoothing of the gray values of all input images (Image). All pixels in a rectangular window (MaskHeight × MaskWidth) are used to determine the new gray value of the central pixel of this window. First, the gray value standard deviation of all pixels in the window is calculated. Then, all pixels of the window with a gray value that differs from the gray value of the central pixel by less than Sigma times this standard deviation are used to calculate the new gray value of the central pixel. The gray value of the central pixel is the average of the gray values of the selected pixels. If no pixel could be selected for the averaging of the gray values, the gray value of the central pixel remains unchanged.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

If even values instead of odd values are given for MaskHeight or MaskWidth, the routine uses the next larger odd values instead (this way the center of the filter mask is always explicitly determined).

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / cyclic / int1 / int2 / uint2 / int4 / real

Image to be smoothed.

▷ **ImageSigma** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / cyclic / int1 / int2 / uint2 / int4 / real

Smoothed image.

▷ **MaskHeight** (input_control) ..... extent.y ⇝ *integer*

Height of the mask (number of lines).
Default: 5
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskHeight
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskHeight)

▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *integer*
Width of the mask (number of columns).
Default: 5
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskWidth
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth)

▷ **Sigma** (input_control) . . . . . integer ⇝ *integer*
Max. deviation to the average.
Default: 3
Suggested values: Sigma ∈ {3, 5, 7, 9, 11, 20, 30, 50}
Value range: 0 ≤ Sigma
Minimum increment: 1
Recommended increment: 2

Example

```
read_image(Image,'fabrik')
sigma_image(Image,ImageSigma,5,5,3)
dev_display(ImageSigma)
```

Complexity

For each pixel: O(MaskHeight × MaskWidth).

Result

If the parameter values are correct the operator sigma_image returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image

Possible Successors

threshold, dyn_threshold, regiongrowing

Alternatives

anisotropic_diffusion, rank_image

See also

smooth_image, binomial_filter, gauss_filter, mean_image

References

R. Haralick, L. Shapiro; “Computer and Robot Vision”; Addison-Wesley, 1992, Seite 325

Module

Foundation

smooth_image (Image : ImageSmooth : Filter, Alpha :)

Smooth an image using various filters.

smooth_image smooths gray images using recursive filters originally developed by Deriche and Shen and using the non-recursive Gaussian filter. The following filters can be chosen via the parameter Filter:

'deriche1', 'deriche2', 'shen' and 'gauss'.

The "filter width" (i.e., the range of the filter and thereby result of the filter) can be of any size. In the case that the Deriche or Shen is chosen it decreases by increasing the filter parameter `Alpha` and increases in the case of the Gauss filter (and `Alpha` corresponds to the standard deviation of the Gaussian function). An approximation of the appropriate size of the filter width `Alpha` is performed by the operator `info_smooth`.

Non-recursive filters like the Gaussian filter are often implemented using filter-masks. In this case the runtime of the operator increases with increasing size of the filter mask. The runtime of the recursive filters remains constant; except the border treatment becomes a little bit more time consuming. The Gaussian filter becomes slow in comparison to the recursive ones but is in contrast to them isotropic (the filter 'deriche2' is only weakly direction sensitive). A comparable result of the smoothing is achieved by choosing the following values for the parameter:

$$\begin{aligned} \texttt{Alpha}('deriche2') &= \frac{\texttt{Alpha}('deriche1')}{2} \\ \texttt{Alpha}('shen') &= \frac{\texttt{Alpha}('deriche1')}{2} \\ \texttt{Alpha}('gauss') &= \frac{1.77}{\texttt{Alpha}('deriche1')} \end{aligned}$$

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Image to be smoothed.

▷ **`ImageSmooth`** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Smoothed image.

▷ **`Filter`** (input_control) ..... string ⇝ *string*
Filter.
Default: 'deriche2'
List of values: `Filter` ∈ {'deriche1', 'deriche2', 'shen', 'gauss'}

▷ **`Alpha`** (input_control) ..... real ⇝ *real*
Filter parameter: small values cause strong smoothing (vice versa by using *'gauss'*).
Default: 0.5
Suggested values: `Alpha` ∈ {0.1, 0.2, 0.3, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0, 5.0, 7.0, 10.0}
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Alpha > 0

Example

```
info_smooth('deriche2',0.5,Size,Coeffs)
smooth_image(Input,Smooth,'deriche2',7)
```

Result

If the parameter values are correct the operator `smooth_image` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system ('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

read_image

Possible Successors

threshold, dyn_threshold, regiongrowing

Alternatives

binomial_filter, gauss_filter, mean_image, derivate_gauss, isotropic_diffusion

See also

info_smooth, median_image, sigma_image, anisotropic_diffusion

References

R.Deriche: "Fast Algorithms for Low-Level Vision"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-12, no. 1; S. 78-87; 1990.

Module

Foundation

trimmed_mean (Image, Mask : ImageTMean : Number, Margin :)

Smooth an image with an arbitrary rank mask.

The operator trimmed_mean carries out a non-linear smoothing of the gray values of all input images (Image). The filter mask (Mask) is passed in the form of a region. The average of Number gray values located near the median is calculated. Several border treatments can be chosen for filtering (Margin):

gray value Pixels outside of the image border are assumed to be constant (with the indicated gray value).

'continued' Continuation of the gray values at the image border.

'cyclic' Cyclic continuation at the image borders.

'mirrored' Reflection of pixels at the image borders.

The indicated mask (= region of the mask image) is put over the image to be filtered in such a way that the center of the mask touches all pixels once. For each of these pixels all neighboring pixels covered by the mask are sorted in an ascending sequence according to their gray values. Thus, each of these sorted gray value sequences contains exactly as many gray values as the mask has pixels. If F is the area of the mask the average of these sequences is calculated as follows: The first (F - Number)/2 gray values are ignored. Then the following Number gray values are summed up and divided by Number. Again the remaining (F - Number)/2 gray values are ignored.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Image to be filtered.

▷ **Mask** (input_object) . . . . . region ⇝ *object*
Image whose region serves as filter mask.

▷ **ImageTMean** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Filtered output image.

▷ **Number** (input_control) . . . . . integer ⇝ *integer*
Number of averaged pixels. Typical value: Surface(Mask) / 2.
Default: 5
Suggested values: Number ∈ {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31}
Value range: 1 ≤ Number ≤ 401
Minimum increment: 1
Recommended increment: 2

▷ **Margin** (input_control) . . . . . string ⇝ *string* / integer / real
Border treatment.
Default: 'mirrored'
Suggested values: Margin ∈ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Example

```
read_image(Image,'fabrik')
draw_region(Region,WindowHandle)
trimmed_mean(Image,Region,TrimmedMean,5,'mirrored')
dev_display(TrimmedMean)
```

Result

If the parameter values are correct the operator trimmed_mean returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image, draw_region, gen_circle, gen_rectangle1

Possible Successors

threshold, dyn_threshold, regiongrowing

Alternatives

sigma_image, median_weighted, median_image

See also

gen_circle, gen_rectangle1, gray_erosion_rect, gray_dilation_rect

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, Seite 320

Module

Foundation

12.17 Texture Inspection

deviation_image (Image : ImageDeviation : Width, Height :)

Calculate the standard deviation of gray values within rectangular windows.

deviation_image calculates the standard deviation of gray values in the image Image within a rectangular mask of size (Height, Width). The resulting image is returned in ImageDeviation. To better use the range of gray values available in the output image, the result is multiplied by 2. If the parameters Height and Width are even, they are changed to the next larger odd value. At the image borders the gray values are mirrored.

Attention

deviation_image can be executed on OpenCL devices. As the same OpenCL code is used, the same limitations as for mean_image apply. Since deviation_image uses single precision floating point arithmetic internally, the results may differ slightly from the CPU version.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int4 / real / int2 / uint2
Image for which the standard deviation is to be calculated.

▷ **ImageDeviation** (output_object) . . . . . image(-array) ⇝ *object :* byte / int4 / real / int2 / uint2
Image containing the standard deviation.

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the mask in which the standard deviation is calculated.
Default: 11
Suggested values: `Width` ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25}
Restriction: 3 <= Width && odd(Width) && Width < width(Image) * 2

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the mask in which the standard deviation is calculated.
Default: 11
Suggested values: `Height` ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25}
Restriction: 3 <= Height && odd(Height) && Height < height(Image) * 2

Example

```
read_image(Image,'fabrik')
dev_display(Image)
deviation_image(Image,Deviation,9,9)
dev_display(Image)
```

Result

`deviation_image` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

`disp_image`

Alternatives

`entropy_image`, `entropy_gray`

See also

`convol_image`, `texture_laws`, `intensity`

Module

Foundation

entropy_image (Image : ImageEntropy : Width, Height :)

Calculate the entropy of gray values within a rectangular window.

`entropy_image` calculates the entropy of gray values in the image `Image` within a rectangular mask of size (`Height`, `Width`). The resulting image is returned in `ImageEntropy`, in which the entropy is multiplied by 32. If the parameters `Height` and `Width` are even, they are changed to the next larger odd value. At the image borders the gray values are mirrored.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Image for which the entropy is to be calculated.

▷ **ImageEntropy** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Entropy image.

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the mask in which the entropy is calculated.
Default: 9
Suggested values: Width ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25}
Restriction: 3 <= Width && odd(Width)

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the mask in which the entropy is calculated.
Default: 9
Suggested values: Height ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25}
Restriction: 3 <= Height && odd(Height)

Example

```
read_image(Image,'fabrik')
dev_display(Image)
entropy_image(Image,Entropy,9,9)
dev_display(Entropy)
```

Result

entropy_image returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

disp_image

Alternatives

entropy_gray

See also

energy_gabor, entropy_gray

Module

Foundation

texture_laws (Image : ImageTexture : FilterTypes, Shift, FilterSize :)

Filter an image using a Laws texture filter.

texture_laws applies a texture transformation (according to Laws) to an image. This is done by convolving the input image with a special filter mask. The filters are:

9 different 3×3 matrices obtainable from the following three vectors:

$$\begin{array}{lll} \mathrm{l} & = & [\ 1 \quad 2 \quad 1] \\ \mathrm{e} & = & [-1 \quad 0 \quad 1] \\ \mathrm{s} & = & [-1 \quad 2 \quad -1] \end{array}$$

25 different 5×5 matrices obtainable from the following five vectors:

$$\begin{array}{lll} \mathrm{l} & = & [\ 1 \quad 4 \quad 6 \quad 4 \quad 1] \\ \mathrm{e} & = & [-1 \quad -2 \quad 0 \quad 2 \quad 1] \\ \mathrm{s} & = & [-1 \quad 0 \quad 2 \quad 0 \quad -1] \\ \mathrm{w} & = & [-1 \quad 2 \quad 0 \quad -2 \quad 1] \\ \mathrm{r} & = & [\ 1 \quad -4 \quad 6 \quad -4 \quad 1] \end{array}$$

49 different 7×7 matrices obtainable from the following seven vectors:

$$\begin{array}{lll} \mathrm{l} & = & [\ 1 \quad 6 \quad 15 \quad 20 \quad 15 \quad 6 \quad 1] \\ \mathrm{e} & = & [-1 \quad -4 \quad -5 \quad 0 \quad 5 \quad 4 \quad 1] \\ \mathrm{s} & = & [-1 \quad -2 \quad 1 \quad 4 \quad 1 \quad -2 \quad -1] \\ \mathrm{w} & = & [-1 \quad 0 \quad 3 \quad 0 \quad -3 \quad 0 \quad 1] \\ \mathrm{r} & = & [\ 1 \quad -2 \quad -1 \quad 4 \quad -1 \quad -2 \quad -1] \\ \mathrm{u} & = & [\ 1 \quad -4 \quad 5 \quad 0 \quad -5 \quad 4 \quad -1] \\ \mathrm{o} & = & [-1 \quad 6 \quad -15 \quad 20 \quad -15 \quad 6 \quad -1] \end{array}$$

The names of the filters are mnemonics for “level,” “edge,” “spot,” “wave,” “ripple,” “undulation,” and “oscillation.”

For most of the filters the resulting gray values must be modified by a `Shift`. This makes the different textures in the output image more comparable to each other, provided suitable filters are used.

The name of the filter is composed of the letters of the two vectors used, where the first letter denotes convolution in the column direction while the second letter denotes convolution in the row direction.

Attention

`texture_laws` can be executed on OpenCL devices.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **`Image`** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Images to which the texture transformation is to be applied.
- ▷ **`ImageTexture`** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2
  Texture images.
- ▷ **`FilterTypes`** (input_control) . . . . . string ⇝ *string*
  Desired filter.
  Default: ’el’
  Suggested values: `FilterTypes` ∈ {’ll’, ’le’, ’ls’, ’lw’, ’lr’, ’lu’, ’lo’, ’el’, ’ee’, ’es’, ’ew’, ’er’, ’eu’, ’eo’, ’sl’, ’se’, ’ss’, ’sw’, ’sr’, ’su’, ’so’, ’wl’, ’we’, ’ws’, ’ww’, ’wr’, ’wu’, ’wo’, ’rl’, ’re’, ’rs’, ’rw’, ’rr’, ’ru’, ’ro’, ’ul’, ’ue’, ’us’, ’uw’, ’ur’, ’uu’, ’uo’, ’ol’, ’oe’, ’os’, ’ow’, ’or’, ’ou’, ’oo’}
- ▷ **`Shift`** (input_control) . . . . . integer ⇝ *integer*
  Shift to reduce the gray value dynamics.
  Default: 2
  Suggested values: `Shift` ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- ▷ **`FilterSize`** (input_control) . . . . . integer ⇝ *integer*
  Size of the filter kernel.
  Default: 5
  List of values: `FilterSize` ∈ {3, 5, 7}

Example

```
* Simple two-dimensional pixel classification
dev_get_window (WindowHandle)
read_image(Image,'combine')
texture_laws(Image,Texture1,'es',3,7)
texture_laws(Image,Texture2,'le',7,7)
MaskSize := 51
mean_image(Texture1,H1,MaskSize,MaskSize)
mean_image(Texture2,H2,MaskSize,MaskSize)
dev_clear_window ()
dev_display (Image)
dev_set_color ('green')
write_string (WindowHandle, 'Mark region within one texture area')
draw_region(Region,WindowHandle)
reduce_domain(H1,Region,Foreground1)
reduce_domain(H2,Region,Foreground2)
histo_2dim(Region,Foreground1,Foreground2,Histo)
get_image_size (Image, Width, Height)
threshold(Histo,Characteristic_area,1,Width*Height)
ShowIntermediateResult := 0
if (ShowIntermediateResult)
  histo_2dim(H1,H1,H2,HistoFull)
  dev_clear_window ()
  dev_set_lut ('sqrt')
  dev_display (HistoFull)
  dev_set_draw ('margin')
  dev_display (Characteristic_area)
  stop ()
  dev_set_lut ('default')
  dev_set_draw ('fill')
endif
class_2dim_sup(H1,H2,Characteristic_area,Seg)
dev_display (Image)
dev_set_color ('red')
dev_display (Seg)
```

Result

`texture_laws` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Successors

`mean_image`, `binomial_filter`, `gauss_filter`, `median_image`, `histo_2dim`, `learn_ndim_norm`, `threshold`

Alternatives

`convol_image`

See also

`class_2dim_sup`, `class_ndim_norm`

References

Laws, Kenneth Ivan. “Textured Image Segmentation”; Ph.D. Thesis, Department of Electrical Engineering, Image Processing Institute, University of Southern California, 1980

Module

Foundation

12.18 Wiener Filter

gen_psf_defocus (: Psf : PSFwidth, PSFheight, Blurring :)

Generate an impulse response of an uniform out-of-focus blurring.

gen_psf_defocus generates an impulse response (spatial domain) of an uniform out-of-focus blurring and writes it into an image of HALCON image type real. Blurring specifies the extent of blurring by defining the "blur radius" (out-of-focus blurring maps each image pixel on a small circle with a radius of Blurring - specified in "number of pixels"). If specified less than zero, the absolute value of Blurring is used. The result image of gen_psf_defocus encloses an spatial domain impulse response of the specified blurring. Its representation presumes the origin in the upper left corner. This results in the following disposition of an NxM sized image:

- first rectangle ("upper left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = 0..(M/2) - 1$)
  - conforms to the fourth quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 0..N/2$ and $y = 0.. - M/2$
- second rectangle ("upper right"): (image coordinates $xb = N/2..N - 1$, $yb = 0..(M/2) - 1$)
  - conforms to the third quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ and $y = -1.. - M/2$
- third rectangle ("lower left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = M/2..M - 1$)
  - conforms to the first quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 1..N/2$ and $y = M/2..0$
- fourth rectangle ("lower right"): (image coordinates $xb = N/2..N - 1$, $yb = M/2..M - 1$)
  - conforms to the second quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ and $y = M/2..1$

This representation conforms to that of the impulse response parameter of the HALCON-operator wiener_filter. So one can use gen_psf_defocus to generate an impulse response for Wiener filtering.

Parameters

▷ **Psf** (output_object) ..... image ⇝ *object :* real
Impulse response of uniform out-of-focus blurring.

▷ **PSFwidth** (input_control) ..... integer ⇝ *integer*
Width of result image.
Default: 256
Suggested values: PSFwidth ∈ {128, 256, 512, 1024}
Value range: 1 ≤ PSFwidth

▷ **PSFheight** (input_control) ..... integer ⇝ *integer*
Height of result image.
Default: 256
Suggested values: PSFheight ∈ {128, 256, 512, 1024}
Value range: 1 ≤ PSFheight

▷ **Blurring** (input_control) ..... real ⇝ *real*
Degree of Blurring.
Default: 5.0
Suggested values: Blurring ∈ {1.0, 5.0, 10.0, 15.0, 18.0}

Result

gen_psf_defocus returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

simulate_motion, gen_psf_motion

Possible Successors

simulate_defocus, wiener_filter, wiener_filter_ni

See also

simulate_defocus, gen_psf_motion, simulate_motion, wiener_filter, wiener_filter_ni

References

Reginald L. Lagendijk, Jan Biemond: Iterative Identification and Restoration of Images, Kluwer Academic Publishers Boston/Dordrecht/London, 1991

M. Lückenhaus:"Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse"; Diplomarbeit; Technische Universität München, Institut für Informatik; Lehrstuhl Prof. Radig; 1995.

Module

Foundation

gen_psf_motion (: Psf : PSFwidth, PSFheight, Blurring, Angle, Type :)

Generate an impulse response of a (linearly) motion blurring.

gen_psf_motion generates an impulse response (spatial domain) of a blurring caused by a relative motion between the object and the camera during exposure. The generated impulse response is output into an image of HALCON image type real. PSFwidth and PSFheight define the width and height of the output image. The blurring motion moves along an even. Angle fixes its direction by specifying the angle between the motion direction and the x-axis (anticlockwise, measured in degrees). To specify different velocity behavior five PSF prototypes can be generated. Type switches between the following prototypes:

1. reverse ramp (crude model for acceleration)
2. reverse trapezoid (crude model for high acceleration)
3. square pulse (exact model for constant velocity), this is default
4. forward trapezoid (crude model for deceleration)
5. forward ramp (crude model for high deceleration)

(default value is 3.)

The blurring affects all part of the image uniformly. Blurring controls the extent of blurring. It specifies the number of pixels (lying one after another) that are affetcetd by the blurring. This number is determined by velocity of the motion and exposure time. If Blurring is a negative number, an adequate blurring in reverse direction is simulated. If Angle is a negative number, it is interpreted clockwise. If Angle exceeds 360 or falls below -360, it is transformed modulo(360) in an adequate number between $[0..360]$ resp. $[-360..0]$. The result image of gen_psf_motion encloses an spatial domain impulse response of the specified blurring. Its representation presumes the origin in the upper left corner. This results in the following disposition of an NxM sized image:

- first rectangle ("upper left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = 0..(M/2) - 1$)
  - conforms to the fourth quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 0..N/2$ and $y = 0.. - M/2$
- second rectangle ("upper right"): (image coordinates $xb = N/2..N - 1$, $yb = 0..(M/2) - 1$)
  - conforms to the third quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ and $y = -1.. - M/2$
- third rectangle ("lower left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = M/2..M - 1$)
  - conforms to the first quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 1..N/2$ and $y = M/2..0$

- fourth rectangle (“lower right”): (image coordinates $xb = N/2..N-1$, $yb = M/2..M-1$)
  - conforms to the second quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2..-1$ and $y = M/2..1$

This representation conforms to that of the impulse response parameter of the HALCON-operator `wiener_filter`. So one can use `gen_psf_motion` to generate an impulse response for Wiener filtering a motion blurred image.

Parameters

▷ **`Psf`** (output_object) ..... image ⇝ *object :* real
Impulse response of motion-blur.

▷ **`PSFwidth`** (input_control) ..... integer ⇝ *integer*
Width of impulse response image.
Default: 256
Suggested values: `PSFwidth` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ PSFwidth

▷ **`PSFheight`** (input_control) ..... integer ⇝ *integer*
Height of impulse response image.
Default: 256
Suggested values: `PSFheight` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ PSFheight

▷ **`Blurring`** (input_control) ..... real ⇝ *real*
Degree of motion-blur.
Default: 20.0
Suggested values: `Blurring` ∈ {5.0, 10.0, 20.0, 30.0, 40.0}

▷ **`Angle`** (input_control) ..... integer ⇝ *integer*
Angle between direction of motion and x-axis (anticlockwise).
Default: 0
Suggested values: `Angle` ∈ {0, 45, 90, 180, 270}

▷ **`Type`** (input_control) ..... integer ⇝ *integer*
PSF prototype resp. type of motion.
Default: 3
List of values: `Type` ∈ {1, 2, 3, 4, 5}

Result

`gen_psf_motion` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`simulate_defocus`, `gen_psf_defocus`

Possible Successors

`simulate_motion`, `wiener_filter`, `wiener_filter_ni`

See also

`simulate_motion`, `simulate_defocus`, `gen_psf_defocus`, `wiener_filter`, `wiener_filter_ni`

References

Anil K. Jain:Fundamentals of Digital Image Processing, Prentice-Hall International Inc., Englewood Cliffs, New Jersey, 1989

M. Lückenhaus:“Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse”; Diplomarbeit; Technische Universität München, Institut für Informatik; Lehrstuhl Prof. Radig; 1995.

Kha-Chye Tan, Hock Lim, B. T. G. Tan:“Restoration of Real-World Motion-Blurred Images”;S. 291-299 in: CVGIP Graphical Models and Image Processing, Vol. 53, No. 3, May 1991

Module

Foundation

simulate_defocus (Image : DefocusedImage : Blurring :)

Simulate an uniform out-of-focus blurring of an image.

simulate_defocus simulates out-of-focus blurring of an image. All parts of the image are blurred uniformly. Blurring specifies the extent of blurring by defining the "blur radius" (out-of-focus blurring maps each image pixel on a small circle with a radius of Blurring - specified in "number of pixels"). If specified less than null, the absolute value of Blurring is used. Simulation of blurring is done by a convolution of the image with a blurring specific impulse response. The convolution is realized by multiplication in the Fourier domain.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image to blur.

▷ **DefocusedImage** (output_object) ..... image(-array) ⇝ *object :* real
Blurred image.

▷ **Blurring** (input_control) ..... real ⇝ *real*
Degree of blurring.
Default: 5.0
Suggested values: Blurring ∈ {1.0, 5.0, 10.0, 15.0, 18.0}

Result

simulate_defocus returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty simulate_defocus returns with an error message.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Predecessors

gen_psf_defocus, simulate_motion, gen_psf_motion

Possible Successors

wiener_filter, wiener_filter_ni

See also

gen_psf_defocus, simulate_motion, gen_psf_motion

References

Reginald L. Lagendijk, Jan Biemond: Iterative Identification and Restoration of Images, Kluwer Academic Publishers Boston/Dordrecht/London, 1991

M. Lückenhaus:"Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse"; Diplomarbeit; Technische Universität München, Institut für Informatik; Lehrstuhl Prof. Radig; 1995.

Module

Foundation

simulate_motion (Image : MovedImage : Blurring, Angle, Type :)

Simulation of (linearly) motion blur.

simulate_motion simulates blurring caused by a relative motion between the object and the camera during exposure. The simulated motion moves along an even. Angle fixes its direction by specifying the angle between the motion direction and the x-axis (anticlockwise, measured in degrees). Simulation is done by a convolution of the image with a blurring specific impulse response. The convolution is realized by multiplication in the Fourier domain. simulate_motion offers five prototypes of impulse responses conforming to different acceleration behaviors. Type allows to choose one of the following PSF prototypes:

1. reverse ramp (crude model for acceleration)
2. reverse trapezoid (crude model for high acceleration)
3. square pulse (exact model for constant velocity), this is default
4. forward trapezoid (crude model for deceleration)
5. forward ramp (crude model for high deceleration)

(default value is 3.)

The simulated blurring affects all part of the image uniformly. `Blurring` controls the extent of blurring. It specifies the number of pixels (lying one after another) that are affected by the blurring. This number is determined by velocity of the motion and exposure time. If `Blurring` is a negative number, an adequate blurring in reverse direction is simulated. If `Angle` is a negative number, it is interpreted clockwise. If `Angle` exceeds 360 or falls below -360, it is transformed modulo(360) in an adequate number between $[0..360]$ resp. $[-360..0]$.

Parameters

▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
image to be blurred.

▷ **`MovedImage`** (output_object) ..... image ⇝ *object :* real
motion blurred image.

▷ **`Blurring`** (input_control) ..... real ⇝ *real*
extent of blurring.
Default: 20.0
Suggested values: `Blurring` ∈ {5.0, 10.0, 20.0, 30.0, 40.0}

▷ **`Angle`** (input_control) ..... integer ⇝ *integer*
Angle between direction of motion and x-axis (anticlockwise).
Default: 0
Suggested values: `Angle` ∈ {0, 45, 90, 180, 270}

▷ **`Type`** (input_control) ..... integer ⇝ *integer*
impulse response of motion blur.
Default: 3
List of values: `Type` ∈ {1, 2, 3, 4, 5}

Result

`simulate_motion` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty `simulate_motion` returns with an error message.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Predecessors

`gen_psf_motion`, `gen_psf_motion`

Possible Successors

`simulate_defocus`, `wiener_filter`, `wiener_filter_ni`

See also

`gen_psf_motion`, `simulate_defocus`, `gen_psf_defocus`

References

Anil K. Jain:Fundamentals of Digital Image Processing, Prentice-Hall International Inc., Englewood Cliffs, New Jersey, 1989

M. Lückenhaus:“Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse”; Diplomarbeit; Technische Universität München, Institut für Informatik; Lehrstuhl Prof. Radig; 1995.

Kha-Chye Tan, Hock Lim, B. T. G. Tan:“Restoration of Real-World Motion-Blurred Images”;S. 291-299 in: CVGIP Graphical Models and Image Processing, Vol. 53, No. 3, May 1991

Module

Foundation

wiener_filter (Image, Psf, FilteredImage : RestoredImage : :)

Image restoration by Wiener filtering.

wiener_filter produces an estimate of the original image (= image without noise and blurring) by minimizing the mean square error between estimated and original image. wiener_filter can be used to restore images corrupted by noise and/or blurring (e.g., motion blur, atmospheric turbulence or out-of-focus blur). Method and realization of this restoration technique bases on the following model: The corrupted image is interpreted as the output of a (disturbed) linear system. Functionality of a linear system is determined by its specific impulse response. So the convolution of original image and impulse response results in the corrupted image. The specific impulse response describes image acquisition and the occurred degradations. In the presence of additive noise an additional noise term must be considered. So the corrupted image can be modeled as the result of

$[convolution(impulse\_response, original\_image)] + noise\_term$

The noise term encloses two different terms describing image-dependent and image-independent noise. According to this model, two terms must be known for restoration by Wiener filtering:

1. degradation-specific impulse response
2. noise term

So wiener_filter needs a smoothed version of the input image to estimate the power spectral density of noise and original image. One can use one of the smoothing HALCON-filters (e.g., eliminate_min_max) to get this version. wiener_filter needs further the impulse response that describes the specific degradation. This impulse response (represented in spatial domain) must fit into an image of HALCON image type real. There exist two HALCON-operators for generation of an impulse response for motion blur and out-of-focus (see gen_psf_motion, gen_psf_defocus). The representation of the impulse response presumes the origin in the upper left corner. This results in the following disposition of an NxM sized image:

- first rectangle (“upper left”): (image coordinates $xb = 0..(N/2) - 1$, $yb = 0..(M/2) - 1$)
  - conforms to the fourth quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 0..N/2$ and $y = 0.. - M/2$
- second rectangle (“upper right”): (image coordinates $xb = N/2..N - 1$, $yb = 0..(M/2) - 1$)
  - conforms to the third quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ and $y = -1.. - M/2$
- third rectangle (“lower left”): (image coordinates $xb = 0..(N/2) - 1$, $yb = M/2..M - 1$)
  - conforms to the first quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 1..N/2$ and $y = M/2..0$
- fourth rectangle (“lower right”): (image coordinates $xb = N/2..N - 1$, $yb = M/2..M - 1$)
  - conforms to the second quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ and $y = M/2..1$

wiener_filter works as follows:

- estimation of the power spectrum density of the original image by using the smoothed version of the corrupted image,
- estimation of the power spectrum density of each pixel by subtracting smoothed version from not smoothed version,
- building the Wiener filter kernel with the quotient of power spectrum densities of noise and original image and with the impulse response,
- processing the convolution of image and Wiener filter frequency response.

The result image has got image type real.

Attention

Psf must be of image type real and conform to Image and FilteredImage in image width and height.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Corrupted image.

▷ **Psf** (input_object) ..... (multichannel-)image ⇝ *object :* real
impulse response (PSF) of degradation (in spatial domain).

▷ **FilteredImage** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Smoothed version of corrupted image.

▷ **RestoredImage** (output_object) ..... image ⇝ *object :* real
Restored image.

Example

```
/* Restoration of a noisy image (size=256x256), that was blurred by motion*/
Hobject object;
Hobject restored;
Hobject psf;
Hobject noisefiltered;
/* 1. Generate a Point-Spread-Function for a motion-blur with       */
/*    parameter a=10 and direction along the x-axis                 */
gen_psf_motion(&psf,256,256,10,0,3);
/* 2. Noisefiltering of the image                                   */
median_image(object,&noisefiltered,"circle",2,0);
/* 3. Wiener-filtering                                              */
wiener_filter(object,psf,noisefiltered,&restored);
```

Result

`wiener_filter` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty `wiener_filter` returns with an error message.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on channel level.

Possible Predecessors

`gen_psf_motion`, `simulate_motion`, `simulate_defocus`, `gen_psf_defocus`, `optimize_fft_speed`

Alternatives

`wiener_filter_ni`

See also

`simulate_motion`, `gen_psf_motion`, `simulate_defocus`, `gen_psf_defocus`

References

M. Lückenhaus:“Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse”; Diplomarbeit; Technische Universität München, Institut für Informatik; Lehrstuhl Prof. Radig; 1995

Azriel Rosenfeld, Avinash C. Kak: Digital Picture Processing, Computer Science and Aplied Mathematics, Academic Press New York/San Francisco/London 1982

Module

Foundation

wiener_filter_ni (Image, Psf, NoiseRegion : RestoredImage : MaskWidth, MaskHeight :)

Image restoration by Wiener filtering.

wiener_filter_ni (ni = noise-estimation integrated) produces an estimate of the original image (= image without noise and blurring) by minimizing the mean square error between estimated and original image. wiener_filter can be used to restore images corrupted by noise and/or blurring (e.g., motion blur, atmospheric turbulence or out-of-focus blur). Method and realization of this restoration technique bases on the following model: The corrupted image is interpreted as the output of a (disturbed) linear system. Functionality of a linear system is determined by its specific impulse response. So the convolution of original image and impulse response results in the corrupted image. The specific impulse response describes image acquisition and the occurred degradations. In the presence of additive noise an additional noise term must be considered. So the corrupted image can be modeled as the result of

$$[convolution(impulse\_response, original\_image)] + noise\_term$$

The noise term encloses two different terms describing image-dependent and image-independent noise. According to this model, two terms must be known for restoration by Wiener filtering:

1. degradation-specific impulse response
2. noise term

wiener_filter_ni estimates the noise term as follows: The user defines a region that is suitable for noise estimation within the image (homogeneous as possible, as edges or textures aggravate noise estimation). After smoothing within this region by an (unweighted) median filter and subtracting smoothed version from unsmoothed, the average noise amplitude of the region is processed within wiener_filter_ni. This amplitude together with the average gray value within the region allows estimating the quotient of the power spectral densities of noise and original image (in contrast to wiener_filter wiener_filter_ni assumes a rather constant quotient within the whole image). The user can define width and height of the rectangular (median-)filter mask to influence the noise estimation (MaskWidth, MaskHeight). wiener_filter_ni needs further the impulse response that describes the specific degradation. This impulse response (represented in spatial domain) must fit into an image of HALCON image type real. There exist two HALCON-operators for generation of an impulse response for motion blur and out-of-focus (see gen_psf_motion, gen_psf_defocus). The representation of the impulse response presumes the origin in the upper left corner. This results in the following disposition of an NxM sized image:

- first rectangle (“upper left”): (image coordinates $xb = 0..(N/2) - 1$, $yb = 0..(M/2) - 1$)
  - conforms to the fourth quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 0..N/2$ and $y = 0.. - M/2$
- second rectangle (“upper right”): (image coordinates $xb = N/2..N - 1$, $yb = 0..(M/2) - 1$)
  - conforms to the third quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ and $y = -1.. - M/2$
- third rectangle (“lower left”): (image coordinates $xb = 0..(N/2) - 1$, $yb = M/2..M - 1$)
  - conforms to the first quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 1..N/2$ and $y = M/2..0$
- fourth rectangle (“lower right”): (image coordinates $xb = N/2..N - 1$, $yb = M/2..M - 1$)
  - conforms to the second quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ and $y = M/2..1$

wiener_filter works as follows:

- estimating the quotient of the power spectrum densities of noise and original image,
- building the Wiener filter kernel with the quotient of power spectrum densities of noise and original image and with the impulse response,
- processing the convolution of image and Wiener filter frequency response.

The result image has got image type real.

Attention

Psf must be of image type real and conform to Image in width and height. The Region used for noise estimation must lie completely within the image. If MaskWidth or MaskHeight is an even number, it is replaced by the next higher odd number (this allows the unique extraction of the center of the filter mask). Width/height of the mask may not exceed the image width/height or be less than null.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Corrupted image.

▷ **Psf** (input_object) ..... (multichannel-)image ⇝ *object :* real
impulse response (PSF) of degradation (in spatial domain).

▷ **NoiseRegion** (input_object) ..... region(-array) ⇝ *object*
Region for noise estimation.

▷ **RestoredImage** (output_object) ..... image ⇝ *object :* real
Restored image.

▷ **MaskWidth** (input_control) ..... integer ⇝ *integer*
Width of filter mask.
Default: 3
Suggested values: `MaskWidth` ∈ {3, 5, 7, 9}
Value range: 0 ≤ `MaskWidth` ≤ width(Image)

▷ **MaskHeight** (input_control) ..... integer ⇝ *integer*
Height of filter mask.
Default: 3
Suggested values: `MaskHeight` ∈ {3, 5, 7, 9}
Value range: 0 ≤ `MaskHeight` ≤ height(Image)

Example

```
/* Restoration of a noisy image (size=256x256), that was blurred by motion*/
Hobject object;
Hobject restored;
Hobject psf;
Hobject noise_region;
/* 1. Generate a Point-Spread-Function for a motion-blur with       */
/*    parameter a=10 and direction of the x-axis                    */
gen_psf_motion(&psf,256,256,10,0,3);
/* 2. Segmentation of a region for the noise-estimation             */
open_window(0,0,256,256,0,"visible",&WindowHandle);
disp_image(object,WindowHandle);
draw_region(&noise_region,draw_region);
/* 3. Wiener-filtering                                              */
wiener_filter_ni(object,psf,noise_region,&restored,3,3);
```

Result

`wiener_filter_ni` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty `wiener_filter_ni` returns with an error message.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on channel level.

Possible Predecessors

`gen_psf_motion`, `simulate_motion`, `simulate_defocus`, `gen_psf_defocus`, `optimize_fft_speed`

Alternatives

`wiener_filter`

See also

`simulate_motion`, `gen_psf_motion`, `simulate_defocus`, `gen_psf_defocus`

References

M. Lückenhaus:“Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse”; Diplomarbeit; Technische Universität München, Institut für Informatik; Lehrstuhl Prof. Radig; 1995

Azriel Rosenfeld, Avinash C. Kak: Digital Picture Processing, Computer Science and Aplied Mathematics, Academic Press New York/San Francisco/London 1982

Module

Foundation

Chapter 13

Graphics

13.1 3D Scene

add_scene_3d_camera (: : Scene3D, CameraParam : CameraIndex)

Add a camera to a 3D scene.

add_scene_3d_camera adds a new camera to the 3D scene Scene3D and returns the index of this camera in CameraIndex. The camera parameters of the camera must be given in CameraParam. By default the new camera is located at the origin of the world coordinate system. The pose of the camera CameraIndex can be set with set_scene_3d_camera_pose.

Attention

Cameras with hypercentric lenses are not supported. For displaying large faces (or primitives) with a non-zero distortion in CameraParam, note that the distortion is only applied to the points of the model. In the projection, these points are subsequently connected by straight lines. For a good approximation of the distorted lines, please use a triangulation with sufficiently small triangles.

Parameters

- ▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
  Handle of the 3D scene.
- ▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
  Parameters of the new camera.
- ▷ **CameraIndex** (output_control) . . . . . integer ⇝ *integer*
  Index of the new camera in the 3D scene.

Result

add_scene_3d_camera returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_scene_3d

Possible Successors

set_scene_3d_camera_pose, display_scene_3d

Module

3D Metrology

add_scene_3d_instance (: : Scene3D, ObjectModel3D, Pose : InstanceIndex)

Add an instance of a 3D object model to a 3D scene.

add_scene_3d_instance adds an instance of the 3D model ObjectModel3D to the 3D scene Scene3D and returns its index in InstanceIndex. If multiple 3D object models are supplied, it is possible to set one pose for all instances or one pose for every instance.

The pose of the object instance in the scene coordinate system must be given in Pose. The operator set_scene_3d_instance_pose can be used to change this pose. As long as no global scene pose is set with set_scene_3d_to_world_pose, this pose is evaluated relative to the world coordinate system.

Parameters of the instance, such as its color, are set with the operator set_scene_3d_instance_param.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **ObjectModel3D** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
Handle of the 3D object model.

▷ **Pose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
Pose of the 3D object model.
Number of elements: 7

▷ **InstanceIndex** (output_control) . . . . . integer ⇝ *integer*
Index of the new instance in the 3D scene.

Result

add_scene_3d_instance returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_scene_3d

Possible Successors

set_scene_3d_instance_param, set_scene_3d_instance_pose

See also

remove_scene_3d_instance

Module

3D Metrology

add_scene_3d_label (: : Scene3D, Text, ReferencePoint, Position, RelatesTo : LabelIndex)

Add a text label to a 3D scene.

add_scene_3d_label adds a label with the text in Text to the 3D scene Scene3D and returns its index in LabelIndex.

The reference point of the label must be given as *[X,Y,Z]* point in the coordinate system of the 3D scene in ReferencePoint. It is valid to create labels without a reference point by passing an empty tuple to ReferencePoint, however RelatesTo must be set to *'window'* in this case (see the description of RelatesTo below).

If multiple label texts are supplied in Text, multiple labels are created. It is possible to set one reference point for all labels or a specific reference point for each label. The same applies to Position and RelatesTo.

Labels can be positioned fixed in window coordinates or relative to the projection of the reference point. The two parameters `Position` and `RelatesTo` are used for positioning.

Fixed positioning

If `RelatesTo` is set to *'window'*, the Label is shown at a fixed position in the window (or rendered image), even if the pose of the 3D scene is changed. There are two ways to specify the position. You can either supply the *[Row,Column]* coordinates of the top left corner of the label in `Position` or use one of the predefined positions:

'top_left'	*'top'*	*'top_right'*
'left'	*'center'*	*'right'*
'bottom_left'	*'bottom'*	*'bottom_right'*

Relative positioning

If `RelatesTo` is set to *'point'*, the Label is shown at a position relative to the projection of the reference point. This way, the label moves with the reference point if the pose of the 3D scene is changed. There are two ways to specify the position. You can either supply the *[Row,Column]* offset of the top left corner of the label in `Position` or use one of the predefined positions:

'top_left'	*'top'*	*'top_right'*
'left'	*'center'*	*'right'*
'bottom_left'	*'bottom'*	*'bottom_right'*

The operator `set_scene_3d_label_param` can be used to change these values and to set other parameters such as the label's color and font.

Parameters

- ▷ **`Scene3D`** (input_control) . . . . . scene_3d ⇝ *handle*
  Handle of the 3D scene.
- ▷ **`Text`** (input_control) . . . . . string(-array) ⇝ *string*
  Text of the label.
  Default: 'label'
- ▷ **`ReferencePoint`** (input_control) . . . . . point-array ⇝ *real* / integer
  Point of reference of the label.
  Number of elements: 3
- ▷ **`Position`** (input_control) . . . . . point-array ⇝ *string* / real / integer
  Position of the label.
  Default: 'top'
  List of values: `Position` ∈ {'top_left', 'top', 'top_right', 'left', 'center', 'right', 'bottom_left', 'bottom', 'bottom_right'}
- ▷ **`RelatesTo`** (input_control) . . . . . string-array ⇝ *string*
  Indicates fixed or relative positioning.
  Number of elements: 1
  Default: 'point'
  List of values: `RelatesTo` ∈ {'point', 'window'}
- ▷ **`LabelIndex`** (output_control) . . . . . integer ⇝ *integer*
  Index of the new label in the 3D scene.

Result

`add_scene_3d_label` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_scene_3d

Possible Successors

set_scene_3d_label_param

See also

remove_scene_3d_instance

Module

3D Metrology

add_scene_3d_light (: : Scene3D, LightPosition, LightKind : LightIndex)

Add a light source to a 3D scene.

add_scene_3d_light adds a new light source to the scene Scene3D and returns its index in LightIndex. The kind of the light source must be specified in LightKind. For LightKind = *'point_light'*, LightPosition is interpreted as the position of the light source. For LightKind = *'directional_light'*, LightPosition is interpreted as the vector of the directional light source.

Currently only one light source is supported, such that add_scene_3d_light overwrites the existing light source. This may be changed in future versions.

If no light source is set, a point light source at [-100.0, -100.0, 0.0] is used.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **LightPosition** (input_control) . . . . . real-array ⇝ *real* / integer
Position of the new light source.
Default: [-100.0,-100.0,0.0]

▷ **LightKind** (input_control) . . . . . string ⇝ *string*
Type of the new light source.
Default: 'point_light'
List of values: LightKind ∈ {'point_light', 'directional_light'}

▷ **LightIndex** (output_control) . . . . . integer ⇝ *integer*
Index of the new light source in the 3D scene.

Result

add_scene_3d_light returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_scene_3d

Possible Successors

set_scene_3d_light_param

See also

remove_scene_3d_light

Module

3D Metrology

clear_scene_3d (: : Scene3D :)

Delete a 3D scene and free all allocated memory.

clear_scene_3d deletes the 3D scene Scene3D.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d(-array) ⇝ *handle*
Handle of the 3D scene.

Result

clear_scene_3d returns 2 (H_MSG_TRUE) if the scene was valid and could be deleted. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- Scene3D

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

display_scene_3d

Module

3D Metrology

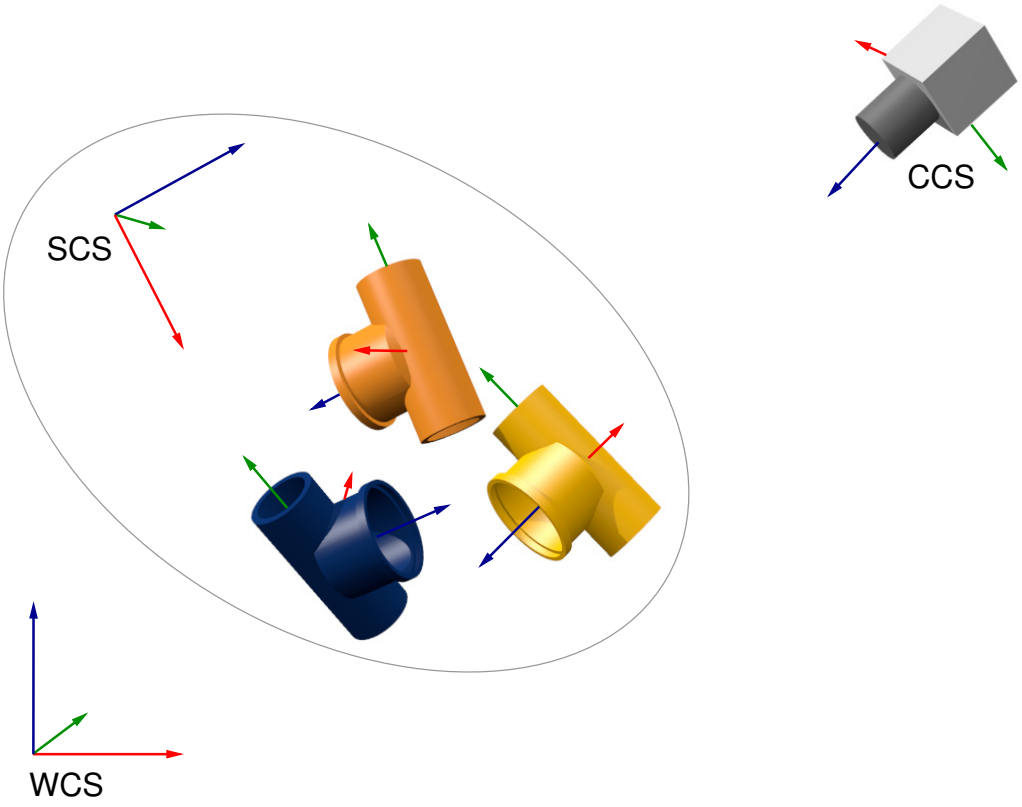
create_scene_3d (: : : Scene3D)

Create the data structure that is needed to visualize collections of 3D objects.

create_scene_3d creates a new 3D scene and returns it in Scene3D.

A 3D scene is a collection of instances of 3D object models, cameras, and light sources. Use the operators add_scene_3d_instance, add_scene_3d_camera, and add_scene_3d_light to add these objects to Scene3D. Use display_scene_3d to display a 3D scene in a window.

A pose is associated (with add_scene_3d_instance or set_scene_3d_instance_pose) to each instance in the Scene, which represents the instance's position within the scene coordinate system SCS. The pose of the scene in the world coordinate system WCS can be set with set_scene_3d_to_world_pose. The pose of the camera in the world coordinate system can be set with set_scene_3d_camera_pose and defines the camera coordinate system CCS.



The coordinate systems in a 3D scene

Parameters influencing the whole scene such as the quality of the rendering can be set with set_scene_3d_param.

Parameters

▷ **Scene3D** (output_control) ..... scene_3d ⇝ *handle*
Handle of the 3D scene.

Example

```
* Create a new scene
create_scene_3d (Scene)
* A scene needs at least one camera. The default pose
* of a camera is located at the origin. The pose can be
* changed with set_scene_3d_camera_pose.
add_scene_3d_camera (Scene, CameraParam, CameraIndex)
* Further a scene needs at least one light.
add_scene_3d_light (Scene, [42.0, 42.0, 42.0], 'point_light', LightIndex)
*
* To add an object, add_scene_3d_instance is called with a
* 3d object model and a pose. add_scene_3d returns an InstanceIndex
* which must be used to reference this instance in subsequent calls.
add_scene_3d_instance (Scene, ObjectModel3D, Pose, InstanceIndex)
* Set its color.
set_scene_3d_instance_param (Scene, InstanceIndex, 'color', 'green')
* Display the scene.
display_scene_3d (WindowHandle, Scene, CameraIndex)
clear_scene_3d (Scene)
```

Result

create_scene_3d returns 2 (H_MSG_TRUE) if the scene could be created successfully. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

add_scene_3d_instance, add_scene_3d_light, add_scene_3d_camera, set_scene_3d_param

See also

clear_scene_3d

Module

3D Metrology

display_scene_3d (: : WindowHandle, Scene3D, CameraIndex :)

Display a 3D scene.

display_scene_3d displays the 3D scene Scene3D in the window WindowHandle. The view of the camera CameraIndex is used to render the scene. Note that sometimes the aspect ratio of WindowHandle should be similar to width and height of the camera in order to obtain the wanted results. See create_scene_3d for a short example.

display_scene_3d requires OpenGL 2.1, GLSL 1.2, and the OpenGL extensions GL_EXT_framebuffer_object and GL_EXT_framebuffer_blit. Otherwise the compatibility mode with less requirements but lower quality is automatically enabled. On graphics cards with low memory the following error messages could occur if rendering in a window with high resolution:

Low level error: 'Incomplete attachment'

Unhanded Exception: 'Required framebuffer object is unsupported'

Solutions:

Set the parameter *'quality'* to *'low'* using set_scene_3d_param or use the compatibility mode to reduce the requirements of the graphics card.

The system variable (see set_system) *'opengl_compatibility_mode_enable'* can be set to *'true'* to permanently enable the visualization in compatibility mode with lower OpenGL requirements. This mode requires OpenGL 1.1. In compatibility mode the parameters *'object_index_persistence'*, *'depth_persistence'* and *'quality'* are not used.

On Linux Remote Desktop *'disp_background'* is not supported.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Scene3D** (input_control) ..... scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **CameraIndex** (input_control) ..... integer ⇝ *string* / integer
Index of the camera used to display the scene.

Result

display_scene_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`create_scene_3d`, `render_scene_3d`

Module

3D Metrology

get_display_scene_3d_info (: : WindowHandle, Scene3D, Row, Column, Information : Value)

Get the depth or the index of instances in a displayed 3D scene.

`get_display_scene_3d_info` returns information on the 3D object models in the 3D scene `Scene3D` that have been displayed with `display_scene_3d` in the window `WindowHandle`. The requested information at the positions (`Row`, `Column`) is returned in `Value`.

The following values can be queried via `Information`:

'object_index' The indices of the 3D object models that have been displayed at the positions (`Row`, `Column`). If no 3D object model was displayed at this position, -1 is returned. In order to retrieve this information, the parameter *'object_index_persistence'* must have been set to *'true'* with `set_scene_3d_param`.

'depth' The depth (i.e. the Z coordinate in the camera coordinate system) at the positions (`Row`, `Column`). If no 3D object model was displayed at one of these positions, -1.0 is returned for this position. In order to retrieve this information, the parameter *'depth_persistence'* must have been set to *'true'* with `set_scene_3d_param`.

The window coordinates `Row`, `Column` must be provided with respect to the current image part. As a consequence, these coordinates are subpixel coordinates. Given the current image part (row1,column1,row2, column2), the upper left corner corresponds to the coordinates (row1 - 0.5, col1 - 0.5). Accordingly, the bottom right corner corresponds to the coordinates (row2 - 0.5, col2 - 0.5). Use `get_mposition_sub_pix` or `get_mbutton_sub_pix` to obtain these coordinates directly.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Scene3D** (input_control) ..... scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **Row** (input_control) ..... integer(-array) ⇝ *real* / integer
Row coordinates.

▷ **Column** (input_control) ..... integer(-array) ⇝ *real* / integer
Column coordinates.

▷ **Information** (input_control) ..... string(-array) ⇝ *string*
Information.
Default: 'depth'
List of values: `Information` ∈ {'depth', 'object_index'}

▷ **Value** (output_control) ..... integer(-array) ⇝ *integer* / real
Indices or the depth of the objects at (Row,Column).

Result

`get_display_scene_3d_info` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

display_scene_3d, get_mbutton, get_mbutton_sub_pix, get_mposition, get_mposition_sub_pix

See also

display_scene_3d

Module

3D Metrology

remove_scene_3d_camera (: : Scene3D, CameraIndex :)

Remove a camera from a 3D scene.

remove_scene_3d_camera removes the camera CameraIndex from the 3D scene Scene3D.

Parameters

- ▷ **Scene3D** (input_control) ..... scene_3d ⇝ *handle*
  Handle of the 3D scene.
- ▷ **CameraIndex** (input_control) ..... integer ⇝ *integer*
  Index of the camera to remove.

Result

remove_scene_3d_camera returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

add_scene_3d_camera

Module

3D Metrology

remove_scene_3d_instance (: : Scene3D, InstanceIndex :)

Remove an object instance from a 3D scene.

remove_scene_3d_instance removes the instance InstanceIndex from the 3D scene Scene3D.

Parameters

- ▷ **Scene3D** (input_control) ..... scene_3d ⇝ *handle*
  Handle of the 3D scene.
- ▷ **InstanceIndex** (input_control) ..... integer(-array) ⇝ *integer*
  Index of the instance to remove.

Result

remove_scene_3d_instance returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

add_scene_3d_instance

Module

3D Metrology

remove_scene_3d_label (: : Scene3D, LabelIndex :)

Remove a text label from a 3D scene.

remove_scene_3d_label removes the text label LabelIndex from the 3D scene Scene3D.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **LabelIndex** (input_control) . . . . . integer(-array) ⇝ *integer*
Index of the text label to remove.

Result

remove_scene_3d_label returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

add_scene_3d_label

Module

3D Metrology

remove_scene_3d_light (: : Scene3D, LightIndex :)

Remove a light from a 3D scene.

remove_scene_3d_light removes the light LightIndex from the 3D scene Scene3D in future versions. Currently only one light source is supported.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **LightIndex** (input_control) . . . . . integer ⇝ *integer*
Light to remove.

Result

remove_scene_3d_light returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

add_scene_3d_light

Module

3D Metrology

render_scene_3d (: Image : Scene3D, CameraIndex :)

Render an image of a 3D scene.

render_scene_3d renders an image of the 3D scene Scene3D and returns the result in Image. The view of the camera CameraIndex is used to render the image.

The set_scene_3d_param parameters *'object_index_persistence'* and *'disp_background'* are ignored. The background of Image is black.

render_scene_3d requires OpenGL 2.1, GLSL 1.2, and the OpenGL extensions GL_EXT_framebuffer_object and GL_EXT_framebuffer_blit. Otherwise the compatibility mode is automatically enabled. The compatibility mode requires OpenGL 1.1.

Parameters

- ▷ **Image** (output_object) ..... (multichannel-)image ⇝ *object :* byte
  Rendered 3D scene.
- ▷ **Scene3D** (input_control) ..... scene_3d ⇝ *handle*
  Handle of the 3D scene.
- ▷ **CameraIndex** (input_control) ..... integer ⇝ *integer*
  Index of the camera used to display the scene.

Result

render_scene_3d returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

display_scene_3d

See also

create_scene_3d

Module

3D Metrology

set_scene_3d_camera_pose (: : Scene3D, CameraIndex, Pose :)

Set the pose of a camera in a 3D scene.

set_scene_3d_camera_pose sets the Pose of the camera CameraIndex in the 3D scene Scene3D.

See create_scene_3d for more details on the coordinate systems used in 3D scenes.

Parameters

- ▷ **Scene3D** (input_control) ..... scene_3d ⇝ *handle*
  Handle of the 3D scene.
- ▷ **CameraIndex** (input_control) ..... integer ⇝ *integer*
  Index of the camera.
- ▷ **Pose** (input_control) ..... pose ⇝ *real* / integer
  New pose of the camera.

Result

`set_scene_3d_camera_pose` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`add_scene_3d_camera`

See also

`set_scene_3d_to_world_pose`, `set_scene_3d_instance_pose`

Module

3D Metrology

set_scene_3d_instance_param (: : Scene3D, InstanceIndex, GenParamName, GenParamValue :)

Set parameters of an instance in a 3D scene.

`set_scene_3d_instance_param` sets parameters of the instance `InstanceIndex` in the 3D scene `Scene3D`. The name and value of a parameter must be given in `GenParamName` and `GenParamValue`. All parameters are applied to all instances.

The following values can be set:

'visible' Visibility of the 3D object models. If set to *'false'* this instance is not displayed.
List of values: *'true'*, *'false'*.
Default: *'true'*.

'attribute' Explicitly select in which way a 3D object model is visualized.
List of values: *'auto'*, *'faces'*, *'primitive'*, *'points'*, *'lines'*.
Default: *'auto'*.

'color': Color of the 3D object model. The available colors can be queried with the operator `query_color`. In addition, the color may be specified as an RGB triplet in the form *'#rrggbb'*, where *'rr'*, *'gg'*, and *'bb'* are hexadecimal numbers between *'00'* and *'ff'*, respectively.
Suggested values: *'red'*, *'green'*.
Default: *'white'*

'alpha': Transparency of the 3D object models. Displaying 3D object models with transparency set to less than 1.0 may significantly increase the runtime of `display_scene_3d` and `render_scene_3d`.
Value range: floating point value between 0.0 (fully transparent) and 1.0 (fully opaque).
Default: 1.0

'disp_pose': Flag, if the pose of the 3D object models should be visualized.
List of values: *'true'*, *'false'*.
Default: *'false'*.

'disp_lines': Flag, if the contours of the 3D object models' polygons should be displayed.
List of values: *'true'*, *'false'*.
Default: *'false'*.

'disp_normals': Flag, if the surface normals of the 3D object models should be visualized.
List of values: *'true'*, *'false'*.
Default: *'false'*.

'line_color': Color of the lines if *'disp_lines'* is set to *'true'*. The available colors can be queried with the operator query_color. In addition, the color may be specified as an RGB triplet in the form *'#rrggbb'*, where *'rr'*, *'gg'*, and *'bb'* are hexadecimal numbers.

Suggested values: *'red'*, *'green'*.

Default: The value of *'color'*.

'line_width': Sets the width of lines in pixel.

Default: *1.0*

'normal_color': Color of the visualized normals if *'disp_normals'* is set to *'true'*. The available colors can be queried with the operator query_color. In addition, the color may be specified as an RGB triplet in the form *'#rrggbb'*, where *'rr'*, *'gg'*, and *'bb'* are hexadecimal numbers.

Suggested values: *'red'*, *'green'*.

Default: The value of *'color'*

'point_size': Sets the diameter of the points in pixel.

Default: *3.5*.

'lut': Sets the LUT that transforms the values of the attribute set with *'color_attrib'* into a color.

See set_lut for available LUTs. If *'lut'* is set to anything but *'default'*, *'color'* is ignored.

Default: *'default'*.

'color_attrib': Name of a point attribute that is used for false color visualization.

If an attribute is set, the color of the displayed 3D points is determined by the point's attribute value and the currently set LUT (see *'lut'*). This way, it is possible to visualize attributes in false colors.

Example: If *'color_attrib'* is set to *'coord_z'*, and *'lut'* is set to *'color1'*, the z-coordinates will be color coded from red to blue.

If *'lut'* is set to *'default'*, the attribute values are used to scale the color that was set by the parameter *'color'*.

If *'lut'* is set to a different value, the attribute values of all points are internally scaled to the interval [0,255] and used as input value for the LUT function.

The mapping is also controlled by the parameters *'color_attrib_start'* and *'color_attrib_end'* (see below).

If faces are displayed, their color is interpolated between the color of the corner points.

Suggested values: *'none'*, *'&distance'*, *'coord_x'*, *'coord_y'*, *'coord_z'*, user defined point attributes, or any other point attribute available.

Default: *'none'*.

'color_attrib_start',*'color_attrib_end'*: The range of interest of the values of the attribute set with *'color_attrib'*.

The attribute values between *'color_attrib_start'* and *'color_attrib_end'* are scaled to the start and end of the selected LUT. Attribute values outside the selected range are clipped. This allows to use a fixed color mapping which will not be distorted by outliers.

If set to *'auto'*, the minimum attribute value is mapped to the start of the LUT, the maximum is mapped to the end of the LUT, *except* if *'color_attrib'* is *'normal_x'*, *'normal_y'*, or *'normal_z'*. In this case, start and end are automatically set to -1 and 1.

It is possible to enter start value that is higher than the end value. This will in effect flip the used LUT.

Suggested values: *0*, *0.1*, *1*, *100*, *255*, *'auto'*.

Default: *'auto'*.

'red_channel_attrib', *'green_channel_attrib'*, *'blue_channel_attrib'*: Name of a point attribute that is used for the red, green, or blue color channel.

This is most useful when used with a group of three connected attributes, like RGB colors or normal vectors. This way it is possible to display points in colored texture, e.g., display the object model with overlayed RGB-sensor data, or display point normals in false colors.

To display only a single attribute in false colors, please use *'color_attrib'* (see above).

By default, the attribute values are assumed to lie between 0 and 255. If the attributes have a different range, you additionally have to set the parameters *'rgb_channel_attrib_start'* and *'rgb_channel_attrib_end'* (see below).

If only 1 or 2 channels are set, the remaining channels use the RGB value of the color set with *'color'*.

If faces are displayed, their color is interpolated between the color of the corner points.

Suggested values: *'none'*, *'&red'*, *'&green'*, *'&blue'*, *'normal_x'*, *'normal_y'*, *'normal_z'*, user defined point attributes, or any other point attribute available.

Default: *'none'*.

'rgb_channel_attrib_start', *'rgb_channel_attrib_end'*: The range of interest of the values of attributes set with *'red_channel_attrib'*, *'green_channel_attrib'*, and *'blue_channel_attrib'*.

These parameters define the value range that is scaled to the full RGB channels. This is useful, if the input attribute values are not in the interval [0,255].

If set to *'auto'*, the minimum attribute value is mapped to 0, the maximum is mapped to 255, *except* if the attribute is *'normal_x'*, *'normal_y'*, or *'normal_z'*. In this case, start and end are automatically set to -1 and 1.

It is possible to enter start value that is higher than the end value. This will in effect invert the displayed RGB colors.

The range can be set for the channels individually by replacing *'rgb'* in the parameter name with the channel name, e.g., *'green_channel_attrib_start'*.

Suggested values: *'auto'*, *0*, *0.1*, *1*, *100*, *255*.

Default: *0*, *255*.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **InstanceIndex** (input_control) . . . . . integer(-array) ⇝ *integer*
Index of the instance.

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of the generic parameters.
Default: 'color'
List of values: `GenParamName` ∈ {'alpha', 'attribute', 'color', 'disp_lines', 'disp_pose', 'disp_normals', 'line_color', 'line_width', 'normal_color', 'color_attrib', 'red_channel_attrib', 'green_channel_attrib', 'blue_channel_attrib', 'rgb_channel_attrib_start', 'rgb_channel_attrib_end', 'color_attrib_start', 'color_attrib_end', 'lut', 'visible', 'point_size'}

▷ **GenParamValue** (input_control) . . . . . string-array ⇝ *string* / integer / real
Values of the generic parameters.
Default: 'green'
Suggested values: `GenParamValue` ∈ {'true', 'false', 'coord_x', 'coord_y', 'coord_z', 'normal_x', 'normal_y', 'normal_z', 'red', 'green', 'blue', 'auto', 'faces', 'primitive', 'points', 'lines'}

Result

`set_scene_3d_instance_param` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`add_scene_3d_instance`

See also

`set_scene_3d_param`

Module

3D Metrology

set_scene_3d_instance_pose (: : Scene3D, InstanceIndex, Pose :)

Set the pose of an instance in a 3D scene.

`set_scene_3d_instance_pose` sets the pose of the instance `InstanceIndex` in the 3D scene `Scene3D` to `Pose`. It's possible to set one pose for multiple instances or one pose for each instance.

See the documentation of create_scene_3d for more details on the coordinate systems used in 3D scenes.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **InstanceIndex** (input_control) . . . . . integer(-array) ⇝ *integer*
Index of the instance.

▷ **Pose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
New pose of the instance.

Result

set_scene_3d_instance_pose returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_scene_3d_instance

See also

set_scene_3d_to_world_pose, set_scene_3d_camera_pose

Module

3D Metrology

set_scene_3d_label_param (: : Scene3D, LabelIndex, GenParamName, GenParamValue :)

Set parameters of a text label in a 3D scene.

set_scene_3d_label_param sets parameters of the label LabelIndex in the 3D scene Scene3D. The name and value of a parameter must be given in GenParamName and GenParamValue. It is only possible to provide one GenParamName,GenParamValue pair but multiple labels. The parameter is applied to all labels in LabelIndex.

The following values can be set:

'reference_point': Coordinates of the reference point, as a tuple of *[X, Y, Z]* coordinates. To remove the current reference point, it is possible to pass an empty tuple for *'reference_point'*. This is only possible if *'relates_to'* is *'window'*.

'position': Position (relative or absolute depending on the parameter *'relates_to'*) of the label, as a tuple of *[Row, Column]* coordinates or one of the following predefined positions.

List of values: *'top_left'*, *'top'*, *'top_right'*, *'left'*, *'center'*, *'right'*, *'bottom_left'*, *'bottom'*, or *'bottom_right'*.

'relates_to': Relation of the position. May be *'window'* (fixed positioning in window coordinates) or *'point'* (positioning relative to the reference point). *'relates_to'* cannot be set to *'point'* if the reference point has been set to an empty tuple.

List of values: *'window'* , *'point'*.

Default: *'window'*.

'text': Text of the label. Multiple Lines must be separated with *'\n'*.

'font': Font of the text label.

Suggested values: Available fonts can be queried using query_font.

Default: Font of the first window the scene is displayed in, default system font otherwise.

'text_color': Color of the text on the label. The available colors can be queried with the operator query_color. In addition, the color may be specified as an RGB triplet in the form *'#rrggbb'*, where *'rr'*, *'gg'*, and *'bb'* are hexadecimal numbers between *'00'* and *'ff'*, respectively.

Suggested values: *'red'*, *'green'*.

Default: *'white'*.

'alpha': Transparency of the text label's background.

Value range: floating point value between 0.0 (fully transparent) and 1.0 (fully opaque).

Default: *1.0*.

'color': Color of the label itself. The available colors can be queried with the operator query_color. In addition, the color may be specified as an RGB triplet in the form *'#rrggbb'*, where *'rr'*, *'gg'*, and *'bb'* are hexadecimal numbers between *'00'* and *'ff'*, respectively.

Suggested values: *'red'*, *'green'*.

Default: *'gray'*.

'disp_background': Flag that determines if the background of the label should be visualized.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'disp_connecting_line': Flag that determines if the connecting line between the label and reference 3D point should be visualized.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'visible': Sets the visibility of the label. If set to *'if_point_is_visible'*, the label is only visible if its reference point is visible. If the compatibility mode is enabled or no reference point is set, *'if_point_is_visible'* is equal to *'true'*.

List of values: *'true'*, *'false'*, *'if_point_is_visible'*.

Default: *'true'*.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **LabelIndex** (input_control) . . . . . integer(-array) ⇝ *integer*
Index of the text label.

▷ **GenParamName** (input_control) . . . . . string ⇝ *string*
Names of the generic parameters.
Default: 'color'
List of values: GenParamName ∈ {'reference_point', 'position', 'relates_to', 'color', 'text_color', 'font', 'text', 'disp_connecting_line', 'disp_background', 'visible'}

▷ **GenParamValue** (input_control) . . . . . string-array ⇝ *string* / integer / real
Values of the generic parameters.
Default: 'red'
List of values: GenParamValue ∈ {'red', 'green', 'blue', 'window', 'point', 'top_left', 'top', 'top_right', 'left', 'center', 'right', 'bottom_left', 'bottom', 'bottom_right'}

Result

set_scene_3d_label_param returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_scene_3d_label

Module

3D Metrology

set_scene_3d_light_param (: : Scene3D, LightIndex, GenParamName, GenParamValue :)

Set parameters of a light in a 3D scene.

set_scene_3d_light_param sets parameters of a light source in the 3D scene Scene3D. The name and value of a parameter must be given in GenParamName and GenParamValue. The following parameters can be set:

'ambient': Ambient part of the light source. Must be given as a tuple of three numbers.
Default: *[0.2, 0.2, 0.2]*.

'diffuse': Diffuse part of the light source. Must be given as a tuple of three numbers.
Default: *[0.8, 0.8, 0.8]*.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **LightIndex** (input_control) . . . . . integer ⇝ *integer*
Index of the light source.

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Names of the generic parameters.
Default: 'ambient'
List of values: GenParamName ∈ {'ambient', 'diffuse'}

▷ **GenParamValue** (input_control) . . . . . string(-array) ⇝ *string* / integer / real
Values of the generic parameters.
Default: [0.2,0.2,0.2]
Suggested values: GenParamValue ∈ {[0.2,0.2,0.2]}

Result

set_scene_3d_light_param returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_scene_3d_instance

Module

3D Metrology

set_scene_3d_param (: : Scene3D, GenParamName, GenParamValue :)

Set parameters of a 3D scene.

set_scene_3d_param sets parameters of the 3D scene Scene3D. The name and value of a parameter must be given in GenParamName and GenParamValue.

All parameters that can be set with set_scene_3d_instance_param can be set with set_scene_3d_param for all instances of a scene.

Additionally, the following parameters can be set:

'disp_background': Flag, if the current window content should be used as background.
List of values: *'true'*, *'false'*.
Default: *'false'*.

'colored': Set the colors of all currently added instances to different colors. The value of this parameter defines the number of colors that are used.
List of values: *3*, *6*, *12*.
Default: All objects are displayed white.

'object_index_persistence': Must be set to *'true'* to enable the object index query with `get_display_scene_3d_info`.
List of values: *'true'*, *'false'*.
Default: *'false'*.

'depth_persistence': Must be set to *'true'* to enable the depth query with `get_display_scene_3d_info`.
List of values: *'true'*, *'false'*.
Default: *'false'*.

'quality': Must be set to *'low'* to enable faster rendering without anti aliasing.
List of values: *'low'*, *'high'*.
Default: *'high'*.

'compatibility_mode_enable': Enforce the usage of the fallback mode to OpenGL 1.1.
List of values: *'true'*, *'false'*.
Default: *'false'*.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **GenParamName** (input_control) . . . . . string ⇝ *string*
Names of the generic parameters.
Default: 'quality'
List of values: `GenParamName` ∈ {'alpha', 'attribute', 'color', 'colored', 'depth_persistence', 'disp_background', 'disp_lines', 'disp_pose', 'disp_normals', 'line_color', 'line_width', 'normal_color', 'object_index_persistence', 'point_size', 'quality', 'compatibility_mode_enable', 'color_attrib', 'color_attrib_start', 'color_attrib_end', 'red_channel_attrib', 'green_channel_attrib', 'blue_channel_attrib', 'rgb_channel_attrib_start', 'rgb_channel_attrib_end', 'lut', 'visible'}

▷ **GenParamValue** (input_control) . . . . . string ⇝ *string* / integer / real
Values of the generic parameters.
Default: 'high'
Suggested values: `GenParamValue` ∈ {'true', 'false', 'high', 'low', 'coord_x', 'coord_y', 'coord_z', 'normal_x', 'normal_y', 'normal_z', 'red', 'green', 'blue', 'auto', 'faces', 'primitive', 'points', 'lines'}

Result

`set_scene_3d_param` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_scene_3d`

See also

`set_scene_3d_instance_param`

Module

3D Metrology

set_scene_3d_to_world_pose (: : Scene3D, ToWorldPose :)

Set the pose of a 3D scene.

set_scene_3d_to_world_pose sets the pose of the 3D scene Scene3D to ToWorldPose.

See the documentation of create_scene_3d for more details on the coordinate systems used in 3D scenes.

Parameters

▷ **Scene3D** (input_control) . . . . . scene_3d ⇝ *handle*
Handle of the 3D scene.

▷ **ToWorldPose** (input_control) . . . . . pose ⇝ *real* / integer
New pose of the 3D scene.

Result

set_scene_3d_to_world_pose returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_scene_3d

See also

set_scene_3d_instance_pose

Module

3D Metrology

13.2 Drawing

This chapter describes operators that allow the user to manually draw geometric shapes. They require mouse interaction and in general block the application until the mouse interaction is finished. For non-blocking interactive creation of geometric shapes, HALCON also provides drawing objects with the operators listed in Graphics / Object.

General Objectives

The draw operators draw_region, draw_xld, draw_xld_mod, draw_nurbs, draw_nurbs_mod, draw_nurbs_interp, draw_nurbs_interp_mod, as well as drag_region1, drag_region2 and drag_region3 return iconic objects. All other draw operators return the geometric parameters that are needed to create regions or contours in the respective shape (see table). Most draw operators have a modifying variant that allows the user to specify the initial parameters of the shown shape.

Drawing in Buffer Windows

Because the draw operators rely on mouse interaction, they generally do not work when used with buffer windows (see open_window), so you should use drawing objects instead. Nevertheless some operators (see table) can be used in buffer windows. These operators are controlled through a mouse state. In order to use these operators (and to not cause a deadlock) the application has to provide the mouse state with the operators send_mouse_double_click_event, send_mouse_down_event, send_mouse_drag_event and send_mouse_up_event as well as the state of the modifier keys (also by using send_mouse_drag_event), whereas the draw operators themselves must be run in another thread. While using the draw operator, the image section must not be changed.

Drawing Operators	Output	Generate Object	Works in Buffer Windows
draw_point	Row, Column	gen_cross_contour_xld	yes
draw_line	Row1, Column1, Row2, Column2	gen_contour_polygon_xld	yes
draw_rectangle1	Row1, Column1, Row2, Column2	gen_rectangle1	yes
draw_rectangle2	Row, Column, Phi, Length1, Length2	gen_rectangle2, gen_rectangle2_contour_xld	yes
draw_circle	Row, Column, Radius	gen_circle, gen_circle_contour_xld	yes
draw_ellipse	Row, Column, Phi, Radius1, Radius2	gen_ellipse, gen_ellipse_contour_xld	yes
draw_xld	Contour	-	yes
draw_region	Region	-	yes
draw_polygon	Region	-	yes
draw_nurbs	Contour, Rows, Columns, Weights	-	no
draw_nurbs_interp	Contour, Rows, Columns, Knots, Tangents	-	no
drag_region1	Region	-	no
drag_region2	Region	-	no
drag_region3	Region	-	no

drag_region1 (SourceRegion : DestinationRegion : WindowHandle :)

Interactive moving of a region.

This operator does not work in an HDevelop graphics window opened with dev_open_window.

drag_region1 is used to move a region on the display by mouse. Calling drag_region1 turns the region visible as soon as the left mouse button is pressed. Therefore the region's edges are displayed only. As representation mode the mode 'not' (see set_draw) is used during procedure's permanence. During the movement the cursor resides in the region's barycenter. If you move the mouse with pressed left mouse button, the depicted region follows - delayed - this movement. If you press the right mouse button you terminate drag_region1. The depicted region disappears from the display. Output is a region which corresponds to the last position on the display. You may pass even several regions at once. The operator affine_trans_image moves the gray values.

Attention

Gray values of regions are not moved. With moving the input region it is not sure whether the gray values of the output regions are filled reasonable. This may occur if the gray values of the input regions do not comprise the whole image.

Parameters

- ▷ **SourceRegion** (input_object) ..... region-array ⇝ *object*
  Regions to move.
- ▷ **DestinationRegion** (output_object) ..... region-array ⇝ *object*
  Moved Regions.
- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.

Example

```
draw_region(Obj,WindowHandle)
dev_set_color ('green')
drag_region1(Obj,New,WindowHandle)
dev_display(New)
get_region_runs (Obj, Rows1, ColumnBegins1, ColumnEnds1)
get_region_runs (New, Rows2, ColumnBegins2, ColumnEnds2)
Row1 := Rows1[0]
Column1 := ColumnBegins1[0]
Row2 := Rows2[0]
Column2 := ColumnBegins2[0]
dev_set_color ('white')
gen_arrow_contour_xld (Arrow, Row1,Column1,Row2,Column2, 5, 5)
dev_display (Arrow)
```

Result

drag_region1 returns 2 (H_MSG_TRUE), if a region is entered, the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception is raised. You may determine the behavior after an empty input with set_system(::'no_object_result',<Result>:).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_region, set_colored, set_line_width, set_draw, set_insert

Alternatives

get_mposition, move_region

See also

set_insert, set_draw, affine_trans_image

Module

Foundation

drag_region2 (SourceRegion : DestinationRegion : WindowHandle, Row, Column :)

Interactive movement of a region with fixpoint specification.

This operator does not work in an HDevelop graphics window opened with dev_open_window.

You use drag_region2 to move a region on the display by mouse. It corresponds to the operator drag_region1 with the difference, that the position of the mouse cursor can be determined.

Attention

Gray values of the regions are not moved. With moving the input region it is not sure whether the gray values of the output regions are filled reasonable. This may occur if the gray values of the input regions do not comprise the whole image.

Parameters

▷ **SourceRegion** (input_object) ..... region-array ⇝ *object*
Regions to move.

▷ **DestinationRegion** (output_object) ..... region-array ⇝ *object*
Moved regions.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Row index of the reference point.
Default: 100
Suggested values: Row ∈ {0, 64, 128, 256, 512}
Value range: 0 ≤ Row ≤ 1024

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column index of the reference point.
Default: 100
Suggested values: Column ∈ {0, 64, 128, 256, 512}
Value range: 0 ≤ Column ≤ 1024

Result

drag_region2 returns 2 (H_MSG_TRUE), if a region is entered, the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception is raised. You may determine the behavior after an empty input with set_system(::'no_object_result',<Result>:).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_region, set_colored, set_line_width, set_draw, set_insert, affine_trans_image

Alternatives

get_mposition, move_region, drag_region1, drag_region3

See also

set_insert, set_draw, affine_trans_image

Module

Foundation

drag_region3 (SourceRegion, MaskRegion : DestinationRegion : WindowHandle, Row, Column :)

Interactive movement of a region with restriction of positions.

This operator does not work in an HDevelop graphics window opened with dev_open_window.

You use drag_region3 to move a region on the display by mouse. It corresponds to the operator drag_region2 with the enhancement, that all points are specified which can be entered by mouse. If you move the mouse outside of this area (MaskRegion), the region on the point with the smallest distance inside MaskRegion will be displayed.

Attention

The region's gray values are not moved. With moving the input region it is not sure whether the gray values of the output regions are filled reasonable. This may occur if the gray values of the input regions do not comprise the whole image.

Parameters

▷ **SourceRegion** (input_object) . . . . . region-array ⇝ *object*
Regions to move.

▷ **MaskRegion** (input_object) . . . . . region-array ⇝ *object*
Points on which it is allowed for a region to move.

▷ **DestinationRegion** (output_object) . . . . . region-array ⇝ *object*
Moved regions.

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row index of the reference point.
Default: 100
Suggested values: Row ∈ {0, 64, 128, 256, 512}
Value range: 0 ≤ Row ≤ 1024

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column index of the reference point.
Default: 100
Suggested values: Column ∈ {0, 64, 128, 256, 512}
Value range: 0 ≤ Column ≤ 1024

Result

drag_region3 returns 2 (H_MSG_TRUE), if a region is entered, if the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception is raised. You may determine the behavior after an empty input with set_system(::'no_object_result',<Result>:).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, get_mposition

Possible Successors

reduce_domain, disp_region, set_colored, set_line_width, set_draw, set_insert, affine_trans_image

Alternatives

get_mposition, move_region, drag_region1, drag_region2

See also

set_insert, set_draw, affine_trans_image

Module

Foundation

draw_circle (: : WindowHandle : Row, Column, Radius)

Interactive drawing of a circle.

draw_circle produces the parameter for a circle created interactive by the user in the window.

To create a circle you have to press the mouse button at the location which is used as the center of that circle. While keeping the mouse button pressed, the Radius's length can be modified through moving the mouse. After another mouse click in the created circle center you can move it. A clicking close to the circular arc you can modify the Radius of the circle. Pressing the right mouse button terminates the procedure. After terminating the procedure the circle is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row** (output_control) ..... circle.center.y ⇝ *real*
Barycenter's row index.

▷ **Column** (output_control) ..... circle.center.x ⇝ *real*
Barycenter's column index.

▷ **Radius** (output_control) ..... circle.radius ⇝ *real*
Circle's radius.

Example

```
read_image(Image,'monkey')
draw_circle(WindowHandle,Row,Column,Radius)
gen_circle(Circle,Row,Column,Radius)
reduce_domain(Image,Circle,ImageReduced)
invert_image (ImageReduced, ImageInvert)
dev_display (ImageInvert)
```

Result

`draw_circle` returns `2` (`H_MSG_TRUE`) if the window is valid and the needed drawing mode (see `set_insert`) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`reduce_domain`, `disp_region`, `set_colored`, `set_line_width`, `set_draw`, `set_insert`

Alternatives

`draw_circle_mod`, `draw_ellipse`, `draw_region`

See also

`gen_circle`, `draw_rectangle1`, `draw_rectangle2`, `draw_polygon`, `set_insert`

Module

Foundation

draw_circle_mod (: : WindowHandle, RowIn, ColumnIn, RadiusIn : Row, Column, Radius)

Interactive drawing of a circle.

`draw_circle_mod` produces the parameter for a circle created interactive by the user in the window.

To create a circle are expected the coordinates `RowIn` and `ColumnIn` of the center of a circle with radius `RadiusIn`. After another mouse click in the created circle center you can move it. A clicking close to the circular arc you can modify the `Radius` of the circle. Pressing the right mouse button terminates the procedure. After terminating the procedure the circle is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **RowIn** (input_control) ..... circle.center.y ⇝ *real*
Row index of the center.

▷ **ColumnIn** (input_control) ..... circle.center.x ⇝ *real*
Column index of the center.

▷ **RadiusIn** (input_control) ..... circle.radius ⇝ *real*
Radius of the circle.

▷ **Row** (output_control) ..... circle.center.y ⇝ *real*
Row index of the center.

▷ **Column** (output_control) ..... circle.center.x ⇝ *real*
Column index of the center.

▷ **Radius** (output_control) ..... circle.radius ⇝ *real*
Circle's radius.

Example

```
read_image(Image,'monkey')
draw_circle_mod(WindowHandle,20,20,15,Row,Column,Radius)
gen_circle(Circle,Row,Column,Radius)
reduce_domain(Image,Circle,ImageReduced)
invert_image (ImageReduced, ImageInvert)
dev_display (ImageInvert)
```

Result

`draw_circle_mod` returns `2` (`H_MSG_TRUE`) if the window is valid and the needed drawing mode (see `set_insert`) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`reduce_domain`, `disp_region`, `set_colored`, `set_line_width`, `set_draw`, `set_insert`

Alternatives

`draw_circle`, `draw_ellipse`, `draw_region`

See also

`gen_circle`, `draw_rectangle1`, `draw_rectangle2`, `draw_polygon`, `set_insert`

Module

Foundation

draw_ellipse (: : WindowHandle : Row, Column, Phi, Radius1, Radius2)

Interactive drawing of an ellipse.

`draw_ellipse` returns the parameter for any orientated ellipse, which has been created interactively by the user in the window.

The created ellipse is described by its center, `Row` and `Column`, its orientation, `Phi`, and its two half axes, `Radius1` and `Radius2`.

To create an ellipse you have to determine the center of the ellipse with the left mouse button. Keeping the button pressed determines the length (Radius1) and the orientation (Phi) of the first half axis. In doing so a temporary default length for the second half axis is assumed, which may be modified afterwards on demand. After another mouse click in the center of the created ellipse you can move it. A mouse click close to a vertex "grips" it to modify the length of the appropriate half axis. You may modify the orientation only, if a vertex of the first half axis is gripped.

Pressing the right mouse button terminates the procedure. After terminating the procedure the ellipse is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Row** (output_control) ..... ellipse.center.y ⇝ *real*
  Row index of the center.
- ▷ **Column** (output_control) ..... ellipse.center.x ⇝ *real*
  Column index of the center.
- ▷ **Phi** (output_control) ..... ellipse.angle.rad ⇝ *real*
  Orientation of the first half axis in radians.
- ▷ **Radius1** (output_control) ..... ellipse.radius1 ⇝ *real*
  First half axis.
- ▷ **Radius2** (output_control) ..... ellipse.radius2 ⇝ *real*
  Second half axis.

Example

```
read_image(Image,'monkey')
draw_ellipse(WindowHandle,Row,Column,Phi,Radius1,Radius2)
gen_ellipse(Ellipse,Row,Column,Phi,Radius1,Radius2)
reduce_domain(Image,Ellipse,GrayEllipse)
sobel_amp(GrayEllipse,Sobel,'sum_abs',3)
dev_display(Sobel)
```

Result

draw_ellipse returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_region, set_colored, set_line_width, set_draw, set_insert

Alternatives

draw_ellipse_mod, draw_circle, draw_region

See also

gen_ellipse, draw_rectangle1, draw_rectangle2, draw_polygon, set_insert

Module

Foundation

draw_ellipse_mod (: : WindowHandle, RowIn, ColumnIn, PhiIn, Radius1In, Radius2In : Row, Column, Phi, Radius1, Radius2)

Interactive drawing of an ellipse.

draw_ellipse_mod returns the parameter for any orientated ellipse, which has been created interactively by the user in the window.

The created ellipse is described by its center, its two half axes and the angle between the first half axis and the horizontal coordinate axis.

To create an ellipse are expected the parameters RowIn, ColumnIn,PhiIn,Radius1In,Radius2In. Keeping the button pressed determines the length (Radius1) and the orientation (Phi) of the first half axis. In doing so a temporary default length for the second half axis is assumed, which may be modified afterwards on demand. After another mouse click in the center of the created ellipse you can move it. A mouse click close to a vertex "grips" it to modify the length of the appropriate half axis. You may modify the orientation only, if a vertex of the first half axis is gripped.

Pressing the right mouse button terminates the procedure. After terminating the procedure the ellipse is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **RowIn** (input_control) ..... ellipse.center.y ⇝ *real*
  Row index of the center.
- ▷ **ColumnIn** (input_control) ..... ellipse.center.x ⇝ *real*
  Column index of the center.
- ▷ **PhiIn** (input_control) ..... ellipse.angle.rad ⇝ *real*
  Orientation of the bigger half axis in radians.
- ▷ **Radius1In** (input_control) ..... ellipse.radius1 ⇝ *real*
  Bigger half axis.
- ▷ **Radius2In** (input_control) ..... ellipse.radius2 ⇝ *real*
  Smaller half axis.
- ▷ **Row** (output_control) ..... ellipse.center.y ⇝ *real*
  Row index of the center.
- ▷ **Column** (output_control) ..... ellipse.center.x ⇝ *real*
  Column index of the center.
- ▷ **Phi** (output_control) ..... ellipse.angle.rad ⇝ *real*
  Orientation of the first half axis in radians.
- ▷ **Radius1** (output_control) ..... ellipse.radius1 ⇝ *real*
  First half axis.
- ▷ **Radius2** (output_control) ..... ellipse.radius2 ⇝ *real*
  Second half axis.

Example

```
read_image(Image,'monkey')
draw_ellipse_mod(WindowHandle,RowIn,ColumnIn,PhiIn,Radius1In,Radius2In,\
                 Row,Column,Phi,Radius1,Radius2)
gen_ellipse(Ellipse,Row,Column,Phi,Radius1,Radius2)
reduce_domain(Image,Ellipse,GrayEllipse)
sobel_amp(GrayEllipse,Sobel,'sum_abs',3)
dev_display(Sobel)
```

Result

draw_ellipse_mod returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_region, set_colored, set_line_width, set_draw, set_insert

Alternatives

draw_ellipse, draw_circle, draw_region

See also

gen_ellipse, draw_rectangle1, draw_rectangle2, draw_polygon, set_insert

Module

Foundation

draw_line (: : WindowHandle : Row1, Column1, Row2, Column2)

Draw a line.

draw_line returns the parameter for a line, which has been created interactively by the user in the window.

To create a line you have to press the left mouse button determining a start point of the line. While keeping the button pressed you may "drag" the line in any direction. After another mouse click in the middle of the created line you can move it. If you click on one end point of the created line, you may move this point. Pressing the right mouse button terminates the procedure.

After terminating the procedure the line is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row1** (output_control) ..... line.begin.y ⇝ *real*
Row index of the first point of the line.

▷ **Column1** (output_control) ..... line.begin.x ⇝ *real*
Column index of the first point of the line.

▷ **Row2** (output_control) ..... line.end.y ⇝ *real*
Row index of the second point of the line.

▷ **Column2** (output_control) ..... line.end.x ⇝ *real*
Column index of the second point of the line.

Example

```
draw_line(WindowHandle,Row1,Column1,Row2,Column2)
gen_contour_polygon_xld (Line, [Row1,Row2], [Column1,Column2])
dev_display (Line)
```

Result

draw_line returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_line, set_colored, set_line_width, set_draw, set_insert

See also

draw_line_mod, gen_rectangle1, draw_circle, draw_ellipse, set_insert

Module

Foundation

draw_line_mod (: : WindowHandle, Row1In, Column1In, Row2In, Column2In : Row1, Column1, Row2, Column2)

Draw a line.

draw_line_mod returns the parameter for a line, which has been created interactively by the user in the window.

To create a line are expected the coordinates of the start point Row1In, Column1In and of the end point Row2In,Column2In. If you click on one end point of the created line, you may move this point. After another mouse click in the middle of the created line you can move it.

Pressing the right mouse button terminates the procedure.

After terminating the procedure the line is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row1In** (input_control) ..... line.begin.y ⇝ *real*
Row index of the first point of the line.

▷ **Column1In** (input_control) ..... line.begin.x ⇝ *real*
Column index of the first point of the line.

▷ **Row2In** (input_control) ..... line.end.y ⇝ *real*
Row index of the second point of the line.

▷ **Column2In** (input_control) ..... line.end.x ⇝ *real*
Column index of the second point of the line.

▷ **Row1** (output_control) ..... line.begin.y ⇝ *real*
Row index of the first point of the line.

▷ **Column1** (output_control) ..... line.begin.x ⇝ *real*
Column index of the first point of the line.

▷ **Row2** (output_control) ..... line.end.y ⇝ *real*
Row index of the second point of the line.

▷ **Column2** (output_control) ..... line.end.x ⇝ *real*
Column index of the second point of the line.

Example

```
draw_line_mod(WindowHandle,10,20,55,124,Row1,Column1,Row2,Column2)
gen_contour_polygon_xld (Line, [Row1,Row2], [Column1,Column2])
dev_display (Line)
```

Result

draw_line_mod returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_line, set_colored, set_line_width, set_draw, set_insert

Alternatives

draw_line, draw_ellipse, draw_region

See also

gen_circle, draw_rectangle1, draw_rectangle2

Module

Foundation

draw_nurbs (: ContOut : WindowHandle, Rotate, Move, Scale, KeepRatio, Degree : Rows, Cols, Weights)

Interactive drawing of a NURBS curve.

This operator does not work in an HDevelop graphics window opened with dev_open_window.

draw_nurbs returns the contour ContOut and control information (Rows, Cols, and Weights) of a NURBS curve of degree Degree, which has been created interactively by the user in the window WindowHandle. For additional information concerning NURBS curves, see the documentation of gen_contour_nurbs_xld. To use the control information Rows, Cols, and Weights in a subsequent call to the operator gen_contour_nurbs_xld, the knot vector Knots should be set to *'auto'*.

The NURBS curve is created and manipulated by the means of its control polygon. By contrast, using the operator draw_nurbs_interp, it is possible to create a NURBS curve that interpolates points specified by the user. Directly after calling draw_nurbs, you can add control points by clicking with the left mouse button in the window at the desired positions.

When there are three control points or more, the first and the last point will be marked with an additional square. By clicking on them you can close the curve or open it again. You delete the point appended last by pressing the Ctrl key.

As soon as the number of control points exceeds Degree, the NURBS curve given by the specified control polygon and weight vector is displayed in addition to the control polygon.

The control point which was handled last is surrounded by a circle representing its weight. By simply dragging the circle you can increase or decrease the weight of this control point.

Existing control points can be moved by dragging them with the mouse. Further new points on the control polygon (to refine the control polygon) can be inserted by a left click on the desired position on the control polygon.

By pressing the Shift key, you can switch into the transformation mode. In this mode you can rotate, move, and scale the contour as a whole, but only if you set the parameters Rotate, Move, and Scale, respectively, to *'true'*. Instead of the pick points and the control polygon, 3 symbols are displayed with the contour: a cross in the middle and an arrow to the right if Rotate is set to *'true'*, and a double-headed arrow to the upper right if Scale is set to *'true'*.

You can

- move the curve by clicking the left mouse button on the cross in the center and then dragging it to the new position,

- rotate it by clicking with the left mouse button on the arrow and then dragging it, till the curve has the right direction, and
- scale it by dragging the double arrow. To keep the ratio the parameter `KeepRatio` has to be set to *'true'*.

By pressing the Shift key again you can switch back to the edit mode. Pressing the right mouse button terminates the procedure.

The appearance of the curve while drawing is determined by the line width, size, and color set via the operators `set_color`, `set_colored`, `set_line_width`, and `set_line_style`. The control polygon and all handles are displayed in the second color set by `set_color` or `set_colored`. Their line width is fixed to 1 and their line style is fixed to a drawn-through line.

Parameters

▷ **`ContOut`** (output_object) . . . . . xld_cont ⇝ *object*
Contour approximating the NURBS curve.

▷ **`WindowHandle`** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **`Rotate`** (input_control) . . . . . string ⇝ *string*
Enable rotation?
Default: 'true'
List of values: `Rotate` ∈ {'true', 'false'}

▷ **`Move`** (input_control) . . . . . string ⇝ *string*
Enable moving?
Default: 'true'
List of values: `Move` ∈ {'true', 'false'}

▷ **`Scale`** (input_control) . . . . . string ⇝ *string*
Enable scaling?
Default: 'true'
List of values: `Scale` ∈ {'true', 'false'}

▷ **`KeepRatio`** (input_control) . . . . . string ⇝ *string*
Keep ratio while scaling?
Default: 'true'
List of values: `KeepRatio` ∈ {'true', 'false'}

▷ **`Degree`** (input_control) . . . . . integer ⇝ *integer*
The degree p of the NURBS curve. Reasonable values are 3 to 25.
Default: 3
Suggested values: `Degree` ∈ {2, 3, 4, 5}
Restriction: Degree >= 2

▷ **`Rows`** (output_control) . . . . . coordinates.y-array ⇝ *real*
Row coordinates of the control polygon.

▷ **`Cols`** (output_control) . . . . . coordinates.x-array ⇝ *real*
Columns coordinates of the control polygon.

▷ **`Weights`** (output_control) . . . . . real-array ⇝ *real*
Weight vector.

Result

`draw_nurbs` returns `2` (`H_MSG_TRUE`), if the window is valid. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`set_colored`, `set_line_width`, `set_draw`, `set_insert`

Alternatives

draw_xld, draw_nurbs_interp

See also

draw_nurbs_mod, draw_nurbs_interp, gen_contour_nurbs_xld

Module

Foundation

draw_nurbs_interp (: ContOut : WindowHandle, Rotate, Move, Scale, KeepRatio, Degree : ControlRows, ControlCols, Knots, Rows, Cols, Tangents)

Interactive drawing of a NURBS curve using interpolation.

This operator does not work in an HDevelop graphics window opened with dev_open_window.

draw_nurbs_interp returns the contour ContOut of a NURBS curve of degree Degree, which has been created interactively by the user in the window WindowHandle using interpolation. That means, that the user specifies a set of points and the operator computes the parameters of a NURBS curve that includes this points. By contrast, using the Operator draw_nurbs it is possible to create a NURBS curve by drawing its control polygon.

In addition to ContOut, the control information of the curve (ControlRows, ControlCols, and Knots), the interpolation points (Rows, Cols) specified by the user and the tangents at the first and the last point (Tangents) are returned. Tangents consists of four values. The first two values correspond to the y (row) and the x (column) value of the tangent at the start of the curve and the second two values to the tangent at the end of the curve, respectively.

The weight vector is not returned because it consists of equal entries. As a consequence, one can use *'auto'* as weight vector if the control information is used in a subsequent call to the operator gen_contour_nurbs_xld. For more information on NURBS see the documentation of gen_contour_nurbs_xld.

Directly after calling draw_nurbs_interp, you can add interpolation points by clicking with the left mouse button in the window at the desired positions. If enough points are specified (at least Degree $-$ 1), a NURBS curve that goes through all specified points (in the order of their generation) is computed and displayed.

When there are three points or more, the first and the last point will be marked with an additional square. By clicking on them you can close the curve or open it again. You delete the point appended last by pressing the Ctrl key.

The tangents (i.e. the first derivative of the curve) of the first and the last point are displayed as lines. They can be modified by dragging their ends using the mouse.

Existing points can be moved by dragging them with the mouse. Further new points on the curve can be inserted by a left click on the desired position on the curve.

By pressing the Shift key, you can switch into the transformation mode. In this mode you can rotate, move, and scale the curve as a whole, but only if you set the parameters Rotate, Move, and Scale, respectively, to *'true'*. Instead of the pick points and the two tangents, 3 symbols are displayed with the curve: a cross in the middle and an arrow to the right if Rotate is set to *'true'*, and a double-headed arrow to the upper right if Scale is set to *'true'*.

You can

- move the curve by clicking the left mouse button on the cross in the center and then dragging it to the new position,
- rotate it by clicking with the left mouse button on the arrow and then dragging it, till the curve has the right direction, and
- scale it by dragging the double arrow. To keep the ratio, the parameter KeepRatio has to be set to *'true'*.

By pressing the Shift key again you can switch back to the edit mode. Pressing the right mouse button terminates the procedure.

The appearance of the curve while drawing is determined by the line width, size, and color set via the operators set_color, set_colored, set_line_width, and set_line_style. The tangents and all handles are

displayed in the second color set by set_color or set_colored. Their line width is fixed to 1 and their line style is fixed to a drawn-through line.

Attention

In contrast to draw_nurbs, each point specified by the user influences the whole curve. Thus, if one point is moved, the whole curve can and will change. To minimize this effects, it is recommended to use a small degree (3-5) and to place the points such that they are approximately equally spaced. In general, uneven degrees will perform slightly better than even degrees.

Parameters

- ▷ **ContOut** (output_object) . . . . . xld_cont ⇝ *object*
  Contour of the curve.
- ▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
  Window handle.
- ▷ **Rotate** (input_control) . . . . . string ⇝ *string*
  Enable rotation?
  Default: 'true'
  List of values: Rotate ∈ {'true', 'false'}
- ▷ **Move** (input_control) . . . . . string ⇝ *string*
  Enable moving?
  Default: 'true'
  List of values: Move ∈ {'true', 'false'}
- ▷ **Scale** (input_control) . . . . . string ⇝ *string*
  Enable scaling?
  Default: 'true'
  List of values: Scale ∈ {'true', 'false'}
- ▷ **KeepRatio** (input_control) . . . . . string ⇝ *string*
  Keep ratio while scaling?
  Default: 'true'
  List of values: KeepRatio ∈ {'true', 'false'}
- ▷ **Degree** (input_control) . . . . . integer ⇝ *integer*
  The degree p of the NURBS curve. Reasonable values are 3 to 5.
  Default: 3
  Suggested values: Degree ∈ {2, 3, 4, 5}
  Restriction: Degree >= 2 && Degree <= 9
- ▷ **ControlRows** (output_control) . . . . . coordinates.y-array ⇝ *real*
  Row coordinates of the control polygon.
- ▷ **ControlCols** (output_control) . . . . . coordinates.x-array ⇝ *real*
  Column coordinates of the control polygon.
- ▷ **Knots** (output_control) . . . . . real-array ⇝ *real*
  Knot vector.
- ▷ **Rows** (output_control) . . . . . coordinates.y-array ⇝ *real*
  Row coordinates of the points specified by the user.
- ▷ **Cols** (output_control) . . . . . coordinates.x-array ⇝ *real*
  Column coordinates of the points specified by the user.
- ▷ **Tangents** (output_control) . . . . . real-array ⇝ *real*
  Tangents specified by the user.

Result

draw_nurbs_interp returns 2 (H_MSG_TRUE), if the window is valid. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

set_colored, set_line_width, set_draw, set_insert

Alternatives

draw_xld, draw_nurbs

See also

draw_nurbs_interp_mod, draw_nurbs, gen_contour_nurbs_xld

Module

Foundation

```
draw_nurbs_interp_mod ( : ContOut : WindowHandle, Rotate, Move,
    Scale, KeepRatio, Edit, Degree, RowsIn, ColsIn,
    TangentsIn : ControlRows, ControlCols, Knots, Rows, Cols,
    Tangents )
```

Interactive modification of a NURBS curve using interpolation.

This operator does not work in an HDevelop graphics window opened with `dev_open_window`.

draw_nurbs_interp_mod returns the contour ContOut of a NURBS curve of degree Degree, which has been modified interactively by the user in the window WindowHandle.

In addition to ContOut the control information of the curve (ControlRows, ControlCols, and Knots), the interpolation points (Rows, Cols) specified by the user and the tangents at the first and the last point (Tangents) are returned. Tangents consists of four values. The first two values correspond to the y (row) and the x (column) value of the tangent at the start of the curve and the second two values to the tangent at the end of the curve, respectively.

The weight vector is not returned because it consists of equal entries. As a consequence one can use *'auto'* as weight vector, if the control information is used in a subsequent call to the operator gen_contour_nurbs_xld. For more information on NURBS see the documentation of gen_contour_nurbs_xld.

The input curve is specified by the interpolation points (RowsIn, ColsIn), its degree Degree and the tangents TangentsIn, such that draw_nurbs_interp_mod can be applied to the output data of draw_nurbs_interp.

You can modify the curve in two ways: by editing the interpolation points, e.g., by inserting or moving points, or by transforming the curve as a whole, e.g., by rotating moving or scaling it. Note that you can only edit the curve if Edit is set to *'true'*. Similarly, you can only rotate, move or scale it if Rotate, Move, and Scale, respectively, are set to *'true'*.

draw_nurbs_interp_mod starts in the transformation mode. In this mode, the curve is displayed together with 3 symbols: a cross in the middle and an arrow to the right if Rotate is set to *'true'*, and a double-headed arrow to the upper right if Scale is set to *'true'*. To switch into the edit mode, press the Shift key; by pressing it again, you can switch back into the transformation mode.

Transformation Mode

In this mode a NURBS curve can be rotated, moved and scaled, as long as the respective parameter Rotate, Move and Scale are set to *'true'*. The NURBS curve is displayed along with a cross in its center, an arrow pointing to the right (if Rotate is set to *'true'*), and a double arrow (if Scale is set to *'true'*).

- To move the curve, click with the left mouse button on the cross in the center and then drag it to the new position, i.e., keep the mouse button pressed while moving the mouse.
- To rotate it, click with the left mouse button on the arrow and then drag it, till the curve has the right direction.
- Scaling is achieved by dragging the double arrow. To keep the ratio, the parameter KeepRatio has to be set to *'true'*.

Edit Mode

In this mode, the curve is displayed together with its interpolation points and the start and end tangent. Start and end point are marked by an additional square. You can perform the following modifications:

- To append new points, click with the left mouse button in the window and a new point is added at this position.

- You can delete the point appended last by pressing the Ctrl key.
- To move a point, drag it with the mouse.
- To insert a point on the curve, click on the desired position on the curve.
- To close respectively open the curve, click on the first or on the last point.

Pressing the right mouse button terminates the procedure.

The appearance of the curve while drawing is determined by the line width, size and color set via the operators `set_color`, `set_colored`, `set_line_width`, and `set_line_style`. The tangents and all handles are displayed in the second color set by `set_color` or `set_colored`. Their line width is fixed to 1 and their line style is fixed to a drawn-through line.

Attention

In contrast to `draw_nurbs`, each point specified by the user influences the whole curve. Thus, if one point is moved, the whole curve can and will change. To minimize this effects, it is recommended to use a small degree (3-5) and to place the points such that they are approximately equally spaced. In general, uneven degrees will perform slightly better than even degrees.

Parameters

▷ **ContOut** (output_object) ..... xld_cont ⇝ *object*
Contour of the modified curve.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Rotate** (input_control) ..... string ⇝ *string*
Enable rotation?
Default: 'true'
List of values: `Rotate` ∈ {'true', 'false'}

▷ **Move** (input_control) ..... string ⇝ *string*
Enable moving?
Default: 'true'
List of values: `Move` ∈ {'true', 'false'}

▷ **Scale** (input_control) ..... string ⇝ *string*
Enable scaling?
Default: 'true'
List of values: `Scale` ∈ {'true', 'false'}

▷ **KeepRatio** (input_control) ..... string ⇝ *string*
Keep ratio while scaling?
Default: 'true'
List of values: `KeepRatio` ∈ {'true', 'false'}

▷ **Edit** (input_control) ..... string ⇝ *string*
Enable editing?
Default: 'true'
List of values: `Edit` ∈ {'true', 'false'}

▷ **Degree** (input_control) ..... integer ⇝ *integer*
The degree p of the NURBS curve. Reasonable values are 3 to 5.
Default: 3
Suggested values: `Degree` ∈ {2, 3, 4, 5}
Restriction: Degree >= 2 && Degree <= 9

▷ **RowsIn** (input_control) ..... coordinates.y-array ⇝ *real*
Row coordinates of the input interpolation points.

▷ **ColsIn** (input_control) ..... coordinates.x-array ⇝ *real*
Column coordinates of the input interpolation points.

▷ **TangentsIn** (input_control) ..... real-array ⇝ *real*
Input tangents.

▷ **ControlRows** (output_control) ..... coordinates.y-array ⇝ *real*
Row coordinates of the control polygon.

▷ **ControlCols** (output_control) ..... coordinates.x-array ⇝ *real*
Column coordinates of the control polygon.

▷ **Knots** (output_control) . . . . . real-array ⇝ *real*
Knot vector.
▷ **Rows** (output_control) . . . . . coordinates.y-array ⇝ *real*
Row coordinates of the points specified by the user.
▷ **Cols** (output_control) . . . . . coordinates.x-array ⇝ *real*
Column coordinates of the points specified by the user.
▷ **Tangents** (output_control) . . . . . real-array ⇝ *real*
Tangents specified by the user.

Result

`draw_nurbs_interp_mod` returns `2` (`H_MSG_TRUE`), if the window is valid. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`set_colored`, `set_line_width`, `set_draw`, `set_insert`

Alternatives

`draw_xld_mod`, `draw_nurbs_mod`

See also

`draw_nurbs_interp`, `gen_contour_nurbs_xld`

Module

Foundation

draw_nurbs_mod (: ContOut : WindowHandle, Rotate, Move, Scale, KeepRatio, Edit, Degree, RowsIn, ColsIn, WeightsIn : Rows, Cols, Weights)

Interactive modification of a NURBS curve.

This operator does not work in an HDevelop graphics window opened with `dev_open_window`.

`draw_nurbs_mod` returns the contour `ContOut` and control information (`Rows`, `Cols`, and `Weights`) of a NURBS curve of degree `Degree`, which has been interactively modified by the user in the window `WindowHandle`. For additional information concerning NURBS curves, see the documentation of `gen_contour_nurbs_xld`. To use the control information `Rows`, `Cols`, and `Weights` in a subsequent call to the operator `gen_contour_nurbs_xld`, the knot vector `Knots` should be set to *'auto'*.

The input NURBS curve is specified by its control polygon (`RowsIn`, `ColsIn`), its weight vector `WeightsIn` and its degree `Degree`. The knot vector is assumed to be uniform (i.e. *'auto'* in `gen_contour_nurbs_xld`).

You can modify the curve in two ways: by editing the control polygon, e.g., by inserting or moving control points, or by transforming the contour as a whole, e.g., by rotating moving or scaling it. Note that you can only edit the control polygon if `Edit` is set to *'true'*. Similarly, you can only rotate, move or scale it if `Rotate`, `Move`, and `Scale`, respectively, are set to *'true'*.

`draw_nurbs_mod` starts in the transformation mode. In this mode, the curve is displayed together with 3 symbols: a cross in the middle and an arrow to the right if `Rotate` is set to *'true'*, and a double-headed arrow to the upper right if `Scale` is set to *'true'*. To switch into the edit mode, press the Shift key; by pressing it again, you can switch back into the transformation mode.

Transformation Mode

In this mode a curve can be rotated, moved and scaled, as long as the respective parameter `Rotate`, `Move` and `Scale` are set to *'true'*. The curve is displayed along with a cross in its center, an arrow pointing to the right (if `Rotate` is set to *'true'*), and a double arrow (if `Scale` is set to *'true'*).

- To move the curve, click with the left mouse button on the cross in the center and then drag it to the new position, i.e., keep the mouse button pressed while moving the mouse.
- To rotate it, click with the left mouse button on the arrow and then drag it, till the curve has the right direction.
- Scaling is achieved by dragging the double arrow. To keep the ratio, the parameter `KeepRatio` has to be set to *'true'*.

Edit Mode

In this mode, the curve is displayed together with its control polygon. Start and end point are marked by an additional square and the point which was handled last is surrounded by a circle representing its weight. You can perform the following modifications:

- To append control points, click with the left mouse button in the window and a new point is added at this position.
- You can delete the point appended last by pressing the Ctrl key.
- To move a point, drag it with the mouse.
- To insert a point on the control polygon, click on the desired position on the polygon.
- To close respectively open the curve, click on the first or on the last control point.
- You can modify the weight of a control point by first clicking on the point itself (if it is not already the point which was modified or created last) and then dragging the circle around the point.

Pressing the right mouse button terminates the procedure.

The appearance of the curve while drawing is determined by the line width, size and color set via the operators `set_color`, `set_colored`, `set_line_width`, and `set_line_style`. The control polygon and all handles are displayed in the second color set by `set_color` or `set_colored`. Their line width is fixed to 1 and their line style is fixed to a drawn-through line.

Parameters

▷ **`ContOut`** (output_object) ..... xld_cont ⇝ *object*
Contour of the modified curve.

▷ **`WindowHandle`** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **`Rotate`** (input_control) ..... string ⇝ *string*
Enable rotation?
Default: 'true'
List of values: `Rotate` ∈ {'true', 'false'}

▷ **`Move`** (input_control) ..... string ⇝ *string*
Enable moving?
Default: 'true'
List of values: `Move` ∈ {'true', 'false'}

▷ **`Scale`** (input_control) ..... string ⇝ *string*
Enable scaling?
Default: 'true'
List of values: `Scale` ∈ {'true', 'false'}

▷ **`KeepRatio`** (input_control) ..... string ⇝ *string*
Keep ratio while scaling?
Default: 'true'
List of values: `KeepRatio` ∈ {'true', 'false'}

▷ **`Edit`** (input_control) ..... string ⇝ *string*
Enable editing?
Default: 'true'
List of values: `Edit` ∈ {'true', 'false'}

▷ **`Degree`** (input_control) ..... integer ⇝ *integer*
The degree p of the NURBS curve. Reasonable values are 3 to 25.
Default: 3
Suggested values: `Degree` ∈ {2, 3, 4, 5}
Restriction: Degree >= 2

▷ **RowsIn** (input_control) ..... coordinates.y-array ⇝ *real*
Row coordinates of the input control polygon.

▷ **ColsIn** (input_control) ..... coordinates.x-array ⇝ *real*
Column coordinates of the input control polygon.

▷ **WeightsIn** (input_control) ..... real-array ⇝ *real*
Input weight vector.

▷ **Rows** (output_control) ..... coordinates.y-array ⇝ *real*
Row coordinates of the control polygon.

▷ **Cols** (output_control) ..... coordinates.x-array ⇝ *real*
Columns coordinates of the control polygon.

▷ **Weights** (output_control) ..... real-array ⇝ *real*
Weight vector.

Result

`draw_nurbs_mod` returns `2` (`H_MSG_TRUE`), if the window is valid. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`set_colored`, `set_line_width`, `set_draw`, `set_insert`

Alternatives

`draw_nurbs_interp_mod`, `draw_xld_mod`

See also

`draw_nurbs`, `draw_nurbs_interp`, `gen_contour_nurbs_xld`

Module

Foundation

draw_point (: : WindowHandle : Row, Column)

Draw a point.

`draw_point` returns the parameter for a point, which has been created interactively by the user in the window.

To create a point you have to press the left mouse button. While keeping the button pressed you may “drag” the point in any direction. Pressing the right mouse button terminates the procedure.

After terminating the procedure the point is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row** (output_control) ..... point.y ⇝ *real*
Row index of the point.

▷ **Column** (output_control) ..... point.x ⇝ *real*
Column index of the point.

Example

```
draw_point(WindowHandle,Row,Column)
gen_cross_contour_xld (Cross, Row, Column, 6, 0)
```

Result

`draw_point` returns `2` (`H_MSG_TRUE`), if the window is valid and the needed drawing mode is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`reduce_domain`, `disp_line`, `set_colored`, `set_line_width`, `set_draw`, `set_insert`

See also

`draw_point_mod`, `draw_circle`, `draw_ellipse`, `set_insert`

Module

Foundation

draw_point_mod (: : WindowHandle, RowIn, ColumnIn : Row, Column)

Draw a point.

`draw_point_mod` returns the parameter for a point, which has been created interactively by the user in the window.

To create a point are expected the coordinates `RowIn` and `ColumnIn`. While keeping the button pressed you may "drag" the point in any direction. Pressing the right mouse button terminates the procedure.

After terminating the procedure the point is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **RowIn** (input_control) ..... point.y ⇝ *real*
  Row index of the point.
- ▷ **ColumnIn** (input_control) ..... point.x ⇝ *real*
  Column index of the point.
- ▷ **Row** (output_control) ..... point.y ⇝ *real*
  Row index of the point.
- ▷ **Column** (output_control) ..... point.x ⇝ *real*
  Column index of the point.

Example

```
draw_point_mod (WindowHandle, 100, 100, Row,Column)
gen_cross_contour_xld (Cross, Row, Column, 6, 0)
```

Result

draw_point_mod returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_line, set_colored, set_line_width, set_draw, set_insert

See also

draw_point, draw_circle, draw_ellipse, set_insert

Module

Foundation

draw_polygon (: PolygonRegion : WindowHandle :)

Interactive drawing of a polygon row.

draw_polygon produces an image. The region of that image spans exactly the image points entered interactively by mouse clicks (gray values remain undefined).

Painting in the output window happens while pressing the left mouse button. Releasing the left mouse button and repressing it at another position effects drawing a line between these two points. Pressing the right mouse button terminates the input. Painting uses that color which has been set by set_color, set_rgb, etc. .

To put gray values on the created PolygonRegion for further processing, you may use the operator reduce_domain.

Attention

The painted contour is not closed automatically, in particular it is not "filled up" either.

Output object's gray values are not defined.

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **PolygonRegion** (output_object) ..... region ⇝ *object*
Region, which encompasses all painted points.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

Example

```
draw_polygon(Polygon,WindowHandle)
shape_trans(Polygon,Filled,'convex')
dev_display (Filled)
```

Result

If the window is valid, draw_polygon returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_region, set_colored, set_line_width, set_draw

Alternatives

draw_region, draw_circle, draw_rectangle1, draw_rectangle2, boundary

See also

reduce_domain, fill_up, set_color

Module

Foundation

draw_rectangle1 (: : WindowHandle : Row1, Column1, Row2, Column2)

Draw a rectangle parallel to the coordinate axis.

draw_rectangle1 returns the parameter for a rectangle parallel to the coordinate axes, which has been created interactively by the user in the window.

To create a rectangle you have to press the left mouse button determining a corner of the rectangle. While keeping the button pressed you may "drag" the rectangle in any direction. After another mouse click in the middle of the created rectangle you can move it. A click close to one side "grips" it to modify the rectangle's dimension in perpendicular direction to this side. If you click on one corner of the created rectangle, you may move this corner. Pressing the right mouse button terminates the procedure.

After terminating the procedure the rectangle is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Row1** (output_control) ..... rectangle.origin.y ⇝ *real*
  Row index of the left upper corner.
- ▷ **Column1** (output_control) ..... rectangle.origin.x ⇝ *real*
  Column index of the left upper corner.
- ▷ **Row2** (output_control) ..... rectangle.corner.y ⇝ *real*
  Row index of the right lower corner.
- ▷ **Column2** (output_control) ..... rectangle.corner.x ⇝ *real*
  Column index of the right lower corner.

Example

```
read_image(Image,'monkey')
get_image_size (Image, Width, Height)
dev_display (Image)
draw_rectangle1(WindowHandle,Row1,Column1,Row2,Column2)
dev_set_part (Row1, Column1, Row2, Column2)
dev_display (Image)
```

Result

draw_rectangle1 returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_region, set_colored, set_line_width, set_draw, set_insert

Alternatives

draw_rectangle1_mod, draw_rectangle2, draw_region

See also

gen_rectangle1, draw_circle, draw_ellipse, set_insert

Module

Foundation

draw_rectangle1_mod (: : WindowHandle, Row1In, Column1In, Row2In, Column2In : Row1, Column1, Row2, Column2)

Draw a rectangle parallel to the coordinate axis.

draw_rectangle1_mod returns the parameter for a rectangle parallel to the coordinate axes, which has been created interactively by the user in the window.

To create a rectangle are expected the parameters Row1In, Column1In,Row2In and Column2In. After a mouse click in the middle of the created rectangle you can move it. A click close to one side "grips" it to modify the rectangle's dimension in perpendicular direction to this side. If you click on one corner of the created rectangle, you may move this corner. Pressing the right mouse button terminates the procedure.

After terminating the procedure the rectangle is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row1In** (input_control) ..... rectangle.origin.y ⇝ *real*
Row index of the left upper corner.

▷ **Column1In** (input_control) ..... rectangle.origin.x ⇝ *real*
Column index of the left upper corner.

▷ **Row2In** (input_control) ..... rectangle.corner.y ⇝ *real*
Row index of the right lower corner.

▷ **Column2In** (input_control) ..... rectangle.corner.x ⇝ *real*
Column index of the right lower corner.

▷ **Row1** (output_control) ..... rectangle.origin.y ⇝ *real*
Row index of the left upper corner.

▷ **Column1** (output_control) ..... rectangle.origin.x ⇝ *real*
Column index of the left upper corner.

▷ **Row2** (output_control) ..... rectangle.corner.y ⇝ *real*
Row index of the right lower corner.

▷ **Column2** (output_control) ..... rectangle.corner.x ⇝ *real*
Column index of the right lower corner.

Example

```
read_image(Image,'monkey')
get_image_size (Image, Width, Height)
```

```
dev_display (Image)
draw_rectangle1_mod(WindowHandle,100,100,200,300,\
                    Row1,Column1,Row2,Column2)
dev_set_part (Row1, Column1, Row2, Column2)
dev_display (Image)
```

Result

`draw_rectangle1_mod` returns `2` (`H_MSG_TRUE`), if the window is valid and the needed drawing mode (see `set_insert`) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`reduce_domain`, `disp_region`, `set_colored`, `set_line_width`, `set_draw`, `set_insert`

Alternatives

`draw_rectangle1`, `draw_rectangle2`, `draw_region`

See also

`gen_rectangle1`, `draw_circle`, `draw_ellipse`, `set_insert`

Module

Foundation

draw_rectangle2 (: : WindowHandle : Row, Column, Phi, Length1, Length2)

Interactive drawing of any orientated rectangle.

`draw_rectangle2` returns the parameter for any orientated rectangle, which has been created interactively by the user in the window.

The created rectangle is described by its center, its two half axes and the angle between the first half axis and the horizontal coordinate axis.

To create a rectangle you have to press the left mouse button for the center of the rectangle. While keeping the button pressed you may dimension the length (`Length1`) and the orientation (`Phi`) of the first half axis. In doing so a temporary default length for the second half axis is assumed, which may be modified afterwards on demand. After another mouse click in the middle of the created rectangle, you can move it. A click close to one side "grips" it to modify the rectangle's dimension in perpendicular direction to this side. You only can modify the orientation, if you grip a side perpendicular to the first half axis. Pressing the right mouse button terminates the procedure.

After terminating the procedure the rectangle is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row** (output_control) ..... rectangle2.center.y ⇝ *real*
Row index of the center.

▷ **Column** (output_control) ..... rectangle2.center.x ⇝ *real*
Column index of the center.

▷ **Phi** (output_control) . . . . . rectangle2.angle.rad ⇝ *real*
Orientation of the bigger half axis in radians.
▷ **Length1** (output_control) . . . . . rectangle2.hwidth ⇝ *real*
Bigger half axis.
▷ **Length2** (output_control) . . . . . rectangle2.hheight ⇝ *real*
Smaller half axis.

Result

`draw_rectangle2` returns `2` (`H_MSG_TRUE`), if the window is valid and the needed drawing mode (see `set_insert`) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`reduce_domain`, `disp_region`, `set_colored`, `set_line_width`, `set_draw`, `set_insert`

Alternatives

`draw_rectangle2_mod`, `draw_rectangle1`, `draw_region`

See also

`gen_rectangle2`, `draw_circle`, `draw_ellipse`, `set_insert`

Module

Foundation

draw_rectangle2_mod (: : WindowHandle, RowIn, ColumnIn, PhiIn, Length1In, Length2In : Row, Column, Phi, Length1, Length2)

Interactive drawing of any orientated rectangle.

`draw_rectangle2_mod` returns the parameter for any orientated rectangle, which has been created interactively by the user in the window.

The created rectangle is described by its center, its two half axes and the angle between the first half axis and the horizontal coordinate axis.

To create a rectangle are expected the parameters `RowIn`, `ColumnIn`,`PhiIn`,`Length1In`,`Length2In`. A click close to one side "grips" it to modify the rectangle's dimension in perpendicular direction (`Length2`) to this side. You only can modify the orientation (`Phi`), if you grip a side perpendicular to the first half axis. After another mouse click in the middle of the created rectangle, you can move it. Pressing the right mouse button terminates the procedure.

After terminating the procedure the rectangle is not visible in the window any longer.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.
▷ **RowIn** (input_control) . . . . . rectangle2.center.y ⇝ *real*
Row index of the center.
▷ **ColumnIn** (input_control) . . . . . rectangle2.center.x ⇝ *real*
Column index of the center.
▷ **PhiIn** (input_control) . . . . . rectangle2.angle.rad ⇝ *real*
Orientation of the bigger half axis in radians.

▷ **Length1In** (input_control) . . . . . rectangle2.hwidth ⇝ *real*
Bigger half axis.

▷ **Length2In** (input_control) . . . . . rectangle2.hheight ⇝ *real*
Smaller half axis.

▷ **Row** (output_control) . . . . . rectangle2.center.y ⇝ *real*
Row index of the center.

▷ **Column** (output_control) . . . . . rectangle2.center.x ⇝ *real*
Column index of the center.

▷ **Phi** (output_control) . . . . . rectangle2.angle.rad ⇝ *real*
Orientation of the bigger half axis in radians.

▷ **Length1** (output_control) . . . . . rectangle2.hwidth ⇝ *real*
Bigger half axis.

▷ **Length2** (output_control) . . . . . rectangle2.hheight ⇝ *real*
Smaller half axis.

Result

`draw_rectangle2_mod` returns `2` (`H_MSG_TRUE`), if the window is valid and the needed drawing mode (see `set_insert`) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`reduce_domain`, `disp_region`, `set_colored`, `set_line_width`, `set_draw`, `set_insert`

Alternatives

`draw_rectangle2`, `draw_rectangle1`, `draw_rectangle2`, `draw_region`

See also

`gen_rectangle2`, `draw_circle`, `draw_ellipse`, `set_insert`

Module

Foundation

draw_region (: Region : WindowHandle :)

Interactive drawing of a closed region.

`draw_region` produces an image. The region of that image spans exactly the image region entered interactively by mouse clicks (gray values remain undefined). Painting happens in the output window while keeping the left mouse button pressed. The left mouse button even operates by clicking in the output window; through this a line between the previous clicked points is drawn. Clicking the right mouse button terminates input and closes the outline. Subsequently the image is “filled up”. Also it contains the whole image area enclosed by the mouse.

Painting uses that color which has been set by `set_color`, `set_rgb`, etc. .

Pressing the right mouse button terminates the procedure.

Attention

The output object’s gray values are not defined.

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **Region** (output_object) ..... region ⇝ *object*
Interactive created region.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

Example

```
read_image(Image,'fabrik')
dev_display (Image)
draw_region(Region,WindowHandle)
reduce_domain(Image,Region,ImageReduced)
invert_image (ImageReduced, ImageInvert)
dev_display (ImageInvert)
```

Result

If the window is valid, `draw_region` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`reduce_domain`, `disp_region`, `set_colored`, `set_line_width`, `set_draw`

Alternatives

`draw_circle`, `draw_ellipse`, `draw_rectangle1`, `draw_rectangle2`

See also

`draw_polygon`, `reduce_domain`, `fill_up`, `set_color`

Module

Foundation

draw_xld (: ContOut : WindowHandle, Rotate, Move, Scale, KeepRatio :)

Interactive drawing of a contour.

`draw_xld` returns a contour, which has been created interactively by the user in the window.

Directly after calling `draw_xld` you can add contour points by clicking with the left mouse button in the window at the desired positions. You delete the point appended last by pressing the Ctrl key. As soon as you add three contour points, 5 so-called pick points are displayed, one in the middle and 4 at the corners of the surrounding rectangle. By clicking on a pick point you can close the contour or open it again. If the contour is closed, the pick points are displayed in form of squares, otherwise they are shaped like a 'u'.

If the contour is closed you can

- move contour points by clicking with the left mouse button on a point marked by a rectangle and keep the mouse button pressed while moving the mouse,
- insert contour points by clicking with the left mouse button in the vicinity of a line and then move the mouse to the position where you want the new point to be placed, and
- delete contour points by selecting the point which should be deleted with the left mouse button and then press the Ctrl key.

By pressing the Shift key, you can switch into the transformation mode. In this mode you can rotate, move, and scale the contour as a whole, but only if you set the parameters `Rotate`, `Move`, and `Scale`, respectively, to *'true'*. Instead of the pick points, 3 symbols are displayed with the contour: a cross in the middle and an arrow to the right if `Rotate` is set to *'true'*, and a double-headed arrow to the upper right if `Scale` is set to *'true'*.

You can

- move the contour by clicking the left mouse button on the cross in the center and then dragging it to the new position,
- rotate it by clicking with the left mouse button on the arrow and then dragging it, till the contour has the right direction, and
- scale it by dragging the double arrow. To keep the ratio the parameter `KeepRatio` has to be set to *'true'*.

Pressing the right mouse button terminates the procedure.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **`ContOut`** (output_object) ..... xld_cont ⇝ *object*
Modified contour.

▷ **`WindowHandle`** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **`Rotate`** (input_control) ..... string ⇝ *string*
Enable rotation?
Default: 'true'
List of values: `Rotate` ∈ {'true', 'false'}

▷ **`Move`** (input_control) ..... string ⇝ *string*
Enable moving?
Default: 'true'
List of values: `Move` ∈ {'true', 'false'}

▷ **`Scale`** (input_control) ..... string ⇝ *string*
Enable scaling?
Default: 'true'
List of values: `Scale` ∈ {'true', 'false'}

▷ **`KeepRatio`** (input_control) ..... string ⇝ *string*
Keep ratio while scaling?
Default: 'true'
List of values: `KeepRatio` ∈ {'true', 'false'}

Result

`draw_xld` returns `2` (`H_MSG_TRUE`), if the window is valid and the needed drawing mode (see `set_insert`) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`reduce_domain`, `disp_region`, `set_colored`, `set_line_width`, `set_draw`, `set_insert`

Alternatives

`draw_rectangle2`, `draw_rectangle1`, `draw_rectangle2`, `draw_region`

See also

`gen_rectangle2`, `draw_circle`, `draw_ellipse`, `set_insert`

Module

Foundation

draw_xld_mod (ContIn : ContOut : WindowHandle, Rotate, Move, Scale, KeepRatio, Edit :)

Interactive modification of a contour.

draw_xld_mod returns a contour, which has been interactively modified by the user in the window.

You can modify the contour in two ways: by editing the contour itself, e.g., by inserting or moving contour points, or by transforming the contour as a whole, e.g., by rotating moving or scaling it. Note that you can only edit a contour if Edit is set to *'true'*. Similarly, you can only rotate, move or scale it if Rotate, Move, and Scale, respectively, are set to *'true'*.

draw_xld_mod starts in the transformation mode. In this mode, the contour is displayed together with 3 symbols: a cross in the middle and an arrow to the right if Rotate is set to *'true'*, and a double-headed arrow to the upper right if Scale is set to *'true'*. To switch into the edit mode, press the Shift key; by pressing it again, you can switch back into the transformation mode.

Transformation Mode

In this mode a contour can be rotated, moved and scaled, as long as the respective parameter Rotate, Move and Scale are set to *'true'*. The contour is displayed along with a cross in its center, an arrow pointing to the right (if Rotate is set to *'true'*), and a double arrow (if Scale is set to *'true'*).

- To move the contour, click with the left mouse button on the cross in the center and then drag it to the new position, i.e., keep the mouse button pressed while moving the mouse.
- To rotate it, click with the left mouse button on the arrow and then drag it, till the contour has the right direction.
- Scaling is achieved by dragging the double arrow. To keep the ratio the parameter KeepRatio has to be set to *'true'*.

Edit Mode

In this mode, the contour is display together with 5 pick points, which are located in the middle and at the corners of the surrounding rectangle. If the contour is closed, the pick points are displayed as squares, otherwise shaped like a 'u'. By clicking on a pick point, you can close an open contour and vice versa. Depending on the state of the contour, you can perform different modifications. Open contours (pick points shaped like a 'u')

- To append points, click with the left mouse button in the window and a new point is added at this position.
- You can delete the point appended last by pressing the Ctrl key.
- To move or insert points, you must first close the contour by clicking on one of the pick points.

Closed contours (square pick points)

- To move a point, click with the left mouse button on a point marked by a rectangle and then drag it to the new position.
- To insert a point, click with the left mouse button in the vicinity of a line and then move the mouse to the position where you want the new point to be placed.
- To delete a point, select the point which should be deleted with the left mouse button and then press the Ctrl key.

Pressing the right mouse button terminates the procedure.

Attention

If used in a buffer window, mouse events have to be supplied by the application, while the draw operator must be run in another thread.

Parameters

▷ **ContIn** (input_object) ..... xld_cont ⇝ *object*
Input contour.

▷ **ContOut** (output_object) ..... xld_cont ⇝ *object*
Modified contour.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Rotate** (input_control) . . . . . string ⇝ *string*
Enable rotation?
Default: 'true'
List of values: Rotate ∈ {'true', 'false'}

▷ **Move** (input_control) . . . . . string ⇝ *string*
Enable moving?
Default: 'true'
List of values: Move ∈ {'true', 'false'}

▷ **Scale** (input_control) . . . . . string ⇝ *string*
Enable scaling?
Default: 'true'
List of values: Scale ∈ {'true', 'false'}

▷ **KeepRatio** (input_control) . . . . . string ⇝ *string*
Keep ratio while scaling?
Default: 'true'
List of values: KeepRatio ∈ {'true', 'false'}

▷ **Edit** (input_control) . . . . . string ⇝ *string*
Enable editing?
Default: 'true'
List of values: Edit ∈ {'true', 'false'}

Result

draw_xld_mod returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, disp_region, set_colored, set_line_width, set_draw, set_insert

Alternatives

draw_rectangle2, draw_rectangle1, draw_rectangle2, draw_region

See also

gen_rectangle2, draw_circle, draw_ellipse, set_insert

Module

Foundation

13.3 LUT

get_lut (: : WindowHandle : LookUpTable)

Get current look-up-table (lut).

get_lut returns the name or the values of the look-up-table (lut) of the window, currently used by disp_image (or indirectly by disp_region, etc.) for output. To set a look-up-table use set_lut. If the current table is a system table without any modification (by set_fix), the name of the table is returned. If it is a modified table, a table read from a file or a table for output with pseudo real colors, the RGB-values of the table are returned.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **LookUpTable** (output_control) . . . . . string-array ⇝ *string* / integer
Name of look-up-table or tuple of RGB-values.

Result

`get_lut` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`set_lut`

See also

`set_lut`

Module

Foundation

query_lut (: : WindowHandle : LookUpTable)

Query all available look-up-tables (lut).

`query_lut` returns the names of all look-up-tables available on the current used device. These tables can be set with `set_lut`. An table named 'default' is always available.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **LookUpTable** (output_control) . . . . . string-array ⇝ *string*
Names of look-up-tables.

Result

`query_lut` returns `2` (`H_MSG_TRUE`) if a window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`set_lut`, `disp_lut`

See also

`set_lut`

Module

Foundation

set_lut (: : WindowHandle, LookUpTable :)

Set "look-up-table" (lut).

set_lut sets look-up-table of the device (monitor) displaying the output window. A look-up-table defines the transformation of a “gray value” within an image into a gray value or color on the screen. It describes the screen gray value/color as a combination of red, green and blue for any image gray value (0..255) (so it is a ’table’ to ’look up’ the screen gray value/color for each image gray value: look-up-table). Transformation into screen-colors is performed in real-time at every time the screen is displayed new (typically this happens about 60 - 70 times per second). So it is possible to change the look-up-table to get a new look of images or regions. Please remind that not all machines support changing the look-up-table (e.g., monochrome resp. real color).

Look-up-tables within HALCON (and on a machine that supports 256 colors) are divided into three areas:

S: system area resp. user area,

G: graphic colors,

B: image data.

Colors in S descend from applications that were active before starting HALCON and should not get lost. Graphic colors in G are used for operators such as disp_region, disp_circle etc. and are set unique within all look-up-tables. An output in a graphic color has always got the same (color-)look, even if different look-up-tables are used. set_color and set_rgb set graphic colors. Gray values resp. colors in B are used by disp_image to display an image. They can change according to the current look-up-table. There exist two exceptions to this concept:

- set_gray allows setting of colors of the area B for operators such as disp_region,

For common monitors only one look-up-table can be loaded per screen. Whereas set_lut can be activated separately for each window. There is the following solution for this problem: It will always be activated the look-up-table that is assigned to the “active window” (a window is set into the state “active” by the window manager).

look-up-table can also be used with true-color displays. In this case the look-up-table will be simulated in software. This means, that the look-up-table will be used each time an image is displayed.

query_lut lists the names of all look-up-tables. They differ from each other in the area used for gray values. Within this area the following behavior is defined:

Gray value tables (1-7 image levels)

Value	Description
’default’	Only the two basic colors (generally black and white) are used.

Color tables (real color, static gray value steps)

Value	Description
’default’	Table proposed by the hardware.

Gray value tables (256 colors)

Value	Description
’default’	As *’linear’*.
’linear’	Linear increasing of gray values from 0 (black) to 255 (white).
’inverse’	Inverse function of *’linear’*.
’sqr’	Gray values increase according to square function.
’inv_sqr’	Inverse function of *’sqr’*.
’cube’	Gray values increase according to cubic function.
’inv_cube’	Inverse function of *’cube’*.
’sqrt’	Gray values increase according to square-root function.
’inv_sqrt’	Inverse Function of *’sqrt’*.
’cubic_root’	Gray values increase according to cubic-root function.
’inv_cubic_root’	Inverse Function of *’cubic_root’*.

Color tables (256 colors)

Value	Description
'color1'	Linear transition from red via green to blue.
'color2'	Smooth transition from yellow via red, blue to green.
'color3'	Smooth transition from yellow via red, blue, green, red to blue.
'color4'	Smooth transition from yellow via red to blue.
'three'	Displaying the three colors red, green and blue.
'six'	Displaying the six basic colors yellow, red, magenta, blue, cyan and green.
'twelve'	Displaying 12 colors.
'twenty_four'	Displaying 24 colors.
'rainbow'	Displaying the spectral colors from red via green to blue.
'temperature'	Temperature table from black via red, yellow to white.
'change1'	Color changes after every pixel within the table alternating the six basic colors.
'change2'	Fivefold color changes from green via red to blue.
'change3'	Threefold color changes from green via red to blue.
'jet'	Smooth transition from blue via green, yellow to red.
'inverse_jet'	Smooth transition from red via yellow, green to blue.
'batlow'	Perceptually uniform and color-vision-deficiency friendly.
'inverse_batlow'	Perceptually uniform and color-vision-deficiency friendly.

The look-up-table `LookUpTable` to be used in `WindowHandle` can be set in the following ways:

Name of the look-up-table Select the look-up-table by setting its name (see `query_lut`).

File name of look-up-table Read a look-up-table from a file. Every line of such a file must contain three numbers in the range of 0 to 255, with the first number describing the amount of red, the second the amount of green and the third the amount of blue of the represented display color. The number of lines can vary. The first line contains information for the first gray value and the last line for the last value. If there are less lines than gray values, the available information values are distributed over the whole interval. If there are more lines than gray values, a number of (uniformly distributed) lines is ignored. The file-name must conform to "`LookUpTable.lut`". Within the parameter the name is specified without file extension. HALCON will search for the file in the current directory and after that in a specified directory (see *'lut_dir'* in `set_system`).

Tuple Set RGB values directly as a tuple. The number of parameter values must conform to the number of pixels currently used within the look-up-table.

Any of the above combined with custom value range Set a custom value range to be mapped by the look-up-table. By default, HALCON uses the minimum and maximum gray values g_min and g_max of an image (except `byte` images) and maps these values to 0 and 255 when applying the look-up-table. It is possible to override this behavior by passing a tuple to `set_lut` in the form *[LUT, g_min, g_max]* (where *'LUT'* is any of the options above). This feature can be particularly useful when dealing with outliers in a `real` image.
The default behavior is reset by using one of the options above without minimum and maximum gray values.

Attention

`set_lut` can only be used with monitors supporting 256 gray levels/colors.

Parameters

▷ **`WindowHandle`** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **`LookUpTable`** (input_control) . . . . . filename.read(-array) ⇝ *string* / integer / real
Name of look-up-table, values of look-up-table (RGB) or file name.
Default: 'default'
Suggested values: `LookUpTable` ∈ {'default', 'linear', 'inverse', 'sqr', 'inv_sqr', 'cube', 'inv_cube', 'sqrt', 'inv_sqrt', 'cubic_root', 'inv_cubic_root', 'color1', 'color2', 'color3', 'color4', 'three', 'six', 'twelve',

'twenty_four', 'rainbow', 'temperature', 'cyclic_gray', 'cyclic_temperature', 'hsi', 'change1', 'change2', 'change3', 'jet', 'inverse_jet', 'batlow', 'inverse_batlow'}

File extension: .lut

Example

```
read_image(Image,'monkey')
query_lut(WindowHandle,LUTs)
for i := 0 to |LUTs|-1 by 1
  set_lut(WindowHandle,LUTs[i])
  fwrite_string(FileHandle, ['current table ',LUTs[i]])
  fnew_line(FileHandle)
  get_mbutton(WindowHandle,_,_,_)
endfor
```

Result

set_lut returns 2 (H_MSG_TRUE) if the hardware supports a look-up-table and the parameter is correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_lut, get_lut

See also

get_lut, query_lut, set_color, set_rgb, set_hsi, write_lut

Module

Foundation

13.4 Mouse

get_mbutton (: : WindowHandle : Row, Column, Button)

Wait until a mouse button is pressed.

get_mbutton returns the pixel accurate image coordinates of the mouse pointer in the output window and the mouse button pressed. In addition the state of the modifier keys is returned. The following values are assigned to the individual buttons and keys:

1: Left button,

2: Middle button,

4: Right button,

8: Shift key,

16: Ctrl key,

32: Alt key.

The sum of the values for all pressed keys added to the value of the pressed mouse button is returned in Button.

The operator waits until a mouse button is pressed in the output window. For graphics windows the coordinates Row and Column are expressed with consideration of the current image part (see set_part).

If subpixel accurate image coordinates are required, you can use the operator get_mbutton_sub_pix.

Attention

get_mbutton only returns if a mouse button is pressed in the window.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row** (output_control) . . . . . point.y ⇝ *integer*
Row coordinate of the mouse cursor in the image coordinate system.

▷ **Column** (output_control) . . . . . point.x ⇝ *integer*
Column coordinate of the mouse cursor in the image coordinate system.

▷ **Button** (output_control) . . . . . integer ⇝ *integer*
Mouse button(s) pressed.

Result

`get_mbutton` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Alternatives

`get_mposition`, `get_mbutton_sub_pix`, `get_mposition_sub_pix`

See also

`open_window`

Module

Foundation

get_mbutton_sub_pix (: : WindowHandle : Row, Column, Button)

Wait until a mouse button is pressed and get the subpixel mouse position.

`get_mbutton_sub_pix` returns the subpixel accurate image coordinates of the mouse pointer in the output window and the mouse button pressed. In addition the state of the modifier keys is returned. The following values are assigned to the individual buttons and keys:

1: Left button,

2: Middle button,

4: Right button,

8: Shift key,

16: Ctrl key,

32: Alt key.

The sum of the values for all pressed keys added to the value of the pressed mouse button is returned in `Button`.

The operator waits until a button is pressed in the output window. For graphics windows the coordinates `Row` and `Column` are expressed with consideration of the current image part (see `set_part`).

If only pixel accurate image coordinates are required, you can use the operator `get_mbutton`.

Attention

`get_mbutton_sub_pix` only returns if a mouse button is pressed in the window.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row** (output_control) . . . . . point.y ⇝ *real*
Row coordinate of the mouse cursor in the image coordinate system.

▷ **Column** (output_control) . . . . . point.x ⇝ *real*
Column coordinate of the mouse cursor in the image coordinate system.

▷ **Button** (output_control) . . . . . integer ⇝ *integer*
Mouse button(s) pressed.

Result

`get_mbutton_sub_pix` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Alternatives

`get_mbutton`, `get_mposition`, `get_mposition_sub_pix`

See also

`open_window`

Module

Foundation

get_mposition (: : WindowHandle : Row, Column, Button)

Query the mouse position.

`get_mposition` returns the pixel accurate image coordinates of the mouse pointer in the output window and the mouse button pressed. These values are returned regardless of the state of the mouse buttons (pressed or not pressed). If at least one mouse button is pressed, the state of the modifier keys is returned as well. If no mouse button is pressed, `get_mposition` returns 0 in `Button`, even if a modifier key is pressed. The following values are assigned to the individual buttons and keys:

0: No button,

1: Left button,

2: Middle button,

4: Right button,

8: Shift key,

16: Ctrl key,

32: Alt key.

The sum of the values for all pressed buttons/keys is returned in `Button`.

The origin of the coordinate system is located in the left upper corner of the window. For graphics windows the coordinates `Row` and `Column` are expressed with consideration of the current image part (see `set_part`).

If subpixel accurate image coordinates are required, you can use the operator `get_mposition_sub_pix`.

Attention

`get_mposition` fails (returns `5` (`H_MSG_FAIL`)) if the mouse pointer is not located within the window. In this case, no values are returned.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row** (output_control) . . . . . point.y ⇝ *integer*
Row coordinate of the mouse cursor in the image coordinate system.

▷ **Column** (output_control) . . . . . point.x ⇝ *integer*
Column coordinate of the mouse cursor in the image coordinate system.

▷ **Button** (output_control) . . . . . integer ⇝ *integer*
Mouse button(s) pressed or 0.

Result

`get_mposition` returns the value `2` (`H_MSG_TRUE`). If the mouse pointer is not located within the window, `5` (`H_MSG_FAIL`) is returned.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Alternatives

`get_mbutton`, `get_mposition_sub_pix`, `get_mbutton_sub_pix`

See also

`open_window`

Module

Foundation

get_mposition_sub_pix (: : WindowHandle : Row, Column, Button)

Query the subpixel mouse position.

`get_mposition_sub_pix` returns the subpixel accurate image coordinates of the mouse pointer in the output window and the mouse button pressed. These values are returned regardless of the state of the mouse buttons (pressed or not pressed). If at least one mouse button is pressed, the state of the modifier keys is returned as well. If no mouse button is pressed, `get_mposition` returns 0 in `Button`, even if a modifier key is pressed. The following values are assigned to the individual buttons and keys:

0: No button,
1: Left button,
2: Middle button,
4: Right button,
8: Shift key,
16: Ctrl key,
32: Alt key.

The sum of the values for all pressed buttons/keys is returned in `Button`.

For graphics windows the coordinates `Row` and `Column` are expressed with consideration of the current image part (see `set_part`).

If only pixel accurate image coordinates are required, you can use the operator `get_mposition`.

Attention

`get_mposition_sub_pix` fails (returns `5` (`H_MSG_FAIL`)) if the mouse pointer is not located within the window. In this case, no values are returned.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row** (output_control) . . . . . point.y ⇝ *real*
Row coordinate of the mouse cursor in the image coordinate system.

▷ **Column** (output_control) . . . . . point.x ⇝ *real*
Column coordinate of the mouse cursor in the image coordinate system.

▷ **Button** (output_control) . . . . . integer ⇝ *integer*
Mouse button(s) pressed or 0.

Result

If the mouse pointer is not located within the window, `5` (`H_MSG_FAIL`) is returned.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Alternatives

`get_mbutton`, `get_mposition`, `get_mbutton_sub_pix`

See also

`open_window`

Module

Foundation

get_mshape (: : WindowHandle : Cursor)

Query the current mouse pointer shape.

This operator does not work in an HDevelop graphics window opened with `dev_open_window`.

`get_mshape` returns the name of the pointer shape set for the window. The mouse pointer shape can be used in the operator `set_mshape`.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Cursor** (output_control) . . . . . string ⇝ *string*
Mouse pointer name.

Result

`get_mshape` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `query_mshape`

Possible Successors

`set_mshape`

See also

set_mshape, query_mshape

Module

Foundation

query_mshape (: : WindowHandle : ShapeNames)

Query all available mouse pointer shapes.

This operator does not work in an HDevelop graphics window opened with dev_open_window.

query_mshape returns the names of all available mouse pointer shapes for the window. These can be used in the operator set_mshape. If no mouse pointers are available, the empty tuple is returned.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **ShapeNames** (output_control) ..... string-array ⇝ *string*
Available mouse pointer names.

Result

query_mshape returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, get_mshape

Possible Successors

set_mshape

See also

set_mshape, get_mshape

Module

Foundation

send_mouse_double_click_event (: : WindowHandle, Row, Column, Button : Processed)

Send an event to a buffer window signaling a mouse double click event.

send_mouse_double_click_event sends an event at the position (Row, Column) to the buffer window WindowHandle, signaling that the mouse button encoded by Button has been double clicked. Please refer to get_mbutton to see how to encode Button.

The purpose of this operator is to select a drawing object in a situation where multiple drawing objects are stacked on top of each other in a buffer window. If the double click happens on either one of the markers of the object border of the active object the object below is selected. To simplify the interaction with the drawing object, the coordinates Row and Column are given in the image coordinate system. To convert window coordinates into image coordinates, the operator convert_coordinates_window_to_image can be used.

If the event could be processed, send_mouse_double_click_event returns *'true'* in Processed. If no action for the event could be determined (e.g., because there is no drawing object below the mouse position), *'false'* is returned in Processed. In this case the caller may determine an action for the event.

The operators send_mouse_down_event, send_mouse_up_event, send_mouse_drag_event and send_mouse_double_click_event are the only means to manipulate drawing objects in buffer windows.

Attention

send_mouse_double_click_event depends on the library libcanvas, which might not be available on embedded systems.

Parameters

- ▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
  Window handle of the buffer window.
- ▷ **Row** (input_control) . . . . . point.y ⇝ *integer* / real
  Row coordinate of the mouse cursor in the image coordinate system.
- ▷ **Column** (input_control) . . . . . point.x ⇝ *integer* / real
  Column coordinate of the mouse cursor in the image coordinate system.
- ▷ **Button** (input_control) . . . . . integer ⇝ *integer*
  Mouse button(s) pressed.
- ▷ **Processed** (output_control) . . . . . string ⇝ *string*
  'true', if HALCON processed the event.

Result

send_mouse_double_click_event returns the value 2 (H_MSG_TRUE) if the window handle WindowHandle corresponds to a window of valid type, and the event is properly encoded. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

See also

send_mouse_up_event, send_mouse_down_event, open_window

Module

Foundation

send_mouse_down_event (: : WindowHandle, Row, Column, Button : Processed)

Send an event to a window buffer signaling a mouse down event.

send_mouse_down_event sends an event at the position (Row, Column) to the buffer window WindowHandle, signaling that the mouse button encoded by Button has been pressed. Please refer to get_mbutton to see how to encode Button.

The purpose of this operator is to manipulate drawing objects in a buffer window. If the coordinates Row and Column are pointing on a drawing object, it is activated and can be moved or changed by sending more events with the send_mouse_down_event operator. To simplify the interaction with the drawing object, the coordinates Row and Column are given in the image coordinate system. To convert window coordinates into image coordinates, the operator convert_coordinates_window_to_image can be used.

If the event could be processed, send_mouse_down_event returns *'true'* in Processed. If no action for the event could be determined (e.g., because there is no drawing object below the mouse position), *'false'* is returned in Processed. In this case the caller may determine an action for the event.

The operators send_mouse_down_event, send_mouse_up_event, send_mouse_drag_event and send_mouse_double_click_event are the only means to manipulate drawing objects in buffer windows.

Attention

send_mouse_down_event depends on the library libcanvas, which might not be available on embedded systems.

Parameters

- ▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
  Window handle of the buffer window.
- ▷ **Row** (input_control) . . . . . point.y ⇝ *integer* / real
  Row coordinate of the mouse cursor in the image coordinate system.
- ▷ **Column** (input_control) . . . . . point.x ⇝ *integer* / real
  Column coordinate of the mouse cursor in the image coordinate system.
- ▷ **Button** (input_control) . . . . . integer ⇝ *integer*
  Mouse button(s) pressed.
- ▷ **Processed** (output_control) . . . . . string ⇝ *string*
  'true', if HALCON processed the event.

Result

`send_mouse_down_event` returns the value `2` (`H_MSG_TRUE`) if the window handle `WindowHandle` corresponds to a window of valid type, and the event is properly encoded. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

See also

`send_mouse_up_event`, `send_mouse_double_click_event`, `open_window`

Module

Foundation

send_mouse_drag_event (: : WindowHandle, Row, Column, Button : Processed)

Send an event to a buffer window signaling a mouse drag event.

`send_mouse_drag_event` sends an event at the position (`Row`, `Column`) to the buffer window `WindowHandle`, signaling that the mouse button encoded by `Button` is been dragged. Please refer to `get_mbutton` to see how to encode `Button`.

If the event could be processed, `send_mouse_drag_event` returns *'true'* in `Processed`. If no action for the event could be determined (e.g., because there is no drawing object below the mouse position), *'false'* is returned in `Processed`. In this case the caller may determine an action for the event.

The operators `send_mouse_down_event`, `send_mouse_up_event`, `send_mouse_drag_event` and `send_mouse_double_click_event` are the only means to manipulate drawing objects in buffer windows.

Attention

`send_mouse_drag_event` depends on the library libcanvas, which might not be available on embedded systems.

Parameters

- ▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
  Window handle of the buffer window.
- ▷ **Row** (input_control) . . . . . point.y ⇝ *integer* / real
  Row coordinate of the mouse cursor in the image coordinate system.
- ▷ **Column** (input_control) . . . . . point.x ⇝ *integer* / real
  Column coordinate of the mouse cursor in the image coordinate system.
- ▷ **Button** (input_control) . . . . . integer ⇝ *integer*
  Mouse button(s) pressed.

▷ **Processed** (output_control) . . . . . string ⇝ *string*
'true', if HALCON processed the event.

Result

send_mouse_drag_event returns the value 2 (H_MSG_TRUE) if the window handle WindowHandle corresponds to a window of valid type, and the event is properly encoded. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

See also

send_mouse_up_event, send_mouse_down_event, send_mouse_double_click_event, open_window

Module

Foundation

send_mouse_up_event (: : WindowHandle, Row, Column, Button : Processed)

Send an event to a buffer window signaling a mouse up event.

send_mouse_up_event sends an event at the position (Row, Column) to the buffer window WindowHandle, signaling that the mouse button encoded by Button has been released. Please refer to get_mbutton to see how to encode Button.

The purpose of this operator is to release manipulated drawing objects in a buffer window. For example, a drawing object that has been moved by a (sequence of) send_mouse_down_event will be released. To simplify the interaction with the drawing object, the coordinates Row and Column are given in the image coordinate system. To convert window coordinates into image coordinates, the operator convert_coordinates_window_to_image can be used.

If the event could be processed, send_mouse_up_event returns *'true'* in Processed. If no action for the event could be determined (e.g., because there is no drawing object below the mouse position), *'false'* is returned in Processed. In this case the caller may determine an action for the event.

The operators send_mouse_down_event, send_mouse_up_event, send_mouse_drag_event and send_mouse_double_click_event are the only means to manipulate drawing objects in buffer windows.

Attention

send_mouse_up_event depends on the library libcanvas, which might not be available on embedded systems.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle of the buffer window.

▷ **Row** (input_control) . . . . . point.y ⇝ *integer* / real
Row coordinate of the mouse cursor in the image coordinate system.

▷ **Column** (input_control) . . . . . point.x ⇝ *integer* / real
Column coordinate of the mouse cursor in the image coordinate system.

▷ **Button** (input_control) . . . . . integer ⇝ *integer*
Mouse button(s) pressed.

▷ **Processed** (output_control) . . . . . string ⇝ *string*
'true', if HALCON processed the event.

Result

send_mouse_up_event returns the value 2 (H_MSG_TRUE) if the window handle WindowHandle corresponds to a window of valid type, and the event is properly encoded. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

See also

send_mouse_down_event, send_mouse_double_click_event, open_window

Module

Foundation

set_mshape (: : WindowHandle, Cursor :)

Set the current mouse pointer shape.

This operator does not work in an HDevelop graphics window opened with dev_open_window.

set_mshape sets the shape of the mouse pointer for the window. A list of the names of all available mouse pointer shapes can be obtained by calling query_mshape. The mouse pointer shape given by Cursor is used if the mouse pointer enters the window, irrespective of which window is the output window at present.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Cursor** (input_control) ..... string ⇝ *string*
Mouse pointer name.
Default: 'arrow'

Result

set_mshape returns the value 2 (H_MSG_TRUE) if the mouse pointer shape Cursor is defined for this window. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, query_mshape, get_mshape

See also

get_mshape, query_mshape

Module

Foundation

13.5 Object

attach_background_to_window (Image : : WindowHandle :)

Attach a background image to a HALCON window.

This operator attaches the background image passed in Image to a HALCON window defined in WindowHandle. The input image is copied. Thus it can be freed safely.

The background image is instantly displayed in the HALCON window when calling attach_background_to_window. All HALCON objects that were previously displayed in the window will still be displayed when calling this operator. Thus the previous window content will not be overwritten.

The window contents are displayed in the following order: first the background image, then HALCON objects, and finally the drawing objects. Thus, the drawing object is displayed always on top. Note that the window will be redrawn for each user interaction within the window and each call of set_part. HALCON objects can be deleted from the window with clear_window.

The background image can be removed from the window with the operator detach_background_from_window.

Attention

Note that using any synchronous operator which actively probe the event queue, e.g., get_mbutton or read_char, will conflict with the interaction with the drawing objects. In case the state of the cursor has to be read, please refer to the documentation of your framework of choice for an appropriate, non-invasive alternative.

Furthermore, the event based functionality should not be used together with the former blocking operators draw_rectangle1, draw_rectangle2, draw_region, draw_xld or draw_circle. They conflict with the event based functionality, since they actively fetch all events sent to the HALCON window.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Background image.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*

Window handle.

Result

If the window exists and the specified parameters are correct attach_background_to_window returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

detach_background_from_window

Module

Foundation

attach_drawing_object_to_window (: : WindowHandle, DrawHandle :)

Attach an existing drawing object to a HALCON window.

attach_drawing_object_to_window attaches the drawing object DrawHandle to the HALCON window specified in WindowHandle. When attached to a window the drawing object is displayed in the specified window. The drawing object can be modified interactively in the HALCON window. The window is automatically updated when the objects properties are modified using the operator set_drawing_object_params.

When two or more drawing objects overlap, a double-click on the currently selected drawing object will shift the focus to the next drawing object.

Furthermore it is possible to react to modifications of the drawing object caused by user interaction when using any of the language interface. Please see the documentation of set_drawing_object_callback for more details.

The drawing object can be removed from the current window with the operator detach_drawing_object_from_window.

As soon as a drawing object is attached to the window, each HALCON object which is displayed by any of the graphical operators, namely disp_obj, disp_region, disp_image and disp_xld, is stored internally in a graphical stack associated with the WindowHandle, so that they can be displayed together with the attached drawing object. These remain in the stack until the user calls clear_window or the window is closed.

The size of this graphical stack can be queried and modified with get_system and set_system, respectively. See the corresponding documentation reference for further details.

Attention

Note that using any synchronous operator which actively probe the event queue, e.g., get_mbutton or read_char, will conflict with the interaction with the drawing objects. In case you need to read the state of the cursor, please refer to the documentation of your framework of choice for an appropriate, non-invasive alternative.

Furthermore, the event based functionality should not be used together with the former blocking operators draw_rectangle1, draw_rectangle2, draw_region, draw_xld or draw_circle. They conflict with the event based functionality, since they actively fetch all events sent to the HALCON window.

When working under UNIX/Linux it is necessary to turn on the support for multithreading in the Xlib. This is achieved by calling the function XInitThreads() before any other function of the Xlib library. This means that you need to call it before any other function or method of your graphical development environment of choice. See the documentation of the function XInitThreads() in the corresponding manual page for further details.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **DrawHandle** (input_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

attach_drawing_object_to_window returns 2 (H_MSG_TRUE), if the DrawHandle and WindowHandle are valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_drawing_object_rectangle1, create_drawing_object_xld, create_drawing_object_circle_sector, create_drawing_object_ellipse_sector, create_drawing_object_ellipse, create_drawing_object_line, create_drawing_object_rectangle2, create_drawing_object_circle, create_drawing_object_text

Possible Successors

detach_drawing_object_from_window, get_drawing_object_params, get_drawing_object_iconic

See also

set_drawing_object_callback, get_drawing_object_iconic, get_drawing_object_params

Module

Foundation

clear_drawing_object (: : DrawID :)

Delete drawing object.

This operator deletes the drawing object DrawID. The object is automatically detached from the windows to which it had been previously attached. After calling clear_drawing_object the draw object can no longer be used.

Parameters

▷ **DrawID** (input_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

clear_drawing_object returns 2 (H_MSG_TRUE), if the DrawID is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DrawID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_drawing_object_rectangle1, create_drawing_object_rectangle2, create_drawing_object_ellipse, create_drawing_object_circle_sector, create_drawing_object_ellipse_sector, create_drawing_object_xld, create_drawing_object_line, create_drawing_object_circle, create_drawing_object_text, detach_drawing_object_from_window, attach_drawing_object_to_window

Possible Successors

clear_window, close_window

See also

create_drawing_object_rectangle1, create_drawing_object_rectangle2, create_drawing_object_ellipse, create_drawing_object_circle_sector, create_drawing_object_ellipse_sector, create_drawing_object_xld, create_drawing_object_line, create_drawing_object_circle, create_drawing_object_text

Module

Foundation

create_drawing_object_circle (: : Row, Column, Radius : DrawID)

Create a circle which can be modified interactively.

create_drawing_object_circle creates a circular region which can be interactively modified in a HALCON window by the user. The circle is defined by the coordinates Row and Column of its center, and by its Radius. The created circle is returned in the handle DrawID.

These parameters can be queried any time with get_drawing_object_params, as well as its corresponding HALCON object with get_drawing_object_iconic.

In addition to those parameters, every drawing object has a number of parameters which determine the appearance of the object in the HALCON window. See set_drawing_object_params for details on the number and meaning of those parameters.

In order to enable the interactive modification with the drawing object, the object must be attached to an existing window. See attach_drawing_object_to_window for further details. Once attached, the circle is editable by the user through interaction within the window. There are two possible transformations for a circle object: resizing and displacement. By clicking close to the circular arc you can modify the Radius of the circle. By clicking on the center, the circle can be dragged across the HALCON window.

In contrast to the operator draw_circle, this interaction does not block the calling thread.

Parameters

▷ **Row** (input_control) ..... circle.center.y ⇝ *real*
Row coordinate of the center.
Default: 100

▷ **Column** (input_control) ..... circle.center.x ⇝ *real*
Column coordinate of the center.
Default: 100

▷ **Radius** (input_control) ..... circle.radius ⇝ *real*
Radius of the circle.
Default: 80

▷ **DrawID** (output_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

If the parameter values are correct the operator create_drawing_object_circle returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

open_window

Possible Successors

attach_drawing_object_to_window, get_drawing_object_params, get_drawing_object_iconic

Alternatives

draw_circle, draw_region, create_drawing_object_circle_sector, create_drawing_object_ellipse, create_drawing_object_ellipse_sector

See also

create_drawing_object_rectangle1, create_drawing_object_rectangle2, create_drawing_object_ellipse, create_drawing_object_circle_sector, create_drawing_object_ellipse_sector, create_drawing_object_xld

Module

Foundation

create_drawing_object_circle_sector (: : Row, Column, Radius, StartAngle, EndAngle : DrawID)

Create a circle sector which can be modified interactively.

create_drawing_object_circle_sector creates a circle sector region which can be modified interactively in a HALCON window by the user. The circle sector is defined by the coordinates Row and Column of the center point, the Radius, and the start and end angle StartAngle and EndAngle.

These parameters can be queried any time with get_drawing_object_params, as well as its corresponding HALCON object with get_drawing_object_iconic.

In addition to those parameters, every drawing object has a number of parameters which determine the appearance of the object in the HALCON window. See set_drawing_object_params for details on the number and meaning of those parameters.

In order to enable the interactive modification with the drawing object, the object must be attached to an existing window. See attach_drawing_object_to_window for further details. Once attached, the circle is editable by the user through interaction with the window. There are three possible transformations for a circle sector object: resizing, displacement and changing the arc angle. The Radius of the circle can be modified by dragging the circular arc. By clicking on the center, the circle sector can be moved across the HALCON window. The angle arc can be modified by dragging the handles of the circle sector.

In contrast to the operator draw_circle, this interaction does not block the calling thread.

Parameters

▷ **Row** (input_control) ..... coordinates.y ⇝ *real*
Row coordinate of the center.
Default: 100

▷ **Column** (input_control) ..... coordinates.x ⇝ *real*
Column coordinate of the center.
Default: 100

▷ **Radius** (input_control) ..... number ⇝ *real*
Radius of the circle.
Default: 80

▷ **StartAngle** (input_control) ..... angle.rad ⇝ *real*
Start angle of the arc.
Default: 0

▷ **EndAngle** (input_control) ..... angle.rad ⇝ *real*
End angle of the arc.
Default: 3.14159

▷ **DrawID** (output_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

If the parameter values are correct the operator create_drawing_object_circle_sector returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

open_window

Possible Successors

attach_drawing_object_to_window, get_drawing_object_params, get_drawing_object_iconic

Alternatives

draw_rectangle1_mod, draw_rectangle2, draw_region, create_drawing_object_circle, create_drawing_object_ellipse_sector

See also

create_drawing_object_rectangle1, create_drawing_object_rectangle2, create_drawing_object_ellipse, create_drawing_object_circle, create_drawing_object_ellipse_sector, create_drawing_object_xld, create_drawing_object_line

Module

Foundation

create_drawing_object_ellipse (: : Row, Column, Phi, Radius1, Radius2 : DrawID)

Create an ellipse which can be modified interactively.

create_drawing_object_ellipse creates an ellipse which can be modified interactively in a HALCON window by the user. The ellipse is defined by the coordinates of its center point, Row and Column, the lengths of its respective half axis, Radius1 and Radius2, and its orientation Phi.

The parameters can be queried with get_drawing_object_params, as well as its corresponding HALCON object with get_drawing_object_iconic.

In addition to those parameters, every drawing object has a number of parameters which determine the appearance of the object in the HALCON window. See set_drawing_object_params for details on the number and meaning of those parameters.

In order to enable the interactive modification with the drawing object, the object must be attached to an existing window. See attach_drawing_object_to_window for further details. Once attached, the ellipse is editable by the user through interaction with the window. There are three possible transformations for an ellipse object: resizing, displacement and rotation. The ellipse can be resized by clicking close to the vertex handles. The orientation and thus the rotation of the ellipse can be modified by dragging on a vertex of the first half axis. The ellipse can be moved in the HALCON window by dragging the center.

In contrast to the operator draw_ellipse, this interaction does not block the calling thread.

Parameters

- ▷ **Row** (input_control) ..... ellipse.center.y ⇝ *real*
  Row index of the center.
  Default: 200
- ▷ **Column** (input_control) ..... ellipse.center.x ⇝ *real*
  Column index of the center.
  Default: 200
- ▷ **Phi** (input_control) ..... ellipse.angle.rad ⇝ *real*
  Orientation of the first half axis in radians.
  Default: 0
- ▷ **Radius1** (input_control) ..... ellipse.radius1 ⇝ *real*
  First half axis.
  Default: 100
- ▷ **Radius2** (input_control) ..... ellipse.radius2 ⇝ *real*
  Second half axis.
  Default: 60
- ▷ **DrawID** (output_control) ..... drawing_object ⇝ *handle*
  Handle of the drawing object.

Result

If the parameter values are correct the operator create_drawing_object_ellipse returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

open_window

Possible Successors

disp_region, set_colored, set_line_width, set_draw, set_insert, attach_drawing_object_to_window

Alternatives

draw_ellipse_mod, draw_ellipse, draw_region

See also

gen_ellipse, draw_circle, draw_rectangle2, set_insert

Module

Foundation

create_drawing_object_ellipse_sector (: : Row, Column, Phi, Radius1, Radius2, StartAngle, EndAngle : DrawID)

Create an elliptic sector which can be modified interactively.

create_drawing_object_ellipse_sector creates an ellipse sector which can be modified interactively in a HALCON window by the user. The ellipse sector is defined by the coordinates Row and Column of its center, the lengths of its respective half axis, Radius1 and Radius2, the start and end angles of the arc, StartAngle and EndAngle, and its orientation Phi.

These parameters can be queried any time with get_drawing_object_params, as well as its corresponding HALCON object with get_drawing_object_iconic.

In addition to those parameters, every drawing object has a number of parameters which determine the appearance of the object in the HALCON window. See set_drawing_object_params for details on the number and meaning of those parameters.

In order to enable the interactive modification with the drawing object, the object must be attached to an existing window. See attach_drawing_object_to_window for further details. Once attached, the ellipse sector is editable by the user through interaction with the window. There are four possible transformations for an ellipse sector object: resizing, displacement, rotation, and changing the arc angle. The ellipse sector can be moved in the HALCON window by dragging the center. The orientation and thus the rotation of the ellipse sector can also be modified by dragging on a vertex handle of the first half axis. The size and angles of the ellipse sector can be modified by dragging close to the vertex handles.

In contrast to the operator draw_ellipse, this interaction does not block the calling thread.

Parameters

▷ **Row** (input_control) . . . . . coordinates.y ⇝ *real*
Row index of the center.
Default: 200

▷ **Column** (input_control) . . . . . coordinates.x ⇝ *real*
Column index of the center.
Default: 200

▷ **Phi** (input_control) . . . . . angle.rad ⇝ *real*
Orientation of the first half axis in radians.
Default: 0

▷ **Radius1** (input_control) . . . . . number ⇝ *real*
First half axis.
Default: 100

▷ **Radius2** (input_control) . . . . . number ⇝ *real*
Second half axis.
Default: 60

▷ **StartAngle** (input_control) . . . . . angle.rad ⇝ *real*
Start angle of the arc.
Default: 0

▷ **EndAngle** (input_control) . . . . . angle.rad ⇝ *real*
End angle of the arc.
Default: 3.14159

▷ **DrawID** (output_control) . . . . . drawing_object ⇝ *handle*
Handle of the drawing object.

Result

If the parameter values are correct the operator `create_drawing_object_ellipse_sector` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`open_window`

Possible Successors

`attach_drawing_object_to_window`, `get_drawing_object_params`, `get_drawing_object_iconic`

Alternatives

`draw_ellipse_mod`, `draw_ellipse`, `draw_region`, `create_drawing_object_ellipse`

See also

`gen_ellipse_contour_xld`, `create_drawing_object_rectangle1`, `create_drawing_object_rectangle2`, `create_drawing_object_ellipse`, `create_drawing_object_xld`, `create_drawing_object_line`, `create_drawing_object_circle`

Module

Foundation

create_drawing_object_line (: : Row1, Column1, Row2, Column2 : DrawID)

Create a line which can be modified interactively.

`create_drawing_object_line` creates a line which can be modified interactively in a HALCON window by the user. The line is defined by the coordinates of the first line point, `Row1` and `Column1`, and the second line point, `Row2` and `Column2`.

These parameters can be queried any time with `get_drawing_object_params`, as well as its corresponding HALCON object with `get_drawing_object_iconic`.

In addition to those parameters, every drawing object has a number of parameters which determine the appearance of the object in the HALCON window. See `set_drawing_object_params` for details on the number and meaning of those parameters.

In order to enable the interactive modification with the drawing object, the object must be attached to an existing window. See `attach_drawing_object_to_window` for further details. Once attached, the line is editable by the user through interaction with the window. There are two possible transformations for a line object: resizing and displacement. A line point can be moved by dragging on the point handle. The whole line can be moved by dragging the center of the line.

In contrast to the operator `draw_line`, this interaction does not block the calling thread.

Parameters

▷ **Row1** (input_control) ..... line.begin.y ⇝ *real*
Row coordinate of the first line point.
Default: 100

▷ **Column1** (input_control) ..... line.begin.x ⇝ *real*
Column coordinate of the first line point.
Default: 100

▷ **Row2** (input_control) ..... line.end.y ⇝ *real*
Row coordinate of the second line point.
Default: 200

▷ **Column2** (input_control) ..... line.end.x ⇝ *real*
Column coordinate of the second line point.
Default: 200

▷ **DrawID** (output_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

If the parameter values are correct the operator `create_drawing_object_circle_sector` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`open_window`

Possible Successors

`attach_drawing_object_to_window`, `get_drawing_object_params`, `get_drawing_object_iconic`

Alternatives

`draw_line`, `draw_line_mod`, `create_drawing_object_xld`

See also

`create_drawing_object_rectangle1`, `create_drawing_object_rectangle2`, `create_drawing_object_ellipse`, `create_drawing_object_circle_sector`, `create_drawing_object_ellipse_sector`, `create_drawing_object_xld`

Module

Foundation

create_drawing_object_rectangle1 (: : Row1, Column1, Row2, Column2 : DrawID)

Create a rectangle parallel to the coordinate axis which can be modified interactively.

`create_drawing_object_rectangle1` creates a rectangle aligned along the coordinate axis which can be modified interactively in a HALCON window by the user. The rectangle is defined by the coordinates `Row1`, `Column1`, `Row2`, and `Column2` of its upper left and lower right corner. The created rectangle is returned in the handle `DrawID`.

These parameters can be queried any time with `get_drawing_object_params`, as well as its corresponding HALCON object with `get_drawing_object_iconic`.

In addition to those parameters, every drawing object has a number of parameters which determine the appearance of the object in the HALCON window. See `set_drawing_object_params` for details on the number and meaning of those parameters.

In order to enable the interactive modification with the drawing object, the object must be attached to an existing window. See `attach_drawing_object_to_window` for further details. Once attached, the rectangle is editable by the user through interaction with the window. There are two possible transformations for a rectangle1 object: resizing and displacement. The rectangle can be moved by dragging in the middle of the rectangle and resized by dragging the handles of the corners.

In contrast to the operator `draw_rectangle1`, this interaction does not block the calling thread.

Parameters

▷ **Row1** (input_control) . . . . . rectangle.origin.y ⇝ *real*
Row coordinate of the upper left corner.
Default: 100

▷ **Column1** (input_control) . . . . . rectangle.origin.x ⇝ *real*
Column coordinate of the upper left corner.
Default: 100

▷ **Row2** (input_control) . . . . . rectangle.corner.y ⇝ *real*
Row coordinate of the lower right corner.
Default: 200

▷ **Column2** (input_control) . . . . . rectangle.corner.x ⇝ *real*
Column coordinate of the lower right corner.
Default: 200

▷ **DrawID** (output_control) . . . . . drawing_object ⇝ *handle*
Handle of the drawing object.

Result

If the parameter values are correct, the operator `create_drawing_object_rectangle1` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`open_window`

Possible Successors

`disp_region`, `set_colored`, `set_line_width`, `set_draw`, `set_insert`, `attach_drawing_object_to_window`

Alternatives

`draw_rectangle1_mod`, `draw_rectangle2`, `draw_region`

See also

`gen_rectangle1`, `draw_circle`, `draw_ellipse`, `set_insert`

Module

Foundation

create_drawing_object_rectangle2 (: : Row, Column, Phi, Length1, Length2 : DrawID)

Create a rectangle of any orientation which can be modified interactively.

`create_drawing_object_rectangle2` creates an oriented rectangle which can be modified interactively in a HALCON window by the user. The rectangle is defined by the coordinates `Row` and `Column` of its center, its orientation `Phi`, and the lengths of its half axis, `Length1` and `Length2` respectively. The created rectangle is returned in the handle `DrawID`.

These parameters can be queried any time with `get_drawing_object_params`, as well as its corresponding HALCON object with `get_drawing_object_iconic`.

In addition to those parameters, every drawing object has a number of parameters which determine the appearance of the object in the HALCON window. See `set_drawing_object_params` for details on the number and meaning of those parameters.

In order to enable the interactive modification with the drawing object, the object must be attached to an existing window. See attach_drawing_object_to_window for further details. Once attached, the rectangle is editable by the user through interaction with the window. There are three possible transformations for a rectangle2 object: resizing, displacement and rotation. The rectangle can be moved by dragging its center. By dragging the side handles, the size of the rectangle can be modified. The rectangle's orientation can only be changed by gripping a side perpendicular to the first half axis.

In contrast to the operator draw_rectangle2, this interaction does not block the calling thread.

Parameters

▷ **Row** (input_control) ..... rectangle2.center.y ⇝ *real*
Row coordinate of the center.
Default: 150

▷ **Column** (input_control) ..... rectangle2.center.x ⇝ *real*
Column coordinate of the center.
Default: 150

▷ **Phi** (input_control) ..... rectangle2.angle.rad ⇝ *real*
Orientation of the first half axis in radians.
Default: 0

▷ **Length1** (input_control) ..... rectangle2.hwidth ⇝ *real*
First half axis.
Default: 100

▷ **Length2** (input_control) ..... rectangle2.hheight ⇝ *real*
Second half axis.
Default: 100

▷ **DrawID** (output_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

If the parameter values are correct the operator create_drawing_object_rectangle2 returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

open_window

Possible Successors

disp_region, set_colored, set_line_width, set_draw, set_insert, attach_drawing_object_to_window

Alternatives

draw_rectangle2_mod, draw_rectangle2, draw_region

See also

gen_rectangle2, draw_circle, draw_ellipse, set_insert

Module

Foundation

create_drawing_object_text (: : Row, Column, String : DrawID)

Create a text object which can be moved interactively.

create_drawing_object_text creates a text object which can be moved interactively across a HALCON window. The coordinates Row and Column define the position of the text.

These parameters can be queried any time with get_drawing_object_params, as well as its corresponding HALCON object with get_drawing_object_iconic.

In addition to those parameters, every drawing object has a number of parameters which determine the appearance of the object in the HALCON window. See set_drawing_object_params for details on the number and meaning of those parameters.

In order to enable the interactive modification with the drawing object, the text object must be attached to an existing window. See attach_drawing_object_to_window for further details. The text will be displayed with the current font settings of the HALCON window by default. However, the font setting can be adjusted with set_drawing_object_params. Once attached, the text can be moved by the user through interaction with the window.

Parameters

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Row coordinate of the text position.
Default: 12

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column coordinate of the text position.
Default: 12

▷ **String** (input_control) ..... string ⇝ *string*
Character string to be displayed.
Default: 'Text'

▷ **DrawID** (output_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

If the parameter values are correct, the operator create_drawing_object_text returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

open_window

Possible Successors

attach_drawing_object_to_window, get_drawing_object_params, set_drawing_object_params

Alternatives

write_string

See also

write_string, attach_drawing_object_to_window

Module

Foundation

create_drawing_object_xld (: : Row, Column : DrawID)

Create a XLD contour which can be modified interactively.

create_drawing_object_xld creates an XLD contour which can be modified interactively in a HALCON window by the user. The contour is defined by the coordinates Row and Column of its points.

These parameters can be queried any time with get_drawing_object_params, as well as its corresponding HALCON object with get_drawing_object_iconic.

In addition to those parameters, every drawing object has a number of parameters which determine the appearance of the object in the HALCON window. See set_drawing_object_params for details on the number and meaning of those parameters.

In order to enable the interactive modification with the drawing object, the object must be attached to an existing window. See attach_drawing_object_to_window for further details. Once attached, the contour is editable by the user through interaction with the window. Further points can be added by left clicking on the window. When clicking the contour segment between two consecutive points, a new point is added between those points. Each contour point can be moved by dragging its point handle. A single contour point can be deleted by dragging it over one of the neighboring contour points directly connected to it. The contour can be closed by moving the last point onto the first contour point. The contour can be moved as a whole by dragging it by the handle in the center. In addition, it is possible to assign a new contour to a drawing object with set_drawing_object_xld.

In contrast to the operator draw_xld, this interaction does not block the calling thread.

Parameters

▷ **Row** (input_control) . . . . . coordinates.y-array ⇝ *real* / integer
Row coordinates of the polygon.
Default: [100,200,200,100]
Suggested values: Row ∈ {0, 10, 20, 50, 100, 200, 500}

▷ **Column** (input_control) . . . . . coordinates.x-array ⇝ *real* / integer
Column coordinates of the polygon.
Default: [100,100,200,200]
Suggested values: Column ∈ {0, 10, 20, 50, 100, 200, 500}

▷ **DrawID** (output_control) . . . . . drawing_object ⇝ *handle*
Handle of the drawing object.

Result

If the parameter values are correct the operator create_drawing_object_xld returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

open_window

Possible Successors

attach_drawing_object_to_window, get_drawing_object_params, get_drawing_object_iconic, set_drawing_object_xld

Alternatives

draw_xld, draw_xld_mod, draw_region

See also

create_drawing_object_rectangle1, create_drawing_object_rectangle2, create_drawing_object_ellipse, create_drawing_object_circle_sector, create_drawing_object_ellipse_sector, create_drawing_object_line, create_drawing_object_circle

Module

Foundation

detach_background_from_window (: : WindowHandle :)

Detach the background image from a HALCON window.

This operator detaches the background image that was previously attached with attach_background_to_window from the HALCON window defined in WindowHandle. Thus the background of the window will be cleared.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

Result

If the window exists and the specified parameters are correct detach_background_from_window returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

attach_background_to_window

Module

Foundation

detach_drawing_object_from_window (: : WindowHandle, DrawHandle :)

Detach an existing drawing object from a HALCON window.

This operator can be used to detach the drawing object DrawHandle from the HALCON window WindowHandle to which it is currently attached. The drawing object will not longer be displayed in the window. If a callback function was defined for the drawing object on the event 'on_detach' or 'on_select' with set_drawing_object_callback this will also be called.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window Handle.

▷ **DrawHandle** (input_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

detach_drawing_object_from_window returns 2 (H_MSG_TRUE), if the DrawHandle and WindowHandle are valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

attach_drawing_object_to_window, get_drawing_object_iconic, get_drawing_object_params

Possible Successors

close_window, clear_drawing_object

See also

attach_drawing_object_to_window

Module

Foundation

get_drawing_object_iconic (: Object : DrawID :)

Return the iconic object of a drawing object.

get_drawing_object_iconic returns the iconic object Object that corresponds to the drawing object DrawID. In addition, the parameters that define the drawing object can be queried with the operator get_drawing_object_params.

Parameters

▷ **Object** (output_object) ..... object(-array) ⇝ *object*
Copy of the iconic object represented by the drawing object.

▷ **DrawID** (input_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

clear_drawing_object returns 2 (H_MSG_TRUE), if the DrawID is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_drawing_object_rectangle1, create_drawing_object_rectangle2, create_drawing_object_ellipse, create_drawing_object_circle, create_drawing_object_line, create_drawing_object_ellipse_sector, create_drawing_object_circle_sector, create_drawing_object_xld

Possible Successors

reduce_domain, gen_region_contour_xld, attach_drawing_object_to_window

Alternatives

get_drawing_object_params

See also

get_drawing_object_params, attach_drawing_object_to_window

Module

Foundation

get_drawing_object_params (: : DrawID, GenParamName : GenParamValue)

Get the parameters of a drawing object.

get_drawing_object_params can be used to query the parameters that describe the drawing object DrawID. One may specify one or several parameters which will be retrieved in the output tuple GenParamValue, in the same order as specified in the input tuple GenParamName.

In addition to the parameters described in set_drawing_object_params following parameters are possible for GenParamName:

'type': Returns the type of the drawing object. Possible return values are: *'circle'*, *'circle_sector'*, *'ellipse'*, *'ellipse_sector'*, *'rectangle1'*, *'rectangle2'*, *'line'*, *'xld'*, *'text'*

Parameters

▷ **DrawID** (input_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Parameter names of the drawing object.
List of values: GenParamName ∈ {'row1', 'row2', 'column1', 'column2', 'radius', 'row', 'column', 'length1', 'length2', 'string', 'phi', 'start_angle', 'end_angle', 'radius1', 'radius2', 'color', 'font', 'line_width', 'line_style', 'marker_size', 'type'}

▷ **GenParamValue** (output_control) . . . . . attribute.name(-array) ⇝ *real* / integer / string
Parameter values.

Result

get_drawing_object_params returns 2 (H_MSG_TRUE), if the DrawID is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_drawing_object_rectangle1, create_drawing_object_rectangle2, create_drawing_object_ellipse, create_drawing_object_circle, create_drawing_object_line, create_drawing_object_ellipse_sector, create_drawing_object_circle_sector, create_drawing_object_xld, create_drawing_object_text, attach_drawing_object_to_window

Possible Successors

set_drawing_object_params, attach_drawing_object_to_window, get_drawing_object_iconic

Alternatives

get_drawing_object_iconic

See also

attach_drawing_object_to_window, get_drawing_object_iconic, set_drawing_object_params

Module

Foundation

get_window_background_image (: BackgroundImage : WindowHandle :)

Gets a copy of the background image of the HALCON window.

get_window_background_image returns a copy of the background image BackgroundImage set in the window WindowHandle. This operator is specially useful in cases where the user would like to perform some operations in one of the callback functions of a drawing object attached to the window. See set_drawing_object_callback for more details.

Parameters

▷ **BackgroundImage** (output_object) . . . . . (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Copy of the background image.

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

Result

If the window exists and the specified parameters are correct get_window_background_image returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

attach_background_to_window

Possible Successors

close_window, detach_background_from_window, attach_drawing_object_to_window

See also

attach_background_to_window, detach_drawing_object_from_window

Module

Foundation

set_content_update_callback (: : WindowHandle, CallbackFunction, CallbackContext :)

Sets the callback for content updates in buffer window.

set_content_update_callback sets a callback CallbackFunction, which is called whenever the contents of the buffer window WindowHandle change. The parameter CallbackContext is always passed to the callback.

The type of CallbackFunction is: Herror callback(void *context)

On Windows systems, the __stdcall calling convention is used: Herror __stdcall callback(void *context)

The callback function must not call a HALCON operator. Doing so is undefined behavior. It is advised to perform only computationally inexpensive tasks in the callback, e.g., setting a flag or triggering an asynchronous operation.

In combination with the operators send_mouse_down_event, send_mouse_up_event, and send_mouse_double_click_event, set_content_update_callback can be used to create a buffer based control for a GUI Toolkit (such as Qt), where the control passes mouse events to the buffer, while the buffer notifies the control on changes by calling the function CallbackFunction. The control then can obtain the contents of the buffer with dump_window_image and display them.

Attention

set_content_update_callback depends on the library libcanvas, which might not be available on embedded systems.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **CallbackFunction** (input_control) ..... pointer ⇝ *integer*
  Callback for content updates.
- ▷ **CallbackContext** (input_control) ..... pointer ⇝ *integer* / handle
  Parameter to CallbackFunction.

Result

If the window exists and is a buffer window, set_content_update_callback returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

flush_buffer

Module

Foundation

set_drawing_object_callback (: : DrawHandle, DrawObjectEvent, CallbackFunction :)

Add a callback function to a drawing object.

set_drawing_object_callback adds the callback function CallbackFunction to be called on the event(s) defined in DrawObjectEvent to the drawing object DrawHandle. An event is an action, i.e., a user interaction with the mouse or a call to a HALCON operator, e.g., attach_drawing_object_to_window), that changes the current state of a given drawing object. The operator set_drawing_object_callback allows to define how to react to those events.

The following predefined events are available:

'on_attach' The corresponding callback will be called right after the object has been attached to the HALCON window and is ready for interaction.

'on_detach' The corresponding callback will be called right after the object has been detached from the window and is no longer available for interaction.

'on_drag' The corresponding callback will be called right after the object has been dragged with mouse interaction.

'on_resize' The corresponding callback will be called right after the object has been resized with mouse interaction.

'on_select' The corresponding callback will be called right after the object has been selected with mouse interaction.

It is possible to specify a particular callback for each desired event. Then the input parameters DrawObjectEvent and CallbackFunction must have the same length. It is also possible to use the same callback for different events. Then DrawObjectEvent has an arbitrary length and CallbackFunction contains one single value.

The callback is an integer of long type containing a pointer to a C function with the following signature:

Herror HDrawObjectCallback(Hphandle DrawHandle, Hphandle WindowHandle, char* type)

On Windows systems, the __stdcall naming convention is used:

Herror __stdcall HDrawObjectCallback(Hphandle DrawHandle, Hphandle WindowHandle, char* type)

The first parameter of the callback function contains the handle to the draw object that generated the event, the second one contains the handle of the window where the interaction took place. Finally, the third parameter indicates which event occurred.

Attention

No graphical operator should be called within the callback, like for example disp_obj, for otherwise a deadlock may occur.

Parameters

▷ **DrawHandle** (input_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

▷ **DrawObjectEvent** (input_control) ..... string(-array) ⇝ *string*
Events to be captured.
Suggested values: DrawObjectEvent ∈ {'on_resize', 'on_drag', 'on_attach', 'on_detach', 'on_select'}

▷ **CallbackFunction** (input_control) ..... pointer(-array) ⇝ *integer*
Callback functions.

Result

set_drawing_object_callback returns 2 (H_MSG_TRUE), if DrawHandle is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

attach_drawing_object_to_window

Possible Successors

close_window, clear_drawing_object, detach_drawing_object_from_window, get_drawing_object_iconic, get_drawing_object_params

See also

create_drawing_object_rectangle1, attach_drawing_object_to_window, detach_drawing_object_from_window

Module

Foundation

set_drawing_object_params (: : DrawID, GenParamName, GenParamValue :)

Set the parameters of a drawing object.

set_drawing_object_params is used to set the parameters GenParamName of the drawing object DrawID to the values specified in GenParamValue.

Depending on the type of the drawing object the following parameters can be set for the different types of drawing object:

circle: *'row'*: Row coordinate of the center of the circle.
'column': Column coordinate of the center of the circle.
'radius': Radius of the circle.

circle_sector: *'row'*: Row coordinate of the center.
'column': Column coordinate of the center.
'radius': Radius of the circle.
'start_angle': Start angle of the circle arc.
'end_angle': End angle of the circle arc.

ellipse: *'row'*: Row coordinate of the center of the ellipse.
'column': Column coordinate of the center of the ellipse.
'phi': Orientation of the first half axis in radians.
'radius1': Length of the first half axis.
'radius2': Length of the second half axis.

ellipse_sector: *'row'*: Row coordinate of the center of the ellipse sector.
'column': Column coordinate of the center of the ellipse sector.
'phi': Orientation of the first half axis in radians.
'radius1': Length of the first half axis.
'radius2': Length of the second half axis.
'start_angle': Starting angle of the ellipse arc.
'end_angle': End angle of the ellipse arc.

rectangle1: *'row1'*: Row coordinate of the upper left corner.
'column1': Column coordinate of the upper left corner.
'row2': Row coordinate of the lower right corner.
'column2': Column coordinate of the lower right corner.

rectangle2: *'row'*: Row coordinate of the center of the rectangle.
'column': Column coordinate of the center of the rectangle.
'phi': Orientation of the first half axis in radians.
'length1': Length of the first half axis.
'length2': Length of the second half axis.

line: *'row1'*: Row coordinate of the first line point.
'column1': Column coordinate of the first line point.
'row2': Row coordinate of the second line point.
'column2': Column coordinate of the second line point.

xld: *'row'*: Row coordinates of the contour points.
'column': Column coordinates of the contour points.
It is possible to either set only the row or column coordinates of the contour points, or both at the same time. In the first case, the number of elements in the input tuple must be the same as the number of points in the contour. In the second case, all existing contour points are discarded and replaced by new ones. The new coordinates are specified in `GenParamValue` in the following order: *[Row_0, Row_1, ..., Row_n, Column_0, Column_1, ... Column_n]*.

text: *'row'*: Row coordinate of the text position.
'column': Column coordinate of the text position.
'string': Text string to be displayed.

In addition, there is an additional number of attributes that model the appearance of the drawing object in the window it is attached to. Depending on the type of the drawing object the following attributes can be set.

- Draw objects of type 'text':
  'color': Set the color of a text object. The color can be specified either by name, e.g., 'green', or by RGB values, e.g., '#ffb529'. See `set_color` for more information.
  'font': Set the font of a text object. A list of all available fonts can be queried with `query_font`. See `set_font` for a detailed description of the available values.
- Draw objects of type 'circle', 'circle_sector', 'ellipse', 'ellipse_sector', 'rectangle1', 'rectangle1', 'xld', 'line':
  'color': Set the color of a drawing object. The color can be specified either by name, e.g., 'green', or by RGB values, e.g., '#ffb529'. See `set_color` for more information.
  'line_style': Set the line style of the drawing object. This attribute requires at least one pair of attribute values in form of a tuple. The first defines the length of the visible part, the second defines the length of the invisible part. See `set_line_style` for more details. Please notice that since the length of the input tuple may vary, *'line_style'* must not be combined with other parameters or attributes in the same call. Otherwise an exception will the thrown.
  'line_width': Set the line width of the drawing object in pixel. See `set_line_width` for more information.
  'marker_size': Set the width and height of the markers used as anchor points for manipulating this drawing object.
  Value range: [*7 ... 32768*]

If more than one parameter shall be set, they have to be passed as a tuple in `GenParamName` containing the name of the parameters. The corresponding values have to be passed as a tuple in `GenParamValue`.

If the drawing object is currently attached to a window, this window is redrawn for each call of `set_drawing_object_params`.

Parameters

▷ **DrawID** (input_control) . . . . . drawing_object ⇝ *handle*
Handle of the drawing object.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Parameter names of the drawing object.
List of values: `GenParamName` ∈ {'row1', 'row2', 'column1', 'column2', 'radius', 'row', 'column', 'length1', 'length2', 'string', 'phi', 'start_angle', 'end_angle', 'radius1', 'radius2', 'color', 'font', 'line_width', 'line_style', 'marker_size'}

▷ **GenParamValue** (input_control) . . . . . attribute.name(-array) ⇝ *real* / integer / string
Parameter values.

Result

set_drawing_object_params returns 2 (H_MSG_TRUE), if the DrawID is valid, and the pairs of names and values of the tuples GenParamName and GenParamValue are coherent with the corresponding drawing object type. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_drawing_object_rectangle1, create_drawing_object_rectangle2, create_drawing_object_ellipse, create_drawing_object_circle, create_drawing_object_line, create_drawing_object_ellipse_sector, create_drawing_object_circle_sector, create_drawing_object_xld, attach_drawing_object_to_window

Possible Successors

get_drawing_object_params, attach_drawing_object_to_window, get_drawing_object_iconic

See also

attach_drawing_object_to_window, get_drawing_object_iconic, set_drawing_object_callback

Module

Foundation

set_drawing_object_xld (Contour : : DrawID :)

Set the contour of an interactive draw XLD.

set_drawing_object_xld substitutes the XLD currently contained in the draw object by the new Contour. The contour must contain at least two points, so that the interaction as described in create_drawing_object_xld is possible.

In addition, if the drawing object DrawID is currently attached to a HALCON window, the window will be redrawn with each call of set_drawing_object_xld.

Parameters

▷ **Contour** (input_object) ..... xld_cont ⇝ *object*
XLD contour.

▷ **DrawID** (input_control) ..... drawing_object ⇝ *handle*
Handle of the drawing object.

Result

set_drawing_object_xld returns 2 (H_MSG_TRUE), if the DrawID is valid and Contour is a contour consisting of at least two points. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_drawing_object_xld

Possible Successors

attach_drawing_object_to_window, clear_drawing_object

Alternatives

`create_drawing_object_xld`

See also

`attach_drawing_object_to_window`

Module

Foundation

13.6 Output

disp_arc (: : WindowHandle, CenterRow, CenterCol, Angle, BeginRow, BeginCol :)

Displays circular arcs in a window.

`disp_arc` displays one or several circular arcs in the output window. An arc is described by its center point (`CenterRow`,`CenterCol`), the angle between start and end of the arc (`Angle` in radians) and the first point of the arc (`BeginRow`,`BeginCol`). The arc is displayed in clockwise direction. The parameters for output can be determined - as with the output of regions - with the operators `set_color`, `set_gray`, `set_draw`, etc. It is possible to draw several arcs with one call by using tuple parameters. For the use of colors with several arcs, see `set_color`.

Attention

The center point has to be within the window. The radius of the arc has be at least 2 pixel.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **CenterRow** (input_control) . . . . . arc.center.y ⇝ *real* / integer
Row coordinate of center point.
Default: 64
Suggested values: CenterRow ∈ {0, 64, 128, 256}
Value range: 0 ≤ CenterRow ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **CenterCol** (input_control) . . . . . arc.center.x ⇝ *real* / integer
Column coordinate of center point.
Default: 64
Suggested values: CenterCol ∈ {0, 64, 128, 256}
Value range: 0 ≤ CenterCol ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Angle** (input_control) . . . . . arc.angle.rad ⇝ *real* / integer
Angle between start and end of the arc (in radians).
Default: 3.1415926
Suggested values: Angle ∈ {0.0, 0.785398, 1.570796, 3.1415926, 6.283185}
Value range: 0.0 ≤ Angle ≤ 6.283185 (lin)
Minimum increment: 0.01
Recommended increment: 0.1
Restriction: Angle > 0.0

▷ **BeginRow** (input_control) . . . . . arc.begin.y(-array) ⇝ *integer* / real
Row coordinate of the start of the arc.
Default: 32
Suggested values: BeginRow ∈ {0, 64, 128, 256}
Value range: 0 ≤ BeginRow ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **BeginCol** (input_control) . . . . . arc.begin.x(-array) ⇝ *integer* / real
Column coordinate of the start of the arc.
Default: 32
Suggested values: BeginCol ∈ {0, 64, 128, 256}
Value range: 0 ≤ BeginCol ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Example

```
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
set_draw(WindowHandle,'fill')
set_color(WindowHandle,'white')
Row := 100
Column := 100
disp_arc(WindowHandle,Row,Column,3.14,Row+10,Column+10)
close_window(WindowHandle)
```

Result

disp_arc returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_draw, set_color, set_colored, set_line_width, set_rgb, set_hsi

Alternatives

disp_circle, disp_ellipse, disp_region, gen_circle, gen_ellipse

See also

open_window, set_color, set_draw, set_rgb, set_hsi

Module

Foundation

disp_arrow (: : WindowHandle, Row1, Column1, Row2, Column2, Size :)

Displays arrows in a window.

disp_arrow displays one or several arrows in the output window. An arrow is described by the coordinates of the start (Row1,Column1) and the end (Row2,Column2). An arrowhead is displayed at the end of the arrow. The size of the arrowhead is specified by the parameter Size. If the arrow consists of just one point (start = end) nothing is displayed. The operators used to control the display of regions (e.g., set_draw, set_color, set_line_width) can also be used with arrows. Several arrows can be displayed with one call by using tuple parameters. For the use of colors with several arcs, see set_color.

Attention

The start and the end of the arrows must fall within the window.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row1** (input_control) . . . . . line.begin.y(-array) ⇝ *real* / integer
Row index of the start.
Default: 10.0
Suggested values: Row1 ∈ {0.0, 64.0, 128.0, 256.0}
Value range: 0.0 ≤ Row1 ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 1.0

▷ **Column1** (input_control) . . . . . line.begin.x(-array) ⇝ *real* / integer
Column index of the start.
Default: 10.0
Suggested values: Column1 ∈ {0.0, 64.0, 128.0, 256.0}
Value range: 0.0 ≤ Column1 ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 1.0

▷ **Row2** (input_control) . . . . . line.end.y(-array) ⇝ *real* / integer
Row index of the end.
Default: 118.0
Suggested values: Row2 ∈ {0.0, 64.0, 128.0, 256.0}
Value range: 0.0 ≤ Row2 ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 1.0

▷ **Column2** (input_control) . . . . . line.end.x(-array) ⇝ *real* / integer
Column index of the end.
Default: 118.0
Suggested values: Column2 ∈ {0.0, 64.0, 128.0, 256.0}
Value range: 0.0 ≤ Column2 ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 1.0

▷ **Size** (input_control) . . . . . number ⇝ *real* / integer
Size of the arrowhead.
Default: 1.0
Suggested values: Size ∈ {1.0, 2.0, 3.0, 5.0}
Value range: 0.0 ≤ Size ≤ 20.0 (lin)
Minimum increment: 1.0
Recommended increment: 1.0
Restriction: Size > 0.0

Example

```
set_color(WindowHandle,['red','green'])
disp_arrow(WindowHandle,[10,10],[10,10],[118,110],[118,118],1.0)
```

Result

disp_arrow returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_draw, set_color, set_colored, set_line_width, set_rgb, set_hsi

Alternatives

disp_line, gen_region_polygon, disp_region

See also

open_window, set_color, set_draw, set_line_width

Module

Foundation

disp_channel (MultichannelImage : : WindowHandle, Channel :)

Displays images with several channels.

disp_channel displays an image in the output window. It is possible to display several images with one call. In this case the images are displayed one after another. If the definition domains of the images overlap only the last image is visible. The parameter Channel defines the number of the channel that is displayed. For RGB-images the three color channels have to be used within a tuple parameter. For more information see disp_image.

Parameters

▷ **MultichannelImage** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multichannel images to be displayed.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*

Window handle.

▷ **Channel** (input_control) ..... integer(-array) ⇝ *integer*

Number of channel or the numbers of the RGB-channels

Default: 1

Suggested values: Channel ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Example

```
read_image(Image,'patras')
count_channels (Image, Channels)
for I := 1 to Channels by 1
    disp_channel (Image, WindowHandle, I)
endfor
```

Result

If the used images contain valid values and a correct output mode is set, disp_channel returns 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_rgb, set_lut, set_hsi

Alternatives

disp_image, disp_color

See also

open_window, reset_obj_db, set_lut, dump_window

Module

Foundation

disp_circle (: : WindowHandle, Row, Column, Radius :)

Displays circles in a window.

disp_circle displays one or several circles in the output window. A circle is described by the center (Row, Column) and the radius Radius. If the used coordinates are not within the window the circle is clipped accordingly.

The operators used to control the display of regions (e.g., set_draw, set_gray, set_draw) can also be used with circles. Several circles can be displayed with one call by using tuple parameters. For the use of colors with several circles, see set_color.

Attention

The center of the circle must be within the window.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row** (input_control) ..... circle.center.y(-array) ⇝ *real* / integer
Row index of the center.
Default: 64
Suggested values: Row ∈ {0, 64, 128, 256}
Value range: 0 ≤ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) ..... circle.center.x(-array) ⇝ *real* / integer
Column index of the center.
Default: 64
Suggested values: Column ∈ {0, 64, 128, 256}
Value range: 0 ≤ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Radius** (input_control) ..... circle.radius(-array) ⇝ *real* / integer
Radius of the circle.
Default: 64
Suggested values: Radius ∈ {0, 64, 128, 256}
Value range: 0 ≤ Radius ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1
Restriction: Radius > 0.0

Example

```
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
set_draw(WindowHandle,'fill')
set_color(WindowHandle,'white')
repeat
  get_mbutton(WindowHandle,Row,Column,Button)
  disp_circle(WindowHandle,Row,Column,(Row + Column) % 50)
until(Button == 1)
close_window(WindowHandle)
```

Result

disp_circle returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_draw, set_color, set_colored, set_line_width, set_rgb, set_hsi

Alternatives

disp_ellipse, disp_region, gen_circle, gen_ellipse

See also

open_window, set_color, set_draw, set_rgb, set_hsi

Module

Foundation

disp_color (ColorImage : : WindowHandle :)

Displays a color (RGB) image

disp_color displays the three channels of a color image in the output window. The channels are ordered in the sequence (red,green,blue). disp_color can be simulated by disp_channel.

Attention

Due to the restricted number of available colors the color appearance is usually different from the original.

Parameters

▷ **ColorImage** (input_object) ..... multichannel-image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Color image to display.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*

Window handle.

Example

```
/* disp_color(ColorImage) is identical to: */
Herror my_disp_color(Hobject ColorImage, Htuple *WindowHandle) {
  Htuple  Tupel;
  create_tuple(&Tupel,3);
  set_i(Tupel,1,0);
  set_i(Tupel,2,1);
  set_i(Tupel,3,2);
  T_disp_channel(ColorImage,*WindowHandle,Tupel);
  destroy_tuple(Tupel);
}
```

Result

If the used image contains valid values and a correct output mode is set, disp_color returns 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_rgb, set_lut, set_hsi

Alternatives

disp_channel, disp_obj

See also

disp_image, open_window, reset_obj_db, set_lut, dump_window

Module

Foundation

disp_cross (: : WindowHandle, Row, Column, Size, Angle :)

Displays crosses in a window.

disp_cross displays one or several crosses in the output window. A cross is described by the coordinates of the center point (Row,Column), the length of its bars Size and the orientation Angle. The operators used to control the display of regions (e.g., set_color, set_gray, set_draw, set_line_width) can also be used with crosses. Several crosses can be displayed with one call by using tuple parameters. For the use of colors with several crosses, see set_color.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row** (input_control) . . . . . coordinates.y(-array) ⇝ *real*
Row coordinate of the center.
Default: 32.0
Suggested values: Row ∈ {0.0, 64.0, 128.0, 256.0, 511.0}

▷ **Column** (input_control) . . . . . coordinates.x(-array) ⇝ *real*
Column coordinate of the center.
Default: 32.0
Suggested values: Column ∈ {0.0, 64.0, 128.0, 256.0, 511.0}

▷ **Size** (input_control) . . . . . number ⇝ *real*
Length of the bars.
Default: 6.0
Suggested values: Size ∈ {4.0, 6.0, 8.0, 10.0}
Value range: 0.0 ≤ Size

▷ **Angle** (input_control) . . . . . angle.rad ⇝ *real*
Orientation.
Default: 0.0
Suggested values: Angle ∈ {0.0, 0.78539816339744830961566084581988}

Result

disp_cross returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_rgb, set_lut, set_hsi, set_draw, set_color, set_colored, set_line_width

Alternatives

disp_arrow, disp_rectangle1, disp_rectangle2, disp_circle

See also

open_window, set_color, set_rgb, set_hsi, set_insert, set_line_width

Module

Foundation

disp_ellipse (: : WindowHandle, CenterRow, CenterCol, Phi, Radius1, Radius2 :)

Displays ellipses.

disp_ellipse displays one or several ellipses in the output window. An ellipse is described by the center (CenterRow, CenterCol), the orientation Phi (in radians) and the radii of the major and the minor axis (Radius1 and Radius2).

The operators used to control the display of regions (e.g., set_draw, set_gray, set_draw) can also be used with ellipses. Several ellipses can be displayed with one call by using tuple parameters. For the use of colors with several ellipses, see set_color.

Attention

The center of the ellipse must be within the window.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **CenterRow** (input_control) . . . . . ellipse.center.y(-array) ⇝ *integer*
Row index of center.
Default: 64
Suggested values: CenterRow ∈ {0, 64, 128, 256}
Value range: 0 ≤ CenterRow ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **CenterCol** (input_control) . . . . . ellipse.center.x(-array) ⇝ *integer*
Column index of center.
Default: 64
Suggested values: CenterCol ∈ {0, 64, 128, 256}
Value range: 0 ≤ CenterCol ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Phi** (input_control) . . . . . ellipse.angle.rad(-array) ⇝ *real* / integer
Orientation of the ellipse in radians
Default: 0.0
Suggested values: Phi ∈ {0.0, 0.785398, 1.570796, 3.1415926, 6.283185}
Value range: 0.0 ≤ Phi ≤ 6.283185 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **Radius1** (input_control) . . . . . ellipse.radius1(-array) ⇝ *real* / integer
Radius of major axis.
Default: 24.0
Suggested values: Radius1 ∈ {0.0, 64.0, 128.0, 256.0}
Value range: 0.0 ≤ Radius1 ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Radius2** (input_control) . . . . . ellipse.radius2(-array) ⇝ *real* / integer
Radius of minor axis.
Default: 14.0
Suggested values: Radius2 ∈ {0.0, 64.0, 128.0, 256.0}
Value range: 0.0 ≤ Radius2 ≤ 511.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

Example

```
set_color(WindowHandle,'red')
draw_region(MyRegion,WindowHandle)
elliptic_axis(MyRegion,Ra,Rb,Phi)
area_center(MyRegion,_,Row,Column)
disp_ellipse(WindowHandle,Row,Column,Phi,Ra,Rb)
```

Result

disp_ellipse returns 2 (H_MSG_TRUE), if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_draw, set_color, set_colored, set_line_width, set_rgb, set_hsi, elliptic_axis, area_center

Alternatives

disp_circle, disp_region, gen_ellipse, gen_circle

See also

open_window, set_color, set_rgb, set_hsi, set_draw, set_line_width

Module

Foundation

disp_image (Image : : WindowHandle :)

Displays gray value images.

disp_image displays the gray values of an image in the output window. The gray value pixels of the definition domain (set_comprise(::WindowHandle,'object':)) or of the whole image (set_comprise(::WindowHandle,'image':)) are used. Restriction to the definition domain is the default.

For the display of gray value images the number of gray values is usually reduced. This is due to the fact that colors have to be reserved for the display of graphics (e.g., set_color) and the window manager. Also depending on the number of bit planes on the used output device often less than 256 colors (eight bit planes) are available. The number of "colors" actually reserved for the display of gray values can be queried by get_system. Before opening the first window this value can be modified by set_system. For instance for 8 bit planes 200 real gray values are the default.

The reduction of the number of gray values does not pose problems as long as only gray value information is displayed, humans cannot distinguish 256 different shades of gray. If certain gray values are used for the representation of region information (which is not the style commonly used in HALCON), confusions might be the result, since different numerical values are displayed on the screen with the same gray value. The operator label_to_region should be used on these images in order to transform the label data into HALCON objects.

If images of type int2, int4, int8, real or complex are displayed, the smallest and largest gray value is computed. For images of the type complex this computation is based on the corresponding power spectrum. Afterwards the pixel data is rescaled according to the number of available gray values (depending on the output device, e.g., 200). It is possible that some pixels have a very different value than the other pixels. This might lead to the display of an (almost) completely white or black image. In order to decide if the current image is a binary image min_max_gray can be used. If necessary the image can be transformed or converted by scale_image and convert_image_type before it is displayed.

Attention

If a wrong output mode was set by set_paint, the error will be reported when disp_image is used.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Gray value image to display.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*

Window handle.

Example

```
* Output of a gray image:
read_image(Image,'monkey')
disp_image(Image,WindowHandle)
```

Result

If the used image contains valid values and a correct output mode is set, disp_image returns 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_rgb, set_lut, set_hsi, scale_image, convert_image_type, min_max_gray

Alternatives

disp_obj, disp_color

See also

open_window, reset_obj_db, set_paint, set_lut, paint_gray, scale_image, convert_image_type, dump_window

Module

Foundation

disp_line (: : WindowHandle, Row1, Column1, Row2, Column2 :)

Draws lines in a window.

disp_line displays one or several lines in the output window. A line is described by the coordinates of the start (Row1,Column1) and the coordinates of the end (Row2,Column2). The operators used to control the display of regions (e.g., set_color, set_gray, set_draw, set_line_width) can also be used with lines. Several lines can be displayed with one call by using tuple parameters. For the use of colors with several lines, see set_color.

Attention

The starting points and the ending points of the lines must be in the window.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row1** (input_control) ..... line.begin.y(-array) ⇝ *real*
Row index of the start.
Default: 32.0
Suggested values: Row1 ∈ {0.0, 64.0, 128.0, 256.0, 511.0}
Value range: 0 ≤ Row1 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Column1** (input_control) ..... line.begin.x(-array) ⇝ *real*
Column index of the start.
Default: 32.0
Suggested values: Column1 ∈ {0.0, 64.0, 128.0, 256.0, 511.0}
Value range: 0 ≤ Column1 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Row2** (input_control) ..... line.end.y(-array) ⇝ *real*
Row index of end.
Default: 64.0
Suggested values: Row2 ∈ {0.0, 64.0, 128.0, 256.0, 511.0}
Value range: 0 ≤ Row2 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Column2** (input_control) ..... line.end.x(-array) ⇝ *real*
Column index of end.
Default: 64.0
Suggested values: Column2 ∈ {0.0, 64.0, 128.0, 256.0, 511.0}
Value range: 0 ≤ Column2 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

Example

```
* Display contour of a rectangle
disp_line(WindowHandle,Row1,Column1,Row1,Column2)
disp_line(WindowHandle,Row1,Column2,Row2,Column2)
disp_line(WindowHandle,Row2,Column2,Row2,Column1)
disp_line(WindowHandle,Row2,Column1,Row1,Column1)
```

Result

disp_line returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_rgb, set_lut, set_hsi, set_draw, set_color, set_colored, set_line_width

Alternatives

disp_arrow, disp_rectangle1, disp_rectangle2, disp_region, gen_region_polygon, gen_region_points

See also

open_window, set_color, set_rgb, set_hsi, set_insert, set_line_width

Module

Foundation

disp_obj (Object : : WindowHandle :)

Displays image objects (image, region, XLD).

disp_obj displays objects depending of their kind. disp_obj is equivalent to disp_image for one channel images, equivalent to disp_color for three channel images, equivalent to disp_region for regions and equivalent to disp_xld for XLDs.

Parameters

▷ **Object** (input_object) ..... object(-array) ⇝ *object*
Image object to be displayed.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

Example

```
* Output of a gray image:
read_image(Image1,'monkey')
disp_obj(Image1,WindowHandle)
threshold(Image,Region,0,128)
disp_obj(Region,WindowHandle)
```

Result

If the used object is valid and a correct output mode is set, disp_obj returns 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_rgb, set_lut, set_hsi, scale_image, convert_image_type, min_max_gray

Alternatives

disp_color, disp_image, disp_xld, disp_region

See also

open_window, reset_obj_db, set_paint, set_lut, paint_gray, scale_image, convert_image_type, dump_window

Module

Foundation

disp_object_model_3d (: : WindowHandle, ObjectModel3D, CamParam, Pose, GenParamName, GenParamValue :)

Display 3D object models.

disp_object_model_3d displays the 3D object models of ObjectModel3D in the window WindowHandle. Set CamParam and the individual poses (in Pose) of the 3D object models to setup the displayed scene. If an empty tuple is given for CamParam, disp_object_model_3d uses default camera parameters that correspond to the window size. If CamParam is not empty, note that sometimes the aspect ratio of WindowHandle should be similar to width and height stored in CamParam in order to obtain the wanted results. Pose can contain either multiple poses (one pose for each 3D object model) or one pose for all 3D object models. Thereby the poses are in the form ${}^{ccs}\mathbf{P}_{mcs}$, where *ccs* denotes the camera coordinate system and *mcs* the model coordinate system (which is a 3D world coordinate system), see Transformations / Poses and "Solution Guide III-C - 3D Vision".

If an empty tuple is given for Pose, disp_object_model_3d estimates a pose, such that all 3D object models are visible.

3D object models containing triangles or polygons are displayed as solids, whereas 3D object models containing only coordinates are displayed as point clouds. 3D primitives (e.g., obtained by fit_primitives_object_model_3d) are displayed as solids as well. The generic parameter *'attribute'* can be used to explicitly select in which way a 3D object model is visualized.

To render the 3D object models into an image, the operator render_object_model_3d can be used.

GenParamName and GenParamValue are used to further configure the scene. The following values influence the whole scene:

'disp_background': Flag, if the current window content should be used as background.
List of values: *'true'*, *'false'*.
Default: *'false'*.

'light_position': Position of the light source. Must be given as a string containing four space separated floating point numbers. If the fourth number is 0.0, a directional light source is used (the first three components represent the direction), otherwise a point light source is used (with the first three components representing the position).
Default: *'-100.0 -100.0 0.0 1.0'*.

'light_ambient': Ambient part of the light source. Must be given as a string containing three space separated floating point numbers.
Default: *'0.2 0.2 0.2'*.

'light_diffuse': Diffuse part of the light source. Must be given as a string containing three space separated floating point numbers.

Default: *'0.8 0.8 0.8'*.

'colored': Display 3D object models in different colors. The value of this parameter defines the number of colors that are used.

List of values: *3*, *6*, *12*.

Default: all objects are displayed white.

'object_index_persistence': Must be set to *'true'* to enable the object index query with `get_disp_object_model_3d_info`.

List of values: *'true'*, *'false'*.

Default: *'false'*.

'depth_persistence': Must be set to *'true'* to enable the depth query with `get_disp_object_model_3d_info`.

List of values: *'true'*, *'false'*.

Default: *'false'*.

'quality': Must be set to *'low'* to enable faster rendering without anti aliasing.

List of values: *'low'*, *'high'*.

Default: *'high'*.

The following parameters can be set for all 3D object models in the scene or for a specific 3D object model by appending the index of the 3D object model to the parameter name (e.g., *'color_0'* to set the color of the first 3D object model).

'attribute' Explicitly select in which way a 3D object model is visualized.

List of values: *'auto'*, *'faces'*, *'primitive'*, *'points'*, *'lines'*.

Default: *'auto'*.

'color': Color of the 3D object models. The available colors can be queried with the operator `query_color`. In addition, the color may be specified as an RGB triplet in the form *'#rrggbb'*, where 'rr', 'gg', and 'bb' are hexadecimal numbers between '00' and 'ff', respectively.

Suggested values: *'red'*, *'green'*.

Default: *'white'*.

'alpha': Translucency of the 3D object models. Displaying 3D object models with translucency set to less than 1.0 may significantly increase the runtime of `disp_object_model_3d`.

Value range: [*0.0* (fully transparent) ... *1.0* (fully opaque)].

Default: *1.0*.

'disp_pose': Flag, if the pose of the 3D object models should be visualized.

List of values: *'true'*, *'false'*.

Default: *'false'*.

'disp_lines': Flag, if the contours of the 3D object models' polygons should be displayed.

List of values: *'true'*, *'false'*.

Default: *'false'*.

'disp_normals': Flag, if the surface normals of the 3D object models should be visualized.

List of values: *'true'*, *'false'*.

Default: *'false'*.

'line_color': Color of the lines if *'disp_lines'* is set to *'true'*. The available colors can be queried with the operator `query_color`. In addition, the color may be specified as an RGB triplet in the form *'#rrggbb'*, where 'rr', 'gg', and 'bb' are hexadecimal numbers.

Suggested values: *'red'*, *'green'*.

Default: The value of *'color'*.

'line_width': Sets the width of lines in pixel.

Default: *1.0*.

'normal_color': Color of the visualized normals if *'disp_normals'* is set to *'true'*. The available colors can be queried with the operator `query_color`. In addition, the color may be specified as an RGB triplet in the form *'#rrggbb'*, where 'rr', 'gg', and 'bb' are hexadecimal numbers.

Suggested values: *'red'*, *'green'*.

Default: The value of *'color'*.

'point_size': Sets the diameter of the points in pixel.

Default: *3.5*.

'lut': Sets the LUT that transforms the values of the attribute set with *'color_attrib'* into a color.

See `set_lut` for available LUTs. If *'lut'* is set to anything but *'default'*, *'color'* is ignored.

Default: *'default'*.

'color_attrib': Name of a point attribute that is used for false color visualization.

If an attribute is set, the color of the displayed 3D points is determined by the point's attribute value and the currently set LUT (see *'lut'*). This way, it is possible to visualize attributes in false colors.

Example: If *'color_attrib'* is set to *'coord_z'*, and *'lut'* is set to *'color1'*, the z-coordinates will be color coded from red to blue.

If *'lut'* is set to *'default'*, the attribute values are used to scale the color that was set by the parameter *'color'*.

If *'lut'* is set to a different value, the attribute values of all points are internally scaled to the interval [0,255] and used as input value for the LUT function.

The mapping is also controlled by the parameters *'color_attrib_start'* and *'color_attrib_end'* (see below).

If faces are displayed, their color is interpolated between the color of the corner points.

Suggested values: *'none'*, *'&distance'*, *'coord_x'*, *'coord_y'*, *'coord_z'*, user defined point attributes, or any other point attribute available.

Default: *'none'*.

'color_attrib_start',*'color_attrib_end'*: The range of interest of the values of the attribute set with *'color_attrib'*.

The attribute values between *'color_attrib_start'* and *'color_attrib_end'* are scaled to the start and end of the selected LUT. Attribute values outside the selected range are clipped. This allows to use a fixed color mapping which will not be distorted by outliers.

If set to *'auto'*, the minimum attribute value is mapped to the start of the LUT, the maximum is mapped to the end of the LUT, *except* if *'color_attrib'* is *'normal_x'*, *'normal_y'*, or *'normal_z'*. In this case, start and end are automatically set to *-1* and *1*.

It is possible to enter start value that is higher than the end value. This will in effect flip the used LUT.

Suggested values: *'auto'*, *0*, *0.1*, *1*, *100*, *255*.

Default: *'auto'*.

'red_channel_attrib', *'green_channel_attrib'*, *'blue_channel_attrib'*: Name of a point attribute that is used for the red, green, or blue color channel.

This is most useful when used with a group of three connected attributes, like RGB colors or normal vectors. This way it is possible to display points in colored texture, e.g., display the object model with overlayed RGB-sensor data, or display point normals in false colors.

To display only a single attribute in false colors, please use *'color_attrib'* (see above).

By default, the attribute values are assumed to lie between *0* and *255*. If the attributes have a different range, you additionally have to set the parameters *'rgb_channel_attrib_start'* and *'rgb_channel_attrib_end'* (see below).

If only 1 or 2 channels are set, the remaining channels use the RGB value of the color set with *'color'*.

If faces are displayed, their color is interpolated between the color of the corner points.

Suggested values: *'none'*, *'&red'*, *'&green'*, *'&blue'*, *'normal_x'*, *'normal_y'*, *'normal_z'*, user defined point attributes, or any other point attribute available.

Default: *'none'*.

'rgb_channel_attrib_start', *'rgb_channel_attrib_end'*: The range of interest of the values of attributes set with *'red_channel_attrib'*, *'green_channel_attrib'*, and *'blue_channel_attrib'*.

These parameters define the value range that is scaled to the full RGB channels. This is useful, if the input attribute values are not in the interval [0,255].

If set to *'auto'*, the minimum attribute value is mapped to *0*, the maximum is mapped to *255*, *except* if the attribute is *'normal_x'*, *'normal_y'*, or *'normal_z'*. In this case, start and end are automatically set to *-1* and *1*.

It is possible to enter start value that is higher than the end value. This will in effect invert the displayed RGB colors.

The range can be set for the channels individually by replacing *'rgb'* in the parameter name with the channel name, e.g., *'green_channel_attrib_start'*.

Suggested values: *'auto'*, *0*, *0.1*, *1*, *100*, *255*.

Default: *0*, *255*.

All generic parameters are evaluated from left to right. For example, to set the color of the first 3D object model to red and the color of all other 3D object models to green, set `GenParamName` to *['color','color_0']* and `GenParamValue` to *['green','red']*.

`disp_object_model_3d` requires OpenGL 2.1, GLSL 1.2, and the OpenGL extensions GL_EXT_framebuffer_object and GL_EXT_framebuffer_blit. Otherwise the compatibility mode is automatically enabled. On graphics cards with low memory the following error messages could occur if rendering in a window with high resolution:

Low level error: 'Incomplete attachment'

Unhanded Exception: 'Required framebuffer object is unsupported'

Solutions:

Set the parameter *'quality'* to *'low'* or use the compatibility mode to reduce the requirements of the graphics card.

The system variable (see `set_system`) *'opengl_compatibility_mode_enable'* can be set to *'true'* to permanently enable the visualization in compatibility mode with lower OpenGL requirements. This mode requires OpenGL 1.1. In compatibility mode the parameters *'object_index_persistence'*, *'depth_persistence'* and *'quality'* are not used.

On Linux Remote Desktop *'disp_background'* is not supported.

Attention

Cameras with hypercentric lenses are not supported. For displaying large faces (or primitives) with a non-zero distortion in `CamParam`, note that the distortion is only applied to the points of the model. In the projection, these points are subsequently connected by straight lines. For a good approximation of the distorted lines, please use a triangulation with sufficiently small triangles.

Parameters

- ▷ **`WindowHandle`** (input_control) . . . . . window ⇝ *handle*
  Window handle.
- ▷ **`ObjectModel3D`** (input_control) . . . . . object_model_3d(-array) ⇝ *handle*
  Handles of the 3D object models.
- ▷ **`CamParam`** (input_control) . . . . . campar ⇝ *real* / integer / string
  Camera parameters of the scene.
  Default: []
- ▷ **`Pose`** (input_control) . . . . . pose(-array) ⇝ *real* / integer
  3D poses of the objects.
  Default: []
- ▷ **`GenParamName`** (input_control) . . . . . string-array ⇝ *string*
  Names of the generic parameters.
  Default: []
  List of values: `GenParamName` ∈ {'alpha', 'attribute', 'color', 'colored', 'depth_persistence', 'disp_background', 'disp_lines', 'disp_pose', 'disp_normals', 'light_position', 'light_ambient', 'light_diffuse', 'line_color', 'line_width', 'normal_color', 'object_index_persistence', 'point_size', 'quality', 'compatibility_mode_enable', 'color_attrib', 'color_attrib_start', 'color_attrib_end', 'red_channel_attrib', 'green_channel_attrib', 'blue_channel_attrib', 'rgb_channel_attrib_start', 'rgb_channel_attrib_end', 'lut'}
- ▷ **`GenParamValue`** (input_control) . . . . . string-array ⇝ *string* / integer / real
  Values of the generic parameters.
  Default: []
  List of values: `GenParamValue` ∈ {'true', 'false', 'coord_x', 'coord_y', 'coord_z', 'normal_x', 'normal_y', 'normal_z', 'red', 'green', 'blue', 'auto', 'faces', 'primitive', 'points', 'lines'}

Result

`disp_object_model_3d` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised. If the image rendering exceeds the available memory of the graphics card, the error 5188 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`find_surface_model`, `fit_primitives_object_model_3d`, `read_object_model_3d`, `segment_object_model_3d`

Possible Successors

`get_disp_object_model_3d_info`

See also

`render_object_model_3d`, `project_object_model_3d`, `project_shape_model_3d`, `object_model_3d_to_xyz`

Module

3D Metrology

disp_polygon (: : WindowHandle, Row, Column :)

Displays a polyline.

`disp_polygon` displays a polyline with the row coordinates `Row` and the column coordinates `Column` in the output window. The parameters `Row` and `Column` have to be provided as tuples. Straight lines are drawn between the given points. The start and the end of the polyline are not connected.

The operators used to control the display of regions (e.g., `set_color`, `set_gray`, `set_draw`, `set_line_width`) can also be used with polylines.

Attention

The given coordinates must lie within the window.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row** (input_control) ..... polygon.y-array ⇝ *integer* / real
Row index
Default: [16,80,80]
Suggested values: Row ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Column** (input_control) ..... polygon.x-array ⇝ *integer* / real
Column index
Default: [48,16,80]
Suggested values: Column ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

Example

```
/* display a rectangle */

disp_rectangle1_margin1(Htuple WindowHandle,
                        Hlong Row1, long Column1,
                        Hlong Row2, long Column2)
{
```

```
  Htuple Row, Col;
  create_tuple(&Row,4);
  create_tuple(&Col,4);

  set_i(Row,Row1,0);
  set_i(Col,Column1,0);

  set_i(Row,Row1,1);
  set_i(Col,Column2,1);

  set_i(Row,Row2,2);
  set_i(Col,Column2,2);

  set_i(Row,Row2,3);
  set_i(Col,Column1,3);

  set_i(Row,Row1,4);
  set_i(Col,Column1,4);

  T_disp_polygon(WindowHandle,Row,Col);

}
```

Result

`disp_polygon` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `set_rgb`, `set_lut`, `set_hsi`, `set_draw`, `set_color`, `set_colored`, `set_line_width`

Alternatives

`disp_line`, `gen_region_polygon`, `disp_region`

See also

`open_window`, `set_color`, `set_rgb`, `set_hsi`, `set_insert`, `set_line_width`

Module

Foundation

disp_rectangle1 (: : WindowHandle, Row1, Column1, Row2, Column2 :)

Display of rectangles aligned to the coordinate axes.

`disp_rectangle1` displays one or several rectangles in the output window. A rectangle is described by the upper left corner (`Row1`,`Column1`) and the lower right corner (`Row2`,`Column2`). If the given coordinates are not within the boundary of the window the rectangle is clipped accordingly. The operators used to control the display of regions (e.g., `set_color`, `set_gray`, `set_draw`, `set_line_width`) can also be used with rectangles. Several rectangles can be displayed with one call by using tuple parameters.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row1** (input_control) . . . . . rectangle.origin.y(-array) ⇝ *real* / integer
Row index of the upper left corner.
Default: 16
Suggested values: Row1 ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ Row1 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Column1** (input_control) . . . . . rectangle.origin.x(-array) ⇝ *real* / integer
Column index of the upper left corner.
Default: 16
Suggested values: Column1 ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ Column1 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Row2** (input_control) . . . . . rectangle.corner.y(-array) ⇝ *real* / integer
Row index of the lower right corner.
Default: 48
Suggested values: Row2 ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ Row2 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10
Restriction: Row2 >= Row1

▷ **Column2** (input_control) . . . . . rectangle.corner.x(-array) ⇝ *real* / integer
Column index of the lower right corner.
Default: 80
Suggested values: Column2 ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ Column2 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10
Restriction: Column2 >= Column1

Example

```
set_color(WindowHandle,'green')
draw_region(MyRegion,WindowHandle)
smallest_rectangle1(MyRegion,R1,C1,R2,C2)
disp_rectangle1(WindowHandle,R1,C1,R2,C2)
```

Result

disp_rectangle1 returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_rgb, set_lut, set_hsi, set_draw, set_color, set_colored, set_line_width

Alternatives

disp_rectangle2, gen_rectangle1, disp_region, disp_line, set_shape

See also

open_window, set_color, set_draw, set_line_width

Module

Foundation

disp_rectangle2 (: : WindowHandle, CenterRow, CenterCol, Phi, Length1, Length2 :)

Displays arbitrarily oriented rectangles.

disp_rectangle2 draws one or several arbitrarily oriented rectangles in the output window. A rectangle is described by the center (CenterRow,CenterCol), the orientation Phi (in radians) and half the lengths of the edges Length1 and Length2. The operators used to control the display of regions (e.g., set_draw, set_gray, set_draw) can also be used with rectangles. Several rectangles can be displayed with one call by using tuple parameters. For the use of colors with several rectangles, see set_color.

Attention

The center must lie within the window boundaries.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **CenterRow** (input_control) . . . . . rectangle2.center.y(-array) ⇝ *real* / integer
Row index of the center.
Default: 48
Suggested values: CenterRow ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ CenterRow ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **CenterCol** (input_control) . . . . . rectangle2.center.x(-array) ⇝ *real* / integer
Column index of the center.
Default: 64
Suggested values: CenterCol ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ CenterCol ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Phi** (input_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real* / integer
Orientation of rectangle in radians.
Default: 0.0
Suggested values: Phi ∈ {0.0, 0.785398, 1.570796, 3.1415926, 6.283185}
Value range: 0.0 ≤ Phi ≤ 6.283185 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **Length1** (input_control) . . . . . rectangle2.hwidth(-array) ⇝ *real* / integer
Half of the length of the longer side.
Default: 48
Suggested values: Length1 ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ Length1 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Length2** (input_control) . . . . . rectangle2.hheight(-array) ⇝ *real* / integer
Half of the length of the shorter side.
Default: 32
Suggested values: Length2 ∈ {0, 64, 128, 256, 511}
Value range: 0 ≤ Length2 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10
Restriction: Length2 < Length1

Example

```
set_color(WindowHandle,'green')
draw_region(MyRegion,WindowHandle)
elliptic_axis(MyRegion,Ra,Rb,Phi)
area_center(MyRegion,_,Row,Column)
disp_rectangle2(WindowHandle,Row,Column,Phi,Ra,Rb)
```

Result

disp_rectangle2 returns 2 (H_MSG_TRUE), if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_rgb, set_lut, set_hsi, set_draw, set_color, set_colored, set_line_width

Alternatives

disp_region, gen_rectangle2, disp_rectangle1, set_shape

See also

open_window, disp_region, set_color, set_draw, set_line_width

Module

Foundation

disp_region (DispRegions : : WindowHandle :)

Displays regions in a window.

disp_region displays the regions in DispRegions in the output window. The parameters for output can be set with the operators set_color, set_gray, set_draw, set_line_width, etc.

The color(s) for the display of the regions are determined with set_color, set_rgb, set_gray or set_colored. If more than one region is displayed and more than one color is set, the colors are assigned in a cyclic way to the regions.

The form of the region for output can be modified by set_paint (e.g., encompassing circle, convex hull). The command set_draw determines if the region is filled or only the boundary is drawn. If only the boundary is drawn, the thickness of the boundary will be determined by set_line_width and the style by set_line_style.

Parameters

▷ **DispRegions** (input_object) ..... region(-array) ⇝ *object*
Regions to display.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

Example

```
* Output with 12 colors:
set_colored(WindowHandle,12)
disp_region(SomeSegments,WindowHandle)

* Symbolic representation:
set_draw(WindowHandle,'margin')
set_color(WindowHandle,'red')
set_shape(WindowHandle,'ellipse')
disp_region(SomeSegments,WindowHandle)

* Representation of a margin with pattern:
set_draw(WindowHandle,'margin')
set_color(WindowHandle,'blue')
set_line_style(WindowHandle,[12,3])
disp_region(Segments,WindowHandle)
```

Result

`disp_region` returns 2 (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `set_rgb`, `set_lut`, `set_hsi`, `set_shape`, `set_line_style`, `set_insert`, `set_draw`, `set_color`, `set_colored`, `set_line_width`

Alternatives

`disp_obj`, `disp_arrow`, `disp_line`, `disp_circle`, `disp_rectangle1`, `disp_rectangle2`, `disp_ellipse`

See also

`open_window`, `set_color`, `set_colored`, `set_draw`, `set_shape`, `set_paint`, `set_gray`, `set_rgb`, `set_hsi`, `set_line_width`, `set_line_style`, `set_insert`, `paint_region`, `dump_window`

Module

Foundation

`disp_xld` (XLDObject : : WindowHandle :)

Display an XLD object.

`disp_xld` serves to display an XLD object of arbitrary type.

Parameters

▷ **`XLDObject`** (input_object) ..... xld-array ⇝ *object*
XLD object to display.

▷ **`WindowHandle`** (input_control) ..... window ⇝ *handle*
Window handle.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`disp_image`, `disp_region`, `disp_channel`, `disp_color`, `disp_line`, `disp_arc`

Module

Foundation

13.7 Parameters

`convert_coordinates_image_to_window` (: : WindowHandle, RowImage, ColumnImage : RowWindow, ColumnWindow)

Convert image coordinates to window coordinates

`convert_coordinates_image_to_window` converts image coordinates `RowImage` and `ColumnImage` into window coordinates `RowWindow` and `ColumnWindow` based on the displayed image part and the window size of the window defined in `WindowHandle`.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle

▷ **RowImage** (input_control) ..... coordinates.y(-array) ⇝ *real*
Row in image coordinates.

▷ **ColumnImage** (input_control) ..... coordinates.x(-array) ⇝ *real*
Column in image coordinates.

▷ **RowWindow** (output_control) ..... coordinates.y(-array) ⇝ *real*
Row (Y) in window coordinates.

▷ **ColumnWindow** (output_control) ..... coordinates.x(-array) ⇝ *real*
Column (X) in window coordinates.

Example

```
read_image (Image, 'printer_chip/printer_chip_01')
dev_get_window (WindowHandle)
get_window_extents (WindowHandle, Row, Column, Width, Height)
dev_set_part (450, 300, 750, 600)
dev_display (Image)
*
* Generate rectangle in image coordinates
Row := [474, 746]
Column := [314, 589]
gen_rectangle1 (Rectangle1, Row[0], Column[0], Row[1], Column[1])
* Convert rectangle corner points to window coordinates
convert_coordinates_image_to_window (WindowHandle, Row[[0,1,0,1]], \
    Column[[0,0,1,1]], RowWindow, ColumnWindow)
*
* Window center in window coordinates
WindowCenterRow := Height/2-1
WindowCenterColumn := Width/2-1
* Convert window center to image coordinates
convert_coordinates_window_to_image (WindowHandle, WindowCenterRow, \
    WindowCenterColumn, RowImage, ColumnImage)
*
* Display all points in image coordinates
dev_display (Image)
disp_cross (WindowHandle, Row[[0,1,0,1]], Column[[0,0,1,1]], 6, rad(45))
disp_cross (WindowHandle, RowImage, ColumnImage, 6, 0)
```

Result

`convert_coordinates_image_to_window` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`set_window_extents`, `set_part`

See also

`convert_coordinates_window_to_image`

Module

Foundation

convert_coordinates_window_to_image (: : WindowHandle, RowWindow, ColumnWindow : RowImage, ColumnImage)

Convert window coordinates to image coordinates

convert_coordinates_window_to_image converts window coordinates RowWindow and ColumnWindow into image coordinates RowImage and ColumnImage based on the displayed image part and the window size of the window defined in WindowHandle.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **RowWindow** (input_control) ..... coordinates.y(-array) ⇝ *real*
  Row (Y) in window coordinates.
- ▷ **ColumnWindow** (input_control) ..... coordinates.x(-array) ⇝ *real*
  Column (X) in window coordinates.
- ▷ **RowImage** (output_control) ..... coordinates.y(-array) ⇝ *real*
  Row in image coordinates.
- ▷ **ColumnImage** (output_control) ..... coordinates.x(-array) ⇝ *real*
  Column in image coordinates.

Example

```
read_image (Image, 'printer_chip/printer_chip_01')
dev_get_window (WindowHandle)
get_window_extents (WindowHandle, Row, Column, Width, Height)
dev_set_part (450, 300, 750, 600)
dev_display (Image)
*
* Generate rectangle in image coordinates
Row := [474, 746]
Column := [314, 589]
gen_rectangle1 (Rectangle1, Row[0], Column[0], Row[1], Column[1])
* Convert rectangle corner points to window coordinates
convert_coordinates_image_to_window (WindowHandle, Row[[0,1,0,1]], \
    Column[[0,0,1,1]], RowWindow, ColumnWindow)
*
* Window center in window coordinates
WindowCenterRow := Height/2-1
WindowCenterColumn := Width/2-1
* Convert window center to image coordinates
convert_coordinates_window_to_image (WindowHandle, WindowCenterRow, \
    WindowCenterColumn, RowImage, ColumnImage)
*
* Display all points in image coordinates
dev_display (Image)
disp_cross (WindowHandle, Row[[0,1,0,1]], Column[[0,0,1,1]], 6, rad(45))
disp_cross (WindowHandle, RowImage, ColumnImage, 6, 0)
```

Result

convert_coordinates_window_to_image returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_window_extents, set_part

See also

convert_coordinates_image_to_window

Module

Foundation

get_contour_style (: : WindowHandle : Style)

Get the current contour display fill style.

get_contour_style returns the contour fill style of the output window as set with set_contour_style.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Style** (output_control) . . . . . string ⇝ *string*
Current contour fill style.

Result

get_contour_style returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_contour_style, disp_xld

See also

set_contour_style, disp_xld

Module

Foundation

get_draw (: : WindowHandle : Mode)

Get the current region fill mode.

get_draw returns the region fill mode of the output window. It is used by operators as disp_region, disp_circle, disp_arrow, disp_rectangle1, disp_rectangle2 etc. The region fill mode is set with set_draw.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Mode** (output_control) . . . . . string ⇝ *string*
Current region fill mode.

Result

get_draw returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Successors

set_draw, disp_region

See also

set_draw, disp_region, set_paint

Module

Foundation

get_hsi (: : WindowHandle : Hue, Saturation, Intensity)

Get the HSI coding of the current color.

get_hsi returns the output color or gray values, respectively, for the window, described in Hue, Saturation and Intensity. The values returned by get_hsi can be set with set_hsi.

Attention

The values returned by get_hsi may be inaccurate due to rounding errors. They do not necessarily match the values set with set_hsi exactly (colors are stored in RGB internally).

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Hue** (output_control) ..... integer-array ⇝ *integer*
  Hue (color value) of the current color.
- ▷ **Saturation** (output_control) ..... integer-array ⇝ *integer*
  Saturation of the current color.
- ▷ **Intensity** (output_control) ..... integer-array ⇝ *integer*
  Intensity of the current color.

Result

get_hsi returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_hsi, set_rgb, disp_image

See also

set_hsi, set_color, set_rgb, trans_to_rgb, trans_from_rgb

Module

Foundation

get_icon (: Icon : WindowHandle :)

Query the icon for region output

get_icon queries the icon that was set with set_icon.

Parameters

- ▷ **Icon** (output_object) ..... region ⇝ *object*
  Icon for the regions center of gravity.
- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.

Example

```
/* draw a region and an icon.  */
/* set it and get it again.    */
T_draw_region(&Region,WindowHandle);
T_draw_region(&Icon,WindowHandle);
set_icon(Icon);
create_tuple_s(&icon,"icon");
T_set_shape(WindowHandle,icon);
destroy_tuple(icon);
T_disp_region(Region,WindowHandle);
get_icon(&OldIcon);
T_disp_region(OldIcon,WindowHandle);
```

Result

`get_icon` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`set_icon`

Possible Successors

`disp_region`

Module

Foundation

`get_line_style ( : : WindowHandle : Style )`

Get the current graphic mode for contours.

`get_line_style` returns the display mode for contours when displaying regions. It is used by operators like `disp_region`, `disp_line`, `disp_polygon`, etc. `Style` is set with the operator `set_line_style`. `Style` is only important for displaying the contour of objects, especially if a line style was set with `set_line_style`.

Parameters

▷ **`WindowHandle`** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **`Style`** (output_control) ..... integer-array ⇝ *integer*
Template for contour display.

Result

`get_line_style` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`set_line_style`, `disp_region`

Module

Foundation

get_line_width (: : WindowHandle : Width)

Get the current line width for contour display.

get_line_width returns the line width for region display in the window. It is used by operators like disp_region, disp_line, disp_polygon, etc. Width is set with the operator set_line_width. Width is only important for displaying the contour of objects.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Width** (output_control) ..... real ⇝ *real*
Current line width for contour display.

Result

get_line_width returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_line_width, set_line_style, disp_region

See also

set_line_width, disp_region

Module

Foundation

get_paint (: : WindowHandle : Mode)

Get the current display mode for gray values.

get_paint returns the display mode for gray values in the window. Mode is used by the operator disp_image. get_paint is used for temporary changes of the gray value display mode. The current value is queried, then changed (with the operator set_paint) and finally the old value is written back. The available modes can be viewed with the operator query_paint. Mode is the name of the display mode. If a mode can be customized with parameters, the parameter values are passed in a tuple after the mode name. The order of values is the same as in set_paint.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Mode** (output_control) ..... string-array ⇝ *string* / integer / real
Name and parameter values of the current display mode.

Result

get_paint returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_paint

Possible Successors

set_paint, disp_region, disp_image

See also

set_paint, query_paint, disp_image

Module

Foundation

get_part (: : WindowHandle : Row1, Column1, Row2, Column2)

Get the image part.

get_part returns the upper left and lower right corner of the image part shown in the window. The image part can be changed with the operator set_part (Default is the whole image).

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Row1** (output_control) ..... rectangle.origin.y ⇝ *integer* / real
  Row index of the image part's upper left corner.
- ▷ **Column1** (output_control) ..... rectangle.origin.x ⇝ *integer* / real
  Column index of the image part's upper left corner.
- ▷ **Row2** (output_control) ..... rectangle.corner.y ⇝ *integer* / real
  Row index of the image part's lower right corner.
- ▷ **Column2** (output_control) ..... rectangle.corner.x ⇝ *integer* / real
  Column index of the image part's lower right corner.

Result

get_part returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_part, disp_region, disp_image

See also

set_part, disp_image, disp_region, disp_color

Module

Foundation

get_part_style (: : WindowHandle : Style)

Get the current interpolation mode for gray value display.

get_part_style returns the interpolation mode used for displaying an image part in the window. An interpolation takes place if the output window is larger than the image format or the image output format (see set_part). HALCON supports three interpolation modes:

0 no interpolation (low quality, very fast).

1 unweighted interpolation (average quality and computation time)

2 weighted interpolation (high quality, slow)

The current mode can be changed with set_part_style.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Style** (output_control) . . . . . integer ⇝ *integer*
Interpolation mode for image display: 0 (fast, low quality) to 2 (slow, high quality).
List of values: Style ∈ {0, 1, 2}

Result

get_part_style returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_part_style, disp_region, disp_image

See also

set_part_style, set_part, disp_image, disp_color

Module

Foundation

get_rgb (: : WindowHandle : Red, Green, Blue)

Get the current color in RGB-coding.

get_rgb returns the output colors or gray values, respectively, for the output window. They are defined by the three color components red, green and blue.

The values returned by get_rgb can be set with set_rgb.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Red** (output_control) . . . . . integer-array ⇝ *integer*
The current color's red value.

▷ **Green** (output_control) . . . . . integer-array ⇝ *integer*
The current color's green value.

▷ **Blue** (output_control) . . . . . integer-array ⇝ *integer*
The current color's blue value.

Result

get_rgb returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_rgb, disp_region, disp_image

See also

set_rgb

Module

Foundation

get_rgba (: : WindowHandle : Red, Green, Blue, Alpha)

Get the current color in RGBA-coding.

get_rgba returns the output colors or gray values, respectively, for the output window. They are defined by the four color components red, green, blue and alpha.

The values returned by get_rgba can be set with set_rgba.

Attention

get_rgba depends on the library libcanvas, which might not be available on embedded systems.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Red** (output_control) ..... integer-array ⇝ *integer*
The current color's red value.

▷ **Green** (output_control) ..... integer-array ⇝ *integer*
The current color's green value.

▷ **Blue** (output_control) ..... integer-array ⇝ *integer*
The current color's blue value.

▷ **Alpha** (output_control) ..... integer-array ⇝ *integer*
The current color's alpha value.

Result

get_rgba returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_rgba, disp_obj

See also

set_rgba

Module

Foundation

get_shape (: : WindowHandle : DisplayShape)

Get the current region output shape.

get_shape returns the shape in which regions are displayed. The available shapes can be queried with query_shape and then changed with set_shape.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **DisplayShape** (output_control) ..... string ⇝ *string*
Current region output shape.

Result

get_shape returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_shape

Possible Successors

set_shape, disp_region

See also

set_shape, query_shape, disp_region

Module

Foundation

get_window_param (: : WindowHandle, Param : Value)

Get window parameters.

The operator get_window_param allows querying different parameters of an open window. For detailed descriptions of the individual parameters see set_window_param.

General window parameters:

'flush' *'true'* if automatic flushing is enabled, *'false'* otherwise.

'region_quality' Quality of the rendering of regions.

'background_color' The background color of the window WindowHandle.

'window_title' The text in the title bar of the window WindowHandle.

'anti_aliasing' *'true'* if anti aliasing is enabled, *'false'* otherwise.

'graphics_stack' *'true'* if the graphics stack is enabled, *'false'* otherwise.

'graphics_stack_max_element_num' Maximum number of elements in the graphics stack.

'graphics_stack_max_memory_size' Maximum memory consumption of the graphics stack.

'pixel_grid_enable' *'true'* if the pixel grid is enabled for the window WindowHandle.

'pixel_grid_min_resolution' Minimal required resolution of one pixel in the window WindowHandle such that the pixel grid becomes visible.

'pixel_grid_color' Color of the pixel grid lines.

'pixel_grid_line_width' Line width of the pixel grid lines.

Parameters concerning the *'3d_plot'*:

'angle_of_view' The angle of view of the virtual camera used to display the 3D plot.

'axis_captions' The captions for row, column, and height axis.

'plot_quality' The quality of the 3D plot.

'scale_plot' *'true'* if the height values of an image are transformed into the interval [0,255] before display, *'false'* otherwise.

'display_grid' *'true'* if the grid is displayed.

'display_axes' *'true'* if the coordinate axes are displayed.

Attention

The parameters *'anti_aliasing'*, *'flush'*, *'graphics_stack'*, *'graphics_stack_max_element_num'*, *'graphics_stack_max_memory_size'*, and *'region_quality'* depend on the library libcanvas, which might not be available on embedded systems.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Param** (input_control) . . . . . string ⇝ *string*
Name of the parameter.
Default: 'flush'
List of values: Param ∈ {'plot_quality', 'axis_captions', 'scale_plot', 'angle_of_view', 'display_grid', 'display_axes', 'window_title', 'background_color', 'graphics_stack', 'graphics_stack_max_element_num', 'graphics_stack_max_memory_size', 'pixel_grid_enable', 'pixel_grid_min_resolution', 'pixel_grid_color', 'pixel_grid_line_width', 'flush', 'region_quality'}

▷ **Value** (output_control) . . . . . string(-array) ⇝ *string* / real / integer
Value of the parameter.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

set_window_param

See also

get_window_attr, get_part_style

Module

Foundation

query_all_colors (: : WindowHandle : Colors)

Query all color names.

query_all_colors returns the names of all colors that are known to HALCON. That doesn't mean that these colors are available for specific screens. On some screens there may only be a subset of colors available (see query_color). The HALCON colors are used to display regions (disp_region, disp_polygon, disp_circle, etc.). They can be defined with set_color.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Colors** (output_control) . . . . . string-array ⇝ *string*
Color names.

Result

query_all_colors always returns 2 (H_MSG_TRUE)

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_system, set_color, disp_region

See also

query_color, set_system, set_color, disp_region, open_window

Module

Foundation

query_color (: : WindowHandle : Colors)

Query all color names displayable in the window.

query_color returns the names of all colors that are usable for region output (disp_region, disp_polygon, disp_circle, etc.). On a b/w screen query_color returns *'black'* and *'white'*. These two "colors" are displayable on any screen. In addition to *'black'* and *'white'* several gray values (e.g., *'dim gray'*) are returned on screens capable of gray values. A list of all displayable colors is returned for screens with color lookup table. The returned tuple of colors begins with b/w, followed by the three primaries (*'red'*,*'green'*,*'blue'*) and several gray values. query_all_colors(::WindowHandle:Colors) returns a list of all available colors. For screens with true-color output the same list is returned by query_color. The list of available colors (to HALCON) must not be confused with the list of displayable colors. For screens with true-color output the available colors are only a small subset of the displayable colors. Colors that are not directly available to HALCON can be chosen manually with set_rgb or set_hsi. If colors are chosen that are known to HALCON but cannot be displayed, HALCON can choose a similar color. To use this feature, set_check(::'~color':) must be set.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Colors** (output_control) ..... string-array ⇝ *string*
Color names.

Example

```
open_window(0,0,-1,-1,'root','invisible','',WindowHandle)
query_color(WindowHandle,Colors)
close_window(WindowHandle)
fwrite_string (FileHandle, ['Displayable colors: ',Colors])
```

Result

query_color returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_color, disp_region

See also

query_all_colors, set_color, disp_region, open_window

Module

Foundation

query_colored (: : : PossibleNumberOfColors)

Query the number of colors for color output.

query_colored returns all possible parameter values for set_colored. set_colored defines how many colors are used for region or graphics output.

Parameters

▷ **PossibleNumberOfColors** (output_control) ..... integer-array ⇝ *integer*
Tuple of the possible numbers of colors.

Example

```
regiongrowing(Image,Regions,5,5,6,100)
query_colored(Colors)
dev_set_colored(Colors[1])
dev_display (Regions)
```

Result

`query_colored` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`set_colored`, `set_color`, `disp_region`

Alternatives

`query_color`

See also

`set_colored`, `set_color`

Module

Foundation

query_gray (: : WindowHandle : Grayval)

Query the displayable gray values.

`query_gray` returns all gray values that are used for gray value output (`disp_image`) and that can be reproduced exactly in the window. They can be set with the `set_gray` call.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Grayval** (output_control) ..... integer-array ⇝ *integer*
Tuple of all displayable gray values.

Result

`query_gray` returns `2` (`H_MSG_TRUE`), if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`set_gray`, `disp_region`

See also

`set_gray`, `disp_image`

Module

Foundation

query_line_width (: : : Min, Max)

Query the possible line widths.

query_line_width returns the minimal (Min) and maximal (Max) values of widths of region border which can be displayed. Setting of the border width is done with set_line_width. Border width is used by operators like disp_region, disp_line, disp_circle, disp_rectangle1, disp_rectangle2 etc. if the drawing mode is "margin" (set_draw(::WindowHandle,'margin':)).

Parameters

▷ **Min** (output_control) ..... integer ⇝ *integer*
Displayable minimum width.

▷ **Max** (output_control) ..... integer ⇝ *integer*
Displayable maximum width.

Result

query_line_width returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_line_width, set_line_width, set_line_style, disp_line

See also

disp_circle, disp_line, disp_rectangle1, disp_rectangle2, disp_region, set_line_width, get_line_width, set_line_style

Module

Foundation

query_paint (: : WindowHandle : Mode)

Query the gray value display modes.

query_paint returns the names of all gray value display modes (e.g., 'gray', '3d_plot', etc.) for the output window. These modes are used by set_paint. query_paint only returns the names of the display values, not the additional parameters that may be necessary for some modes.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Mode** (output_control) ..... string-array ⇝ *string*
Gray value display mode names.

Result

query_paint returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_paint, set_paint, disp_image

See also

set_paint, get_paint, disp_image

Module

Foundation

query_shape (: : : DisplayShape)

Query the region display modes.

query_shape returns the names of all region display modes (e.g., 'original', 'circle', 'rectangle1', 'rectangle2', 'ellipse', etc.) for the window. They are used by set_shape.

Parameters

▷ **DisplayShape** (output_control) ..... string-array ⇝ *string*
region display mode names.

Result

query_shape returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_shape, set_shape, disp_region

See also

set_shape, get_shape, disp_region

Module

Foundation

set_color (: : WindowHandle, Color :)

Set output color.

set_color defines the colors for region output in the window. The available colors can be queried with the operator query_color. In addition, the Color may be specified as hexadecimal RGB triplet or RGBA quadruplet in the form *'#rrggbb'* and *'#rrggbbaa'*. 'rr', 'gg', 'bb', and 'aa' are hexadecimal numbers between '00' and 'ff', respectively. 'aa' denotes the alpha value of a color and can be used to display transparent regions.

The parameter Color can contain a single color or a tuple with up to 256 colors. If only a single color is passed, all output is in this color. If a tuple of colors is passed, the output color of regions is modulo to the number of colors. In the example below, the first circle is displayed red, the second in transparent green and the third in red again. HALCON always begins output with the first color passed. Note, that the number of output colors depends on the number of objects that are displayed in one operator call. If only single objects are displayed, they always appear in the first color, even if they consist of more than one connected component.

The defined colors are used until set_color, set_rgb, set_rgba, set_hsi or set_gray is called again.

Colors are defined separately for each window. They can only be changed for the valid window.

Color is used in operators with region output like disp_region, disp_line, disp_rectangle1, or disp_arrow.

Color name	75% alpha	50% alpha	25% alpha
'black'	'#000000c0'	'#00000080'	'#00000040'
'white'	'#ffffffc0'	'#ffffff80'	'#ffffff40'
'red'	'#ff0000c0'	'#ff000080'	'#ff000040'
'green'	'#00ff00c0'	'#00ff0080'	'#00ff0040'
'blue'	'#0000ffc0'	'#0000ff80'	'#0000ff40'
'dim gray'	'#696969c0'	'#69696980'	'#69696940'
'gray'	'#bebebec0'	'#bebebe80'	'#bebebe40'
'light gray'	'#d3d3d3c0'	'#d3d3d380'	'#d3d3d340'
'cyan'	'#00ffffc0'	'#00ffff80'	'#00ffff40'
'magenta'	'#ff00ffc0'	'#ff00ff80'	'#ff00ff40'
'yellow'	'#ffff00c0'	'#ffff0080'	'#ffff0040'
'medium slate blue'	'#7b68eec0'	'#7b68ee80'	'#7b68ee40'
'coral'	'#ff7f50c0'	'#ff7f5080'	'#ff7f5040'
'slate blue'	'#6a5acdc0'	'#6a5acd80'	'#6a5acd40'
'spring green'	'#00ff7fc0'	'#00ff7f80'	'#00ff7f40'
'orange red'	'#ff4500c0'	'#ff450080'	'#ff450040'
'dark olive green'	'#556b2fc0'	'#556b2f80'	'#556b2f40'
'pink'	'#ffc0cbc0'	'#ffc0cb80'	'#ffc0cb40'
'cadet blue'	'#5f9ea0c0'	'#5f9ea080'	'#5f9ea040'
'goldenrod'	'#daa520c0'	'#daa52080'	'#daa52040'
'orange'	'#ffa500c0'	'#ffa50080'	'#ffa50040'
'gold'	'#ffd700c0'	'#ffd70080'	'#ffd70040'
'forest green'	'#228b22c0'	'#228b2280'	'#228b2240'
'cornflower blue'	'#6495edc0'	'#6495ed80'	'#6495ed40'
'navy'	'#000080c0'	'#00008080'	'#00008040'
'turquoise'	'#40e0d0c0'	'#40e0d080'	'#40e0d040'
'dark slate blue'	'#483d8bc0'	'#483d8b80'	'#483d8b40'
'light blue'	'#add8e6c0'	'#add8e680'	'#add8e640'
'indian red'	'#cd5c5cc0'	'#cd5c5c80'	'#cd5c5c40'
'violet red'	'#d02090c0'	'#d0209080'	'#d0209040'
'light steel blue'	'#b0c4dec0'	'#b0c4de80'	'#b0c4de40'
'medium blue'	'#0000cdc0'	'#0000cd80'	'#0000cd40'
'khaki'	'#f0e68cc0'	'#f0e68c80'	'#f0e68c40'
'violet'	'#ee82eec0'	'#ee82ee80'	'#ee82ee40'
'firebrick'	'#b22222c0'	'#b2222280'	'#b2222240'
'midnight blue'	'#191970c0'	'#19197080'	'#19197040'

Examples of possible color strings

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Color** (input_control) . . . . . string(-array) ⇝ *string*
Output color names.
Default: 'white'
Suggested values: Color ∈ {'black', 'white', 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'dim gray', 'gray', 'light gray', 'medium slate blue', 'coral', 'slate blue', 'spring green', 'orange red', 'orange', 'dark olive green', 'pink', 'cadet blue', '#003075', '#e53019', '#ffb529', '#f28d26bb'}

Example

```
set_color(WindowHandle, ['red', '#00ff00a0'])
disp_circle(WindowHandle, [100,200,300], [200,300,100], [100,100,100])
```

Result

set_color returns 2 (H_MSG_TRUE) if the window is valid and the passed colors are displayable on the screen. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_color

Possible Successors

disp_region

Alternatives

set_rgb, set_hsi

See also

get_rgb, disp_region, set_paint

Module

Foundation

set_colored (: : WindowHandle, NumberOfColors :)

Set multiple output colors.

set_colored is a shortcut for certain set_color calls. It allows the user to display a region set in different colors. NumberOfColors defines the number of colors that are used. Valid values for NumberOfColors can be queried with query_colored.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **NumberOfColors** (input_control) . . . . . integer ⇝ *integer*
Number of output colors.
Default: 12
List of values: NumberOfColors ∈ {3, 6, 12}

Result

set_colored returns 2 (H_MSG_TRUE) if NumberOfColors is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_colored, set_color

Possible Successors

disp_region

See also

query_colored, set_color, disp_region

Module

Foundation

set_contour_style (: : WindowHandle, Style :)

Define the contour display fill style.

set_contour_style defines the fill style of contour displays. The following values are supported for Style:

- *'stroke'*: Only the line of the contour gets displayed.
- *'fill'*: The area enclosed by the contour is filled.
- *'stroke_and_fill'*: The line of the contour gets displayed and the enclosed area filled. especially makes a difference for larger line widths, see set_line_width.

For all styles the current drawing color is used.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Style** (input_control) ..... string ⇝ *string*
Fill style of contour displays.
Default: 'stroke'
List of values: Style ∈ {'stroke', 'fill', 'stroke_and_fill'}

Result

set_contour_style returns 2 (H_MSG_TRUE) if Style is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_contour_style

Possible Successors

disp_xld, disp_obj

See also

get_contour_style, disp_xld, set_line_width, set_line_style

Module

Foundation

set_draw (: : WindowHandle, Mode :)

Define the region fill mode.

set_draw defines the region fill mode. If Mode is set to 'fill', output regions are filled, if set to 'margin', only contours are displayed. Setting Mode only affects the valid window. It is used by operators with region output like disp_region, disp_circle, disp_rectangle1, disp_rectangle2, disp_arrow etc. It is also used by operators with gray value output for some gray value output modes. If the mode is 'margin', the contour can be affected with set_line_width and set_line_style.

Attention

If the output mode is 'margin' and the line width is more than one, objects may not be displayed.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Mode** (input_control) ..... string ⇝ *string*
Fill mode for region output.
Default: 'fill'
List of values: Mode ∈ {'fill', 'margin'}

Result

set_draw returns 2 (H_MSG_TRUE) if Mode is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_draw

Possible Successors

disp_region

See also

get_draw, disp_region, set_paint, disp_image, set_line_width, set_line_style

Module

Foundation

set_gray (: : WindowHandle, GrayValues :)

Define gray values for region output.

set_gray defines the gray values for region output. Gray values are defined as the range of the color lookup table that is used for gray value output with disp_image in conjunction with set_paint(::WindowHandle,'gray':). These entries can be modified by set_lut. So a 'gray value' is the color in which a pixel with the same value is displayed (not necessarily really gray). In general, when changing the color lookup table with set_lut, the colors of the displayed image will change too.

If a gray value is needed as a color for image output (i.e. no color changes with set_lut are possible), it can be set with set_color(::WindowHandle,'gray':).

If only a single gray value is passed, all output will take place in that gray value. If a tuple of gray values is passed, all output will take place in gray values modulo the number of tuple elements. In the example below, the first circle is displayed with gray value 100, the second with 200 and the third with 100 again. Every output operator starts with the first gray value. Note, that the number of output gray values depends on the number of objects that are displayed in one operator call. If only single objects are displayed, they always appear in the first gray value, even if the consist of more than one connected components.

When the operators set_gray, set_color, set_rgb, set_hsi are called, the overwrite the existing values. If not all gray values are displayable on the output device, the number range of GrayValues (0..255) is dithered to the range of displayable gray values. In any case 0 is displayed as black and 255 as white. The displayable gray values can be queried with the operator query_gray. With set_check('~color') error messages can be suppressed if a gray value can't be displayed on the screen. In that case, a similar gray value is displayed.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **GrayValues** (input_control) ..... integer(-array) ⇝ *integer*
Gray values for region output.
Default: 255
Suggested values: GrayValues ∈ {0, 1, 2, 10, 16, 32, 64, 100, 120, 128, 250, 251, 252, 253, 254, 255}
Value range: 0 ≤ GrayValues ≤ 255

Example

```
set_gray(WindowHandle,[100,200])
disp_circle(WindowHandle,[100,200,300],[200,300,100],[100,100,100])
```

Result

set_hsi returns 2 (H_MSG_TRUE) if the window is valid and the given gray value is displayable. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

disp_region

See also

set_color

Module

Foundation

set_hsi (: : WindowHandle, Hue, Saturation, Intensity :)

Define output colors (HSI-coded).

set_hsi sets the region output color(s)/gray value(s) for the valid window. Colors are passed as Hue, Saturation, and Intensity. Transformation from HSI to RGB is done with:

$$
\begin{aligned}
H &= (\texttt{Hue}/255.0) * 2\pi \\
S &= (\texttt{Saturation}/255.0) * 2.0/\sqrt{6} \\
I &= \texttt{Intensity} \\
\\
M1 &= S\cos(H)/\sqrt{6} * 255.0 \\
M2 &= S\sin(H)/\sqrt{2} * 255.0 \\
\\
Red &= 2M1 + I \\
Green &= -M1 + M2 + I \\
Blue &= -M1 - M2 + I
\end{aligned}
$$

The value range is clipped to 0 to 255.

If only one combination is passed, all output will take place in that color. If a tuple of colors is passed, the output color of regions and geometric objects is modulo to the number of colors. HALCON always begins output with the first color passed. Note, that the number of output colors depends on the number of objects that are displayed in one operator call. If only single objects are displayed, they always appear in the first color, even if the consist of more than one connected components.

Selected colors are used until the next call of set_color,set_rgb or set_gray. Colors are relevant to windows, i.e., only the colors of the valid window can be set. Region output colors are used by operators like disp_region, disp_line, disp_rectangle1, disp_rectangle2, disp_arrow, etc. It is also used by operators with gray value output in certain output modes (e.g., *'3d_plot'*, see set_paint).

Attention

The colors are internally stored as RGB triples. Some HSI triples can not be represented by a valid RGB triple (i.e. in the range 0..255). In this case the nearest color which can be represented is used instead.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Hue** (input_control) ..... integer(-array) ⇝ *integer*
  Hue for region output.
  Default: 30
  Value range: $0 \leq$ Hue ≤ 255
- ▷ **Saturation** (input_control) ..... integer(-array) ⇝ *integer*
  Saturation for region output.
  Default: 255
  Value range: $0 \leq$ Saturation ≤ 255
- ▷ **Intensity** (input_control) ..... integer(-array) ⇝ *integer*
  Intensity for region output.
  Default: 84
  Value range: $0 \leq$ Intensity ≤ 255

Result

set_hsi returns 2 (H_MSG_TRUE) if the window is valid and the output colors are displayable. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_hsi

Possible Successors

disp_region

See also

get_hsi, trans_from_rgb, trans_to_rgb, disp_region

Module

Foundation

set_icon (Icon : : WindowHandle :)

Icon definition for region output.

set_icon defines an icon for region output (disp_region). It is displayed in the regions center of gravity. The use of this icon is activated with set_shape.

Parameters

▷ **Icon** (input_object) . . . . . region ⇝ *object*
Icon for center of gravity.

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

Example

```
/* draw a region and an icon */
T_draw_region(&Region,WindowHandle);
T_draw_region(&Icon,WindowHandle);
set_icon(Icon);
create_tuple_s(&icon,"icon");
T_set_shape(WindowHandle,icon);
destroy_tuple(icon);
T_disp_region(Region,WindowHandle);
```

Result

`set_icon` returns `2` (`H_MSG_TRUE`), if exactly one region is supplied. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`

Possible Successors

`set_shape`, `disp_region`

Module

Foundation

set_line_style (: : WindowHandle, Style :)

Define a contour output pattern.

`set_line_style` defines the output pattern of the margin of regions and of XLD contours. The information is used by operators like `disp_region`, `disp_line`, `disp_polygon` etc. The current value can be queried with `get_line_style`. `Style` contains up to five pairs of values. The first value is the length of the visible contour part, the second is the length of the invisible part. The value pairs are used cyclical for contour output.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Style** (input_control) . . . . . integer-array ⇝ *integer*
Contour pattern.
Default: []
Value range: $1 \leq$ `Style` ≤ 120

Example

```
* stroke line: X-Windows
set_line_style(WindowHandle,[20,7])
* point-stroke line: X-Windows
set_line_style(WindowHandle,[20,7,3,7])
* passing line (standard)
set_line_style(WindowHandle,[])
```

Result

`set_line_style` returns `2` (`H_MSG_TRUE`) if the parameter is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_line_style`

Possible Successors

`disp_region`

See also

`get_line_style`, `set_line_approx`, `disp_region`

Module

Foundation

`set_line_width ( : : WindowHandle, Width : )`

Define the line width for region contour output.

`set_line_width` defines the line width (in pixel) in which a region contour is displayed (e.g., with `disp_region`, `disp_line`, `disp_polygon`, etc.) The operator `get_line_width` returns the current value for the window. Some output devices may not allow to change the contour width. If it is possible for the current device, it can be queried with `query_line_width`.

Attention

The line width is important if the output mode was set to 'margin' (see `set_draw`). If the line width is greater than one, regions may not always be displayed correctly.

Parameters

▷ **`WindowHandle`** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **`Width`** (input_control) ..... real ⇝ *real*
Line width for region output in contour mode.
Default: 1.0
Restriction: Width >= 1.0 && Width <= 2000.0

Result

`set_line_width` returns `2` (`H_MSG_TRUE`) if the parameter is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`query_line_width`, `get_line_width`

Possible Successors

`disp_region`

See also

`get_line_width`, `query_line_width`, `set_draw`, `disp_region`

Module

Foundation

set_paint (: : WindowHandle, Mode :)

Define the gray value output mode.

set_paint defines the output mode for gray value display (single- or multichannel) in the window. The mode is used by disp_obj, disp_image, and disp_color.

Different image types and their possible implication on the Mode:

- Gray images can also be interpreted as 3D data, depending on the gray value. To view these 3D plots, select the modes *'3d_plot'*.
- Three-channel images are interpreted as RGB images.
- Vector field images can be viewed as *'vector_field'*.

In most cases the setting Mode *'default'* is the most suitable.

Depending on the selected mode, the Mode may accept a tuple as value. In these cases the entries can be passed the following ways:

- Only the name of the mode is passed: The defaults or the most recently used values are used, respectively.
  Example: `set_paint(WindowHandle,'3d_plot')`
- All values are passed: All output characteristics can be set.
  Example: `set_paint(WindowHandle, ['3d_plot', 'shaded', 8, 0.939, -0.052, 0.296, -0.165, 0.401, 0.072, -0.047, -0.073])`
- Only the first n values are passed: Only the passed values are changed.
  Example: `set_paint(WindowHandle,['3d_plot','texture'])`
- Some of the values are replaced by an asterisk ('*'): The value of the replaced parameters is not changed.
  Example: `set_paint(WindowHandle,['3d_plot', '*' ,8])`

If the current mode is *'default'*, HALCON chooses a suitable algorithm for the output of 2- and 3-channel images. No set_paint call is necessary in this case.

Apart from set_paint there are other operators that affect the output of gray values. The most important of them are set_part, set_part_style and set_lut. Some output modes display gray values using region output (e.g., *'3d_plot'*). In these modes, parameters set with set_color, set_rgb, set_hsi, set_shape, set_line_width and set_insert influence gray value output. In case of unexpected results, check the values of the display parameters.

All available painting modes can be queried with query_paint. Possible values for Mode:

'default' Optimal display on given hardware (default value). Its behavior depends on the number of channels of the input image:

- One-channel image: Output the single channel.
- Two-channel images: Output the first channel.
- Three-channel images: Output as RGB image.

['3d_plot', Mode, Step, qa, qb, qc, qd, ZoomFac, CenterRow, CenterCol, CenterHeight] Gray values are interpreted as a 3D height field and displayed using OpenGL. Thereby the different parameters are:

- *'Mode'*: Specifies the way in which the height field is displayed. The following modes are supported:
  - *'texture'*: The height field is displayed as a closed surface and colored using texture mapping. The texture is passed as the second channel (for a gray-valued texture) or as the second to fourth channel (for a colored texture) of the image to display.
  - *'shaded'*: The height field is displayed as a closed surface and colored using the current LUT.
  - *'hidden_lines'*: The height field is displayed as a hidden line plot and colored using the current LUT.
  - *'contour_lines'*: Height lines are extracted and displayed at their actual height. Again the lines are colored using the current LUT.
- *'Step'*: Step width, i.e., at what intervals samples in row and column direction are taken from the image. The smaller this parameter is chosen, the finer the resulting height field. However, computation time increases while this parameter decreases. In the case of *'contour_lines'* this parameter corresponds to the number of intervals the height range is divided into.

- *'qa'*,*'qb'*,*'qc'*,*'qd'*: The four values of a unit quaternion, describing the orientation of the height field (see `axis_angle_to_quat` for more information on quaternions).
- *'ZoomFac'*: Zooming factor, whereby smaller values implicate higher zooming.
- *'CenterRow'*, *'CenterCol'*, *'CenterHeight'*: Position of the actual center of view defined by row, column, and height. These values must be scaled to the interval [*-0.5*, *0.5*].

Automatic parameter determination: You can set *'qa'* (fourth value in the tuple) or *'ZoomFac'* (eighth value in the tuple) to *'auto'*. Then, the zooming factor and the center of view are determined automatically such that the plot of the height field fills the window as good as possible. By setting *'qa'* to *'auto'*, the previous orientation (or the default one) is kept. By setting *'ZoomFac'* to *'auto'*, the provided orientation is used. It is important that the part of the window is set correctly (using `set_part`) before `set_paint` is called. It is not possible to set any parameters after *'auto'* in the tuple. Furthermore *'auto'* will only work correctly if *'scale_plot'* is set to *'true'* using `set_window_param`.

Example: `set_paint(WindowHandle, ['3d_plot','shaded',4,'auto'])`

This mode allows interactive display of the 3D data. For an interactive rate, your graphics card needs to be powerful enough and support OpenGL. See the operator `update_window_pose` for an intuitive way of modifying the parameters of the 3D plot (e.g., with the mouse).

The colors of the axis are influenced by the colors set using `set_rgb`, `set_color`, and `set_colored`. The first three colors set this way are used as colors for row, column, and height axis (in this order).

Additional parameters concerning the 3D plot can be set using the operator `set_window_param`.

If you intend to use *'3d_plot'* on a Unix-like system, make sure you have sufficient permissions on your graphics device.

Default: *['3d_plot', 'shaded', 8, 0.939, -0.052, 0.296, -0.165, 0.401, 0.072, -0.047, -0.073]*

Restriction: Only for one-channel images.

['vector_field', Mode, Step, MinLength, ScaleLength, CircleSize] Output a vector field. Thus, for `Mode` *'vector_field'* a circle is drawn for each vector at the position of the pixel. Furthermore, a line segment is drawn with the current vector. Thereby the different parameters are:

- *'Mode'*: Tells whether the visualized vector fields contain absolute or relative coordinates. Possible values:
  - *'auto'* (*default*): The visualization depends on the semantic type of the vector field.
  - *'absolute'*: Display vector fields containing absolute coordinates.
  - *'relative'*: Display vector fields containing relative coordinates.
- *'Step'*: Step size for drawing the vectors, i.e., the distance between the drawn vectors.
- *'MinLength'*: Minimum length a vector needs in order to be displayed.
- *'ScaleLength'*: Scales the vector length.
- *'CircleSize'*: Diameter of the drawn circles.

It should be noted that by setting *'vector_field'* only the internal parameters *'Mode'*, *'Step'*, *'MinLength'*, *'ScaleLength'*, and *'CircleSize'* are changed. The current display mode is not changed.

Vector field images are always displayed as vector fields, no matter which `Mode` is selected with `set_paint`.

Example: `set_paint(WindowHandle,['vector_field','auto',16,2,3,5])`

This results in an output of every 16th vector, that is longer than 2 pixels. Each vector is multiplied by 3 for output.

Restriction: Only for vector field images.

Parameters

▷ **`WindowHandle`** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **`Mode`** (input_control) ..... string-array ⇝ *string* / integer / real
Output mode. Additional parameters possible.
Default: 'default'
List of values: `Mode` ∈ {'default', '3d_plot', 'vector_field'}

Result

`set_paint` returns `2` (`H_MSG_TRUE`) if the parameter is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_paint, get_paint

Possible Successors

disp_image

See also

get_paint, query_paint, disp_image, set_shape, set_rgb, set_color, set_gray, set_window_param, update_window_pose

Module

Foundation

set_part (: : WindowHandle, Row1, Column1, Row2, Column2 :)

Modify the displayed image part.

set_part modifies the image part that is displayed in the window. (Row1,Column1) denotes the upper left corner and (Row2,Column2) the lower right corner of the image part to display. The changed values are used by gray value output operators (disp_image, disp_color) as well as region output operators (disp_region).

If only part of an image is displayed, it will be zoomed to full window size. The zooming interpolation method can be set with set_part_style. get_part returns the values of the image part to display. Adapting the size of the window to the size of the image part to be displayed will prevent slowing down the display due to necessary interpolations. Thus, the window preferably has the same size as the image part to be displayed, or otherwise half its size, quarter its size, etc.

Beside setting the image part directly, the following special modes are supported:

Row1 = Column1 = Row2 = Column2 = *-1*: The window size is chosen as the image part, i.e. no zooming of the image will be performed.

Row1, Column1 > *-1* and Row2 = Column2 = *-1*: The size of the last displayed image (in this window) is chosen as the image part, i.e. the image can completely be displayed in the image. For this the image will be zoomed if necessary.

Row1 = Column1 = *0* and Row2 = Column2 = *-2*: The size of the last displayed image (in this window) is used to adapt the image part such that the image fits completely into the window, preserving the aspect ratio of the image. For this the image will be zoomed if necessary.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row1** (input_control) ..... rectangle.origin.y ⇝ *integer* / real
Row of the upper left corner of the chosen image part.
Default: 0

▷ **Column1** (input_control) ..... rectangle.origin.x ⇝ *integer* / real
Column of the upper left corner of the chosen image part.
Default: 0

▷ **Row2** (input_control) ..... rectangle.corner.y ⇝ *integer* / real
Row of the lower right corner of the chosen image part.
Default: -1
Restriction: Row2 >= Row1 || Row2 == -1 || Row1 == 0 && Row2 == -2

▷ **Column2** (input_control) ..... rectangle.corner.x ⇝ *integer* / real
Column of the lower right corner of the chosen image part.
Default: -1
Restriction: Column2 >= Column1 || Column2 == -1 || Column1 == 0 && Column2 == -2

Example

```
get_system('width',Width)
get_system('height',Height)
set_part(WindowHandle,0,0,Height-1,Width-1)
disp_image(Image,WindowHandle)
draw_rectangle1(WindowHandle,Row1,Column1,Row2,Column2)
set_part(WindowHandle,Row1,Column1,Row2,Column2)
disp_image(Image,WindowHandle)
```

Result

`set_part` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_part`

Possible Successors

`set_part_style`, `disp_image`, `disp_region`

Alternatives

`affine_trans_image`

See also

`get_part`, `set_part_style`, `disp_region`, `disp_image`, `disp_color`

Module

Foundation

set_part_style (: : WindowHandle, Style :)

Define an interpolation method for gray value output.

`set_part_style` defines the interpolation method to zoom an image part which is displayed in the window. Interpolation takes place, if the output window has different size than the image to display (e.g., after a call to `set_part` or a window resize). Three modes are supported:

0 no interpolation (low quality, very fast).

1 unweighted interpolation (medium quality and run time)

2 weighted interpolation (high quality, slow)

The current value can be queried with `get_part_style`.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Style** (input_control) . . . . . integer ⇝ *integer*
Interpolation method for image output: 0 (fast, low quality) to 2 (slow, high quality).
Default: 0
List of values: `Style` ∈ {0, 1, 2}

Result

`set_part_style` returns `2` (`H_MSG_TRUE`) if the parameter is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_part_style

Possible Successors

set_part, disp_image, disp_region

Alternatives

affine_trans_image

See also

get_part_style, set_part, disp_image, disp_color

Module

Foundation

set_rgb (: : WindowHandle, Red, Green, Blue :)

Set the color definition via RGB values.

set_rgb sets the output color(s) or the gray values, respectively, for region output for the window. The colors are defined with the red, green and blue components. If only one combination is passed, all output takes place in that color. If a tuple is passed, region output and output of geometric objects takes place modulo the passed colors.

For every call of an output operator, output is started with the first color. If only one object is displayed per call, it will always be displayed in the first color. This is even true for objects with multiple connection components. If multiple objects are displayed per operator call, multiple colors are used. The defined colors are used until set_color, set_rgb or set_gray is called again. The values are used by operators like disp_region, disp_line, disp_rectangle1, disp_rectangle2, disp_arrow, etc.

Attention

If a passed color is not available, an exception is raised. If set_check('~color') was called before, HALCON uses a similar color and suppresses the error.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Red** (input_control) ..... integer(-array) ⇝ *integer*
Red component of the color.
Default: 255
Value range: $0 \leq$ Red ≤ 255

▷ **Green** (input_control) ..... integer(-array) ⇝ *integer*
Green component of the color.
Default: 0
Value range: $0 \leq$ Green ≤ 255

▷ **Blue** (input_control) ..... integer(-array) ⇝ *integer*
Blue component of the color.
Default: 0
Value range: $0 \leq$ Blue ≤ 255

Result

set_rgb returns 2 (H_MSG_TRUE) if the window is valid and all passed colors are available and displayable. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Successors

disp_image, disp_region

Alternatives

set_hsi, set_color, set_gray

See also

disp_region

Module

Foundation

set_rgba (: : WindowHandle, Red, Green, Blue, Alpha :)

Set the color definition via RGBA values.

set_rgba sets the output color(s) or the gray values, respectively, for region output for the window. The colors are defined with the red, green, blue, and alpha components. If only one combination is passed, all output takes place in that color. If a tuple is passed, region output and output of geometric objects takes place modulo the passed colors.

For every call of an output operator, output is started with the first color. If only one object is displayed per call, it will always be displayed in the first color. This is even true for objects with multiple connection components. If multiple objects are displayed per operator call, multiple colors are used. The defined colors are used until set_color, set_rgba, or set_rgb is called again. The values are used by operators like disp_region, disp_line, disp_rectangle1, disp_rectangle2, disp_arrow, etc.

Attention

set_rgba depends on the library libcanvas, which might not be available on embedded systems.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Red** (input_control) ..... integer(-array) ⇝ *integer*
Red component of the color.
Default: 255
Value range: 0 ≤ Red ≤ 255
Restriction: 0 <= Red && Red <= 255

▷ **Green** (input_control) ..... integer(-array) ⇝ *integer*
Green component of the color.
Default: 0
Value range: 0 ≤ Green ≤ 255
Restriction: 0 <= Green && Green <= 255

▷ **Blue** (input_control) ..... integer(-array) ⇝ *integer*
Blue component of the color.
Default: 0
Value range: 0 ≤ Blue ≤ 255
Restriction: 0 <= Blue && Blue <= 255

▷ **Alpha** (input_control) ..... integer(-array) ⇝ *integer*
Alpha component of the color.
Default: 255
Value range: 0 ≤ Alpha ≤ 255
Restriction: 0 <= Alpha && Alpha <= 255

Result

set_rgba returns 2 (H_MSG_TRUE) if the window is valid and all passed colors are available and displayable. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

disp_obj

Alternatives

set_rgb, set_color

Module

Foundation

set_shape (: : WindowHandle, Shape :)

Define the region output shape.

set_shape defines the shape for region output. It is only valid for the window with the logical window number WindowHandle. The output shape is used by disp_region. The available shapes can be queried with query_shape.

Available modes:

'original': The shape is displayed unchanged. Nevertheless modifications via parameters like set_line_width can take place. This is also true for all other modes.

'outer_circle': Each region is displayed by the smallest surrounding circle. (See smallest_circle.)

'inner_circle': Each region is displayed by the largest included circle. (See inner_circle.)

'ellipse': Each region is displayed by an ellipse with the same moments and orientation (See elliptic_axis.)

'rectangle1': Each region is displayed by the smallest surrounding rectangle parallel to the coordinate axes. (See smallest_rectangle1.)

'rectangle2': Each region is displayed by the smallest surrounding rectangle. (See smallest_rectangle2.)

'convex': Each region is displayed by its convex hull (See convexity.)

'icon': Each region is displayed by the icon set with set_icon in the center of gravity.

Attention

Caution is advised for gray value output operators with output parameter settings that use region output.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Shape** (input_control) ..... string ⇝ *string*
Region output mode.
Default: 'original'
List of values: Shape ∈ {'original', 'convex', 'outer_circle', 'inner_circle', 'rectangle1', 'rectangle2', 'ellipse', 'icon'}

Example

```
read_image(Image,'fabrik')
regiongrowing(Image,Seg,5,5,6,100)
set_colored(WindowHandle,12)
set_shape(WindowHandle,'rectangle2')
disp_region(Seg,WindowHandle)
```

Result

set_shape returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_icon, query_shape, get_shape

Possible Successors

disp_region

See also

get_shape, query_shape, disp_region

Module

Foundation

set_window_param (: : WindowHandle, Param, Value :)

Set window parameters.

The operator set_window_param allows setting different parameters of an open window.

General window parameters:

'flush' Enables or disables flushing the contents of the window after the display of every object. If *'flush'* is set to *'false'*, flush_buffer must be called to update the contents of the window. If you want to interact with the content of the graphics window (e.g., zoom or move the content), you have to set *'flush'* to *'true'*.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'region_quality' Influences the quality of the rendering of region objects. Especially if the region to display has to be scaled down for visualization, *'good'* produces a more accurate and pleasing impression. *'low'* is a bit inaccurate in these cases but significantly faster. If only the border of the region is displayed (see set_draw), the fractional part of the line width (see set_line_width) is ignored.

List of values: *'low'*, *'good'*.

Default: *'low'*.

'background_color' Sets the background color of the window WindowHandle. The operator clear_window uses this color to clear the window.

Suggested values: a string containing the color name (e.g., *'black'*, *'red'*).

'window_title' Allows changing the text in the title bar of an already opened HALCON window.

Note that you cannot set *'window_title'* for buffer windows. This restriction also applies to HDevelop windows, that is windows opened via dev_open_window, which are in fact buffer windows.

Value range: a string containing up to 1023 characters.

'anti_aliasing' Enables or disables the anti aliasing of contours and regions.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'graphics_stack' Enables or disables the graphics stack of a window. With activated graphics stack changing the window part with set_part triggers the display of previously displayed objects with respect to the new part. With this it is possible to move or scale the contents of the window. This parameter has no effect on the graphics stack of a HDevelop graphics window.

List of values: *'true'*, *'false'*.

Default: *'false'*.

'graphics_stack_max_element_num' Maximum number of elements in the graphics stack. This number corresponds to the number of display operations recorded. This parameter has no effect on the graphics stack of a HDevelop graphics window and is effective not before the next call to a display operator.

Suggested values: positive number or *'unlimited'*.

Default: *50*.

'graphics_stack_max_memory_size' Limits the memory consumption of the graphics stack. If this limit is exceeded, older display operations are removed from the graphics stack. This parameter has no effect on the graphics stack of a HDevelop graphics window and is effective not before the next call to a display operator.

Suggested values: memory limit in bytes or *'unlimited'*.

Default: *'unlimited'*.

'pixel_grid_enable' Enables or disables the pixel grid. If the pixel grid is activated, a grid is drawn around the displayed pixels as soon as the image part is enlarged sufficiently (see *'pixel_grid_min_resolution'*). This parameter is effective as soon as the window content is updated.

List of values: *'true'*, *'false'*.

Default: *'false'*.

Restriction: Has no effect, when the HDevelop option `Pixel Grid` is enabled.

'pixel_grid_min_resolution' Specifies the minimum required resolution of an image pixel in the graphics window which is necessary for the pixel grid to be drawn. This parameter has an effect only if *'pixel_grid_enable'* has been set to *'true'*. Assuming this value is set to *20*, the pixel grid will be visible as soon as a pixel in the displayed image part takes up at least *20* x *20* pixels in the graphics window. Furthermore, this parameter is effective as soon as the window content is updated.

Suggested values: Positive integer.

Default: *20*.

'pixel_grid_color' Specifies the color of the lines of the pixel grid. This parameter has an effect only if *'pixel_grid_enable'* has been set to *'true'*. Furthermore, this parameter is effective as soon as the window content is updated.

Suggested values: Color value, see `set_color` for an overview.

Default: *'#696969c0'*.

Restriction: Has no effect, when the HDevelop option `Pixel Grid` is enabled.

'pixel_grid_line_width' Specifies the line width in pixels that is used to display the lines of the pixel grid. This parameter has an effect only if *'pixel_grid_enable'* has been set to *'true'*. Furthermore, this parameter is effective as soon as the window content is updated.

Value range: [*1.0*, *2000.0*].

Default: *1.0*.

Parameters concerning the *'3d_plot'* (see `set_paint`):

'angle_of_view' Set the angle of view of the virtual camera used to display the 3D plot. The angle must be provided in radians and lie within the interval $[0, PI)$. An angle of 0 radians implies orthogonal projection. Another way to choose orthogonal projection is to set *'angle_of_view'* to *'orthogonal'*.

Suggested values: *'orthogonal'* or an angle in radians.

Default: *1.22173* (70 degrees).

'axis_captions' The axis captions for row, column, and height can be set by passing a tuple containing the strings for the captions in the specified order row, column, and height axis, (e.g., ['row','col','height']). Pass empty strings to display no captions at all.

Value range: A tuple containing three strings. Each string can contain up to 31 characters.

Default: *[' ',' ',' ']*.

'caption_color' The color of the axis captions for row, column, and height. You can either pass one color for all three axes or a tuple of three colors to set individual colors for each axis.

Suggested values: a string or tuple with three strings containing the color name (e.g., *'black'*, *'red'*).

Default: *'black'*.

'plot_quality' Influences the quality of the 3D plot. Depending on the capabilities of your graphics card better settings for the quality may reduce the frame rate of the 3D plot significantly.

List of values: *'low'*, *'medium'*, *'good'*, *'best'*.

Default: *'medium'*.

'scale_plot' If set to *'true'* the height values of an image are transformed into the interval [0,255] before display. If set to 'false' the aspect ratio between row/column and height is considered. Images of the type byte, cyclic, or direction are not scaled, i.e. in their cases this parameter is ignored.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'display_grid' If set to *'true'* a grid is displayed at height 0.

List of values: *'true'*, *'false'*.

Default: *'true'*.

'display_axes' If set to *'true'* coordinate axes are displayed. **List of values:** *'true'*, *'false'*.

Default: *'true'*.

Attention

The parameters *'anti_aliasing'*, *'flush'*, *'graphics_stack'*, *'graphics_stack_max_element_num'*, *'graphics_stack_max_memory_size'*, and *'region_quality'* depend on the library libcanvas, which might not be available on embedded systems.

The parameter *'window_title'* cannot be set for buffer windows.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Param** (input_control) ..... string ⇝ *string*
Name of the parameter.
Default: 'flush'
List of values: `Param` ∈ {'plot_quality', 'axis_captions', 'caption_color', 'scale_plot', 'angle_of_view', 'display_grid', 'display_axes', 'window_title', 'background_color', 'anti_aliasing', 'flush', 'graphics_stack', 'graphics_stack_max_element_num', 'graphics_stack_max_memory_size', 'pixel_grid_enable', 'pixel_grid_min_resolution', 'pixel_grid_color', 'pixel_grid_line_width', 'region_quality'}

▷ **Value** (input_control) ..... string(-array) ⇝ *string* / real / integer
Value to be set.
Default: 'false'
List of values: `Value` ∈ {'true', 'false', 'low', 'good', 'medium', 'best', 'orthogonal', 'black', 'white', 'red', 'green', 'blue', 'unlimited'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`disp_image`

See also

`get_window_param`, `set_window_attr`, `set_part_style`

Module

Foundation

13.8 Text

```
disp_text ( : : WindowHandle, String, CoordSystem, Row, Column,
    Color, GenParamName, GenParamValue : )
```

Display text in a window.

`disp_text` displays text in the graphics window `WindowHandle` at the position (`Row`,`Column`).

If only a single position is defined, one text line is displayed for each element of `String`. Also, *'\n'* will be interpreted as a newline character, i.e., a line break is performed.

If multiple positions are defined, only a single string or one string for each position is allowed in `String`. In this case, line breaks have to be forced with *'\n'*.

Newlines (*'\n'*) at the end of `String` are ignored.

The position of the text may be specified in window coordinates (`CoordSystem` = *'window'*) or in image coordinates (`CoordSystem` = *'image'*), which is useful when using zoomed images.

In addition to supplying (`Row`,`Column`) coordinates, it is also possible to pass predefined values to `Row` and `Column` to display the text at a fixed Position in the window (only if `CoordSystem` = *'window'*).:

'top', *'left'*	*'top'*, *'center'*	*'top'*, *'right'*
'center', *'left'*	*'center'*, *'center'*	*'center'*, *'right'*
'bottom', *'left'*	*'bottom'*, *'center'*	*'bottom'*, *'right'*

The parameter `Color` also accepts tuples of values. In that case, the specified colors are used cyclically for every new text position or for every new text line if a single position is used.

Generic Parameters

`disp_text` may display the `String` within a box (default). This behavior and the look of the box are defined with the generic parameters in `GenParamName` and `GenParamValue`.

'box' If *'box'* is set to *'true'*, the text is written within a box. The look of the box and its optional shadow can be configured with the generic parameters below.
List of values: *'true'* and *'false'*
Default: *'true'*

'box_color' Sets the color of the box.
List of values: a string containing the color name (e.g., *'white'*, *'red'*, or *'#aa00bba0'*)
Default: *'#fce9d4'* (which is a light orange)

'shadow' If *'shadow'* is set to *'true'*, an additional shadow is displayed beneath the box (if *'box'* is *'true'*).
List of values: *'true'* and *'false'*
Default: *'true'* if *'box_color'* is set to a color without alpha value, *'false'* otherwise

'shadow_color' Sets the color of the shadow if *'shadow'* is *'true'*.
List of values: a string containing the color name (e.g., *'black'*, *'red'*, or *'#aa00bba0'*)
Default: *'#f28d26'* (which is a darker orange) if *'box_color'* is not set, *'white'* otherwise

'border_radius' Controls the roundness of the box's corners. For sharp corners set it to *0*, for smoother corners to higher values.
List of values: positive real numbers or *0*
Default: 2

'box_padding' Controls to which amount in pixels the box is extended around the text.
List of values: positive real number
Default: 0

'shadow_sigma' Controls to which amount the shadow beneath the box is blurred. Set it to *0* for a sharp shadow.
List of values: positive real number or *0*
Default: 1.5

'shadow_dx' and 'shadow_dy' Controls the offset of the shadow in column (*'shadow_dx'*) and row (*'shadow_dy'*) direction in pixels.
List of values: any real number
Default: 2

'backdrop_blur_sigma' Controls to which amount the background of the box is blurred. s only an effect if *'box'* is *'true'* and *'box_color'* is not opaque (i.e. alpha is not 255). It is recommended to use *'backdrop_blur_sigma'* with *'shadow'* set to *'false'*.
List of values: positive real number or *0*
Default: 0

'rotate_phi' Angle in degrees to rotate the displayed text.
List of values: any real number
Default: 0

'rotate_col' Column coordinate of the rotation center if *'rotate_phi'* is different to zero. Choose either a column coordinate by setting a real number or determine the rotation center on the left edge (*'text_left'*), the right edge (*'text_right'*), or in the center (*'text_center'*) of the text box.

List of values: any real number or *'text_left'*, *'text_center'*, *'text_right'*

Default: *'text_center'*

'rotate_row' Row coordinate of the rotation center if *'rotate_phi'* is different to zero. Choose either a row coordinate by setting a real number or determine the rotation center on the top edge (*'text_top'*), the bottom edge (*'text_bottom'*), or in the center (*'text_center'*) of the text box.

List of values: any real number or *'text_top'*, *'text_center'*, *'text_bottom'*

Default: *'text_center'*

'border_color' Sets the color of the border of the text box.

List of values: a string containing the color name (e.g., *'black'*, *'red'*, or *'#aa00bba0'*)

Default: *'#ffffff'*

'border_width' Controls the width of the text box border.

List of values: positive real numbers or *0*

Default: 0

Attention

`disp_text` depends on the library libhcanvas, which might not be available on embedded systems.

Parameters

▷ **`WindowHandle`** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **`String`** (input_control) . . . . . string(-array) ⇝ *string*
A tuple of strings containing the text message to be displayed. Each value of the tuple will be displayed in a single line.
Default: 'hello'

▷ **`CoordSystem`** (input_control) . . . . . string ⇝ *string*
If set to 'window', the text position is given with respect to the window coordinate system. If set to 'image', image coordinates are used (this may be useful in zoomed images).
Default: 'window'
List of values: `CoordSystem` ∈ {'window', 'image'}

▷ **`Row`** (input_control) . . . . . point.y(-array) ⇝ *integer* / real / string
The vertical text alignment or the row coordinate of the desired text position.
Default: 12
List of values: `Row` ∈ {12, 'bottom', 'center', 'top'}

▷ **`Column`** (input_control) . . . . . point.x(-array) ⇝ *integer* / real / string
The horizontal text alignment or the column coordinate of the desired text position.
Default: 12
List of values: `Column` ∈ {12, 'center', 'left', 'right'}

▷ **`Color`** (input_control) . . . . . string(-array) ⇝ *string*
A tuple of strings defining the colors of the texts.
Default: 'black'
List of values: `Color` ∈ {'black', 'blue', 'yellow', 'red', 'green', 'cyan', 'magenta', 'forest green', 'lime green', 'coral', 'slate blue'}

▷ **`GenParamName`** (input_control) . . . . . attribute.name-array ⇝ *string*
Generic parameter names.
Default: []
List of values: `GenParamName` ∈ {'backdrop_blur_sigma', 'box', 'shadow', 'box_color', 'shadow_color', 'border_radius', 'box_padding', 'shadow_sigma', 'shadow_dx', 'shadow_dy', 'rotate_phi', 'rotate_col', 'rotate_row', 'border_color', 'border_width'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value-array ⇝ *string* / integer / real
Generic parameter values.
Default: []
List of values: `GenParamValue` ∈ {'true', 'false', 'white', 'red', 'forest green', 'black', 'blue', 5.0}

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
disp_text (WindowHandle, 'Display some text in a box', 'window', 12, 12, \
           'black', [], [])
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_font, get_string_extents

Alternatives

write_string

See also

get_string_extents, set_font

Module

Foundation

get_font (: : WindowHandle : Font)

Get the current font.

get_font queries the name of the font used in the output window WindowHandle. The font is used by the operators write_string, read_string etc. The font is set by the operator set_font. Text windows as well as windows for image display use fonts. Both types of windows have a default font that can be modified with set_system('default_font',Font) prior to opening the window. A list of all available fonts can be obtained using query_font.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Font** (output_control) ..... string ⇝ *string*
Name of the current font.

Example

```
get_font(WindowHandle,CurrentFont)
set_font(WindowHandle,MyFont)
write_string(WindowHandle,['The name of my Font is:',Myfont])
new_line(WindowHandle)
set_font(WindowHandle,CurrentFont)
```

Result

get_font returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, query_font

Possible Successors

set_font

See also

set_font, query_font, open_window, set_system

Module

Foundation

get_font_extents (: : WindowHandle : MaxAscent, MaxDescent, MaxWidth, MaxHeight)

Get the maximum size of all characters of a font.

get_font_extents queries the maximum width (MaxWidth), height (MaxHeight), and extension above and below the baseline (MaxAscent and MaxDescent, respectively) of the font that is currently set for the window WindowHandle. The sizes are measured in pixels.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **MaxAscent** (output_control) ..... extent.y ⇝ *integer* / real
  Maximum height above baseline.
- ▷ **MaxDescent** (output_control) ..... extent.y ⇝ *integer* / real
  Maximum extension below baseline.
- ▷ **MaxWidth** (output_control) ..... extent.x ⇝ *integer* / real
  Maximum character width.
- ▷ **MaxHeight** (output_control) ..... extent.y ⇝ *integer* / real
  Maximum character height.

Result

get_font_extents returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_font

Possible Successors

set_tposition, write_string, read_string, read_char

See also

get_string_extents, set_tposition, set_font

Module

Foundation

get_string_extents (: : WindowHandle, Values : Ascent, Descent, Width, Height)

Get the spatial size of a string.

get_string_extents queries width and height of the output size of a string using the font of the window. Thereby strings with multiple lines are treated as if concatenated into a single line. In addition, the extension above and below the baseline is returned (Ascent and Descent, respectively).

The sizes are measured in the coordinate system of the window (for text windows in pixels). Using get_string_extents, it is possible to determine text output and input independently from the used font. The conversion from integer numbers and floating point numbers to text strings is the same as in write_string.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Values** (input_control) ..... string(-array) ⇝ *string* / real / integer
  Values to consider.
  Default: 'test_string'
- ▷ **Ascent** (output_control) ..... extent.y ⇝ *integer* / real
  Maximum height above baseline.
- ▷ **Descent** (output_control) ..... extent.y ⇝ *integer* / real
  Maximum extension below baseline.
- ▷ **Width** (output_control) ..... extent.x ⇝ *integer* / real
  Text width.
- ▷ **Height** (output_control) ..... extent.y ⇝ *integer* / real
  Text height.

Result

get_string_extents returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_font

Possible Successors

set_tposition, write_string, read_string, read_char

See also

get_font_extents, set_tposition, set_font

Module

Foundation

get_tposition (: : WindowHandle : Row, Column)

Get cursor position.

get_tposition queries the current position of the text cursor in the output window. The position is measured in the image coordinate system. The next output of text in this window starts at the cursor position. The left end of the baseline for writing the next string (not considering descenders) is placed on this position. The position is changed by the output or input of text (write_string, read_string) or by an explicit change of position by (set_tposition, new_line).

Attention

If the output text does not fit completely into the window, an exception is raised. This can be avoided by set_check('~text').

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Row** (output_control) ..... point.y ⇝ *integer*
  Row index of text cursor position.

▷ **Column** (output_control) . . . . . point.x ⇝ *integer*
Column index of text cursor position.

Result

`get_tposition` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `set_font`

Possible Successors

`set_tposition`, `write_string`, `read_string`, `read_char`

See also

`new_line`, `read_string`, `set_tposition`, `write_string`, `set_check`

Module

Foundation

new_line (: : WindowHandle :)

Set the position of the text cursor to the beginning of the next line.

`new_line` sets the position of the text cursor to the beginning of the next line. The new position depends on the current font. The left end of the baseline for writing the following text string (not considering descenders) is placed on this position.

If the next line does not fit into the window the content of the window is scrolled by the height of one line in the upper direction. In order to reach the correct new cursor position the font used in the next line must be set before `new_line` is called. The position is changed by the output or input of text (`write_string`, `read_string`) or by an explicit change of position by (`set_tposition`).

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

Result

`new_line` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `set_font`, `write_string`

Alternatives

`get_tposition`, `get_string_extents`, `set_tposition`

See also

`write_string`, `set_font`

Module

Foundation

query_font (: : WindowHandle : Font)

Query the available fonts.

query_font queries the fonts available for text output in the output window. They can be set with the operator set_font using the appropriate syntax. Fonts are used by the operators write_string, read_char, read_string and new_line.

Attention

For different machines the available fonts may differ a lot. Therefore query_font will return different fonts on different machines.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Font** (output_control) ..... string-array ⇝ *string*
  Tuple with available font names.

Example

```
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
set_check('~text')
query_font(WindowHandle,Fontlist)
set_color(WindowHandle,'white')
for i:=0 to |Fontlist|-1 by 1
  set_display_font (WindowHandle,16,Fontlist[i],'true','false')
  write_string(WindowHandle,Fontlist[i])
  new_line(WindowHandle)
endfor
```

Result

query_font returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

set_font, write_string, read_string, read_char

See also

set_font, write_string, read_string, read_char, new_line

Module

Foundation

read_char (: : WindowHandle : Char, Code)

Read a character from a window.

read_char reads a character from the keyboard in the input window (= output window). If the character is printable it is returned in Char. If a control key has been pressed, this will be indicated by the value of Code. Some important keys are recognizable by this value. Possible values are:

'character': printable character

'left': cursor left

'right': cursor right

'up': cursor up

'down': cursor down

'insert': insert

'none': none of these keys

'canceled': `read_char` has been canceled (not always possible, e.g., in native X11 windows)

Parameters

▷ **`WindowHandle`** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **`Char`** (output_control) ..... string ⇝ *string*
Input character (if it is not a control character).

▷ **`Code`** (output_control) ..... string ⇝ *string*
Code for input character.

Result

`read_char` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `set_font`

Alternatives

`read_string`, `fread_char`, `fread_string`

See also

`write_string`, `set_font`

Module

Foundation

`read_string` `( : : WindowHandle, InString, Length : OutString )`

Read a string in a text window.

This operator does not work in an HDevelop graphics window opened with `dev_open_window`.

`read_string` reads a string with a predetermined maximum length (`Length`) from the keyboard in the input window (= output window). The string is read from the current position of the text cursor using the current font. The maximum length has to be small enough to keep the string within the right window boundary. A default string which can be edited or simply accepted by the user may be provided. After text input the text cursor is positioned at the end of the edited string. Commands for editing:

RETURN finish input

BACKSPACE delete the character on the left side of the cursor and move the cursor to this position.

The length is stated as number of characters. If *'filename_encoding'* is set to *'locale'* with `set_system`, the length is stated in number of bytes.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **InString** (input_control) . . . . . string ⇝ *string*
Default string (visible before input).
Default: ”

▷ **Length** (input_control) . . . . . integer ⇝ *integer*
Maximum number of characters.
Default: 32
Restriction: Length > 0 && Length <= 1024

▷ **OutString** (output_control) . . . . . string ⇝ *string*
Read string.

Result

`read_string` returns `2` (`H_MSG_TRUE`) if the text window is valid and a string of maximal length fits within the right window boundary. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `set_font`

Alternatives

`read_char`, `fread_string`, `fread_char`

See also

`set_tposition`, `new_line`, `open_window`, `set_font`, `set_color`

Module

Foundation

set_font (: : WindowHandle, Font :)

Set the font used for text output.

`set_font` sets the font `Font` for the output window `WindowHandle`. All subsequent text outputs (e.g., with the operators `disp_text`, `write_string` or `read_string`) will now use the new font instead of the default font (see `set_system('default_font',Font)`). All available fonts can be queried with `query_font`.

The syntax for the specification of `Font` is the following:

```
FONTNAME[-STYLE]-FONT\_SIZE.
```

The optional STYLE may be one of the following (other values are possible as well):

- Normal,
- Bold,
- BoldItalic, or
- Italic.

The FONT_SIZE is measured in pixels.

An example of a valid string for `Font` would be

’Courier-Bold-14’.

Attention

For different machines the available fonts may differ a lot. Therefore it is suggested to use the operator query_font or the procedure set_display_font.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Font** (input_control) ..... string ⇝ *string*
Name of new font.

Example

```
dev_get_window (WindowHandle)
query_font (WindowHandle, Font)
* Specify font name and size
FontWithSize := Font[0]+'-20'
set_font (WindowHandle, FontWithSize)
dev_disp_text ('Font set to: '+FontWithSize, 'window', 20, 12, 'black', \
               [], [])
* Specify font name, style, and size
FontWithStyleAndSize := Font[0]+'-Bold-20'
set_font (WindowHandle, FontWithStyleAndSize)
dev_disp_text ('Font set to: '+FontWithStyleAndSize, 'window', 50, 12, \
               'black', [], [])
```

Result

set_font returns 2 (H_MSG_TRUE) if the font name can be resolved. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

query_font

See also

get_font, query_font, open_window

Module

Foundation

set_tposition (: : WindowHandle, Row, Column :)

Set the position of the text cursor.

set_tposition sets the position of the text cursor in the output window. The reference position is the upper left corner of an upper case character.

The position is measured in the image coordinate system. The position of the text cursor can be marked, e.g., by an underscore. The next text output in this window starts at the cursor position. The left end of the baseline for writing the following text string (not considering descenders) is placed on this position.

The position is changed by the output or input of text (write_string, read_string) or by an explicit change of position by (set_tposition, new_line). In order to stop the display of the cursor, the operator set_tshape with the parameter ”invisible” can be used.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row index of text cursor position.
Default: 24

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column index of text cursor position.
Default: 12

Result

`set_tposition` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`write_string`, `read_string`

Alternatives

`new_line`

See also

`read_string`, `write_string`

Module

Foundation

write_string (: : WindowHandle, String :)

Print text in a window.

`write_string` prints `String` in the output window starting at the current cursor position. The output text has to fit within the right window boundary (the width of the string can be queried by `get_string_extents`).

The font currently assigned to the window will used. The text cursor is positioned at the end of the text.

`write_string` can output all three types of data used in HALCON . The conversion to a string is guided by the following rules:

- strings are not converted.
- integer numbers are converted without any spaces before or after the number.
- floating numbers are printed (if possible) with a floating point and without an exponent.
- the resulting strings are concatenated without spaces.

For buffering of texts see `set_system` with the flag 'flush_graphic'.

Attention

If clipping at the window boundary is desired, exceptions can be switched off by `set_check('~text')`.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **String** (input_control) . . . . . string(-array) ⇝ *string* / integer / real
Tuple of output values (all types).
Default: 'hello'

Result

`write_string` returns `2` (`H_MSG_TRUE`) if the window is valid and the output text fits within the current line (see `set_check`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `set_font`, `get_string_extents`

Alternatives

`fwrite_string`

See also

`set_tposition`, `get_string_extents`, `set_font`, `set_system`, `set_check`

Module

Foundation

13.9 Window

clear_window (: : WindowHandle :)

Delete the contents of an output window.

`clear_window` deletes all entries in the output window. The window (background and edge) is reset to its original state. Parameters assigned to this window (e.g., with `set_color`, `set_paint`, etc.) remain unmodified.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

Result

If the output window is valid `clear_window` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Alternatives

`clear_rectangle`, `disp_rectangle1`

See also

`open_window`

Module

Foundation

close_window (: : WindowHandle :)

Close an output window.

close_window closes a window which have been opened by open_window. Afterwards the output device or the window area, respectively, is ready to accept new calls of open_window.

Parameters

▷ **WindowHandle** (input_control) ..... window(-array) ⇝ *handle*
Window handle.

Result

If the output window is valid close_window returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- WindowHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

open_window

See also

open_window

Module

Foundation

copy_rectangle (: : WindowHandleSource, WindowHandleDestination, Row1, Column1, Row2, Column2, DestRow, DestColumn :)

Copy all pixels within rectangles between output windows.

copy_rectangle copies all pixels from the window WindowHandleSource in the window WindowHandleDestination. It copies pixels which reside inside a rectangle which is specified by parameters Row1, Column1, Row2 and Column2. The target position is specified through the upper left corner of the rectangle (DestRow, DestColumn).

If you want to move more than one rectangle, you may pass them at once (in form of the tuple mode).

You may use copy_rectangle to copy edited graphics from an “invisible” window in a visible window. Therefore a window with the option ’buffer’ is opened. The graphics is then displayed in this window and is copied in a visible window afterwards. The advantage of this strategy is, that copy_rectangle is much more rapid than output operators as, e.g., disp_channel. This means a particular advantage while using demo programs. You could even realize short “clips”: you have to create for every image of a sequence a window of a ’buffer’ type and pass the data into it. Output is the image sequence whereat all buffers are copied one after another in a visible window.

Attention

Both windows have to reside on the same display.

Parameters

▷ **WindowHandleSource** (input_control) . . . . . window ⇝ *handle*
Source window handle.

▷ **WindowHandleDestination** (input_control) . . . . . window ⇝ *handle*
Destination window handle.

▷ **Row1** (input_control) . . . . . rectangle.origin.y(-array) ⇝ *integer*
Row index of upper left corner in the source window.
Default: 0
Value range: 0 ≤ `Row1` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column1** (input_control) . . . . . rectangle.origin.x(-array) ⇝ *integer*
Column index of upper left corner in the source window.
Default: 0
Value range: 0 ≤ `Column1` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Row2** (input_control) . . . . . rectangle.corner.y(-array) ⇝ *integer*
Row index of lower right corner in the source window.
Default: 128
Value range: 0 ≤ `Row2` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1
Restriction: Row2 >= Row1

▷ **Column2** (input_control) . . . . . rectangle.corner.x(-array) ⇝ *integer*
Column index of lower right corner in the source window.
Default: 128
Value range: 0 ≤ `Column2` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1
Restriction: Column2 >= Column1

▷ **DestRow** (input_control) . . . . . point.y(-array) ⇝ *integer*
Row index of upper left corner in the target window.
Default: 0
Value range: 0 ≤ `DestRow` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **DestColumn** (input_control) . . . . . point.x(-array) ⇝ *integer*
Column index of upper left corner in the target window.
Default: 0
Value range: 0 ≤ `DestColumn` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Example

```
read_image(Image,'monkey')
open_window(0,0,-1,-1,'root','buffer','',WindowHandleBuffer)
disp_obj(Image,WindowHandleBuffer)
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
draw_rectangle1 (WindowHandle, Row1, Column1, Row2, Column2)
copy_rectangle(WindowHandleBuffer,WindowHandle,\
               Row1, Column1, Row2, Column2,Row1,Column1)
close_window(WindowHandleBuffer)
```

Result

If the output window is valid and if the specified parameters are correct `close_window` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Possible Successors

`close_window`

Alternatives

`move_rectangle`, `slide_image`

See also

`open_window`

Module

Foundation

dump_window (: : WindowHandle, Device, FileName :)

Write the window content to a file.

`dump_window` writes the content of the window to a file. You may continue to process this file by convenient printers or other programs. The content of a display is prepared for each special device (`Device`), i.e., it is formatted in a manner, that you may print this file directly or it can be processed furthermore by a graphical program.

To transform gray values the current color table of the window is used.

Possible values for `Device`

'postscript': PostScript - file. File extension: '.ps'

'postscript',Width,Height: PostScript - file with specification of the output size. $Width$ and $Height$ refer to the size. In this case a tuple with three values as `Device` is passed.
File extension: '.ps'

'tiff': TIFF format; Compression is possible with Adobe deflate compression (*'deflate [num]'*, lossless), JPEG (*'jpeg [num]'*, lossy), LZW (*'lzw'*, lossless), and PackBits (*'packbits'*, lossless). Here, *'[num]'* denotes an optional specification of a compression parameter. For *'deflate'*, a number between *0* (no compression) and *9* (maximum compression) can be specified. For *'jpeg'*, a number between *0* and *100* can be specified. The semantics are identical to the semantics of 'jpeg' described below. The different options can be accumulated by appending them separated by a space character. Examples: *'tiff deflate 9'*: Maximum Adobe deflate compression; *'tiff jpeg 90'*: JPEG compression with high quality; *'tiff lzw'*: LZW compression; *'tiff'* or *'tiff none'*: no compression.
File extension: '.tif'

'bmp': Windows-BMP format, RGB image, 3 bytes per pixel. The color resolution depends on the VGA card.
File extension: '.bmp'

'jpeg': JPEG format, with lost of information; together with the format string the quality value determining the compression rate can be provided: e.g., 'jpeg 30'.
File extension: '.jpg'

'jpegxr': JPEG-XR format (lossless and lossy compression); together with the format string the quality value determining the compression rate can be provided: e.g., 'jpegxr 30'.
File extension: '.jxr'

'jp2': JPEG2000 format (lossless and lossy compression); together with the format string the quality value determining the compression rate can be provided (e.g., 'jp2 40'). This value corresponds to the ratio of the size

of the compressed image and the size of the uncompressed image (in percent). As lossless JPEG2000 compression reduces the file size significantly already, only smaller values (typically smaller than 50) influence the file size. Is no value provided (and only then), the image is compressed without loss.

File extension: '.jp2'

'png': PNG format (lossless compression); together with the format string, a compression level between 0 and 9 can be specified, where 0 corresponds to no compression and 9 to the best possible compression. Alternatively, the compression can be selected with the following strings: 'best', 'fastest', and 'none'. Hence, examples for correct parameters are 'png', 'png 7', and 'png none'.

File extension: '.png'

Attention

Under Unix-like systems, the graphics window must be completely visible on the root window, because otherwise the contents of the window cannot be read due to limitations in X Windows. If larger graphical displays are to be written to a file, the window type *'pixmap'* can be used.

Parameters

▷ **`WindowHandle`** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **`Device`** (input_control) ..... string(-array) ⇝ *string* / integer
Name of the target device or of the graphic format.
Default: 'postscript'
Suggested values: `Device` ∈ {'postscript', 'tiff', 'tiff deflate 9', 'tiff jpeg 90', 'tiff lzw', 'tiff packbits', 'bmp', 'jpeg', 'jpeg 100', 'jpeg 80', 'jpeg 60', 'jpeg 40', 'jpeg 20', 'jpegxr', 'jpegxr 50', 'jpegxr 40', 'jpegxr 30', 'jpegxr 20', 'jp2', 'jp2 50', 'jp2 40', 'jp2 30', 'jp2 20', 'png', 'png best', 'png fastest', 'png none'}

▷ **`FileName`** (input_control) ..... filename.write ⇝ *string*
File name (without extension).
Default: 'halcon_dump'

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
read_image (Image, 'fabrik')
gen_circle (Circle, 200, 200, 100.5)
dev_display (Image)
dev_display (Circle)
dump_window(WindowHandle,'postscript','halcon_dump')
```

Result

If the appropriate window is valid and the specified parameters are correct `dump_window` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `set_draw`, `set_color`, `set_colored`, `set_line_width`, `disp_region`

Possible Successors

`system_call`

See also

`open_window`, `set_system`, `dump_window_image`

Module

Foundation

dump_window_image (: Image : WindowHandle :)

Write the window content in an image object.

dump_window_image writes the content of the graphics window (WindowHandle) in an image (Image). To transform gray values the current color table of the window is used.

Attention

Under Unix-like systems, the graphics window must be completely visible on the root window, because otherwise the contents of the window cannot be read due to limitations in X Windows.

Parameters

▷ **Image** (output_object) ..... image ⇝ *object :* byte
Saved image.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

Result

If the window is valid dump_window_image returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_draw, set_color, set_colored, set_line_width, disp_region

See also

open_window, set_system, dump_window

Module

Foundation

flush_buffer (: : WindowHandle :)

Flush the contents of a window.

flush_buffer flushes the contents of the window WindowHandle.

flush_buffer is only necessary if the window parameter *'flush'* has been set to *'false'* with set_window_param. If *'flush'* is *'false'* all display operations (such as disp_obj or disp_text) are redirected to a buffer and have no effect on WindowHandle (this applies for all window modes). flush_buffer copies the contents of this buffer to the window WindowHandle.

This is very useful to avoid flickering by batching several display operations (e.g., a clear_window followed by a disp_obj) and displaying the final result with flush_buffer.

This does not apply to drawing objects, which are always updated.

Attention

flush_buffer depends on the library libcanvas, which might not be available on embedded systems.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

Example

```
read_image (Image, 'printer_chip/printer_chip_01')
threshold (Image, Region, 128, 255)
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
```

```
set_window_param (WindowHandle, 'flush', 'false')
dev_display (Image)
dev_display (Region)
disp_text (WindowHandle, 'Result of threshold', 'window', \
           12, 12, 'black', [], [])
* nothing is displayed until flush_buffer is called
flush_buffer (WindowHandle)
```

Result

If the window exists `flush_buffer` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`disp_obj`

Possible Successors

`dump_window_image`

See also

`set_window_param`

Module

Foundation

get_disp_object_model_3d_info (: : WindowHandle, Row, Column, Information : Value)

Get the depth or the index of a displayed 3D object model.

`get_disp_object_model_3d_info` returns information on the 3D object models that have been displayed with `disp_object_model_3d` in the window `WindowHandle`. The requested information at the positions (`Row`, `Column`) is returned in `Value`.

The following values can be queried via `Information`:

'object_index' The indices of the 3D object models that have been displayed at the positions (`Row`, `Column`). If no 3D object model was displayed at this position, -1 is returned. In order to retrieve this information, `disp_object_model_3d` must have been called with the generic parameter *'object_index_persistence'* set to *'true'*.

'depth' The depth (i.e. the Z coordinate in the camera coordinate system) at the positions (`Row`, `Column`). If no 3D object model was displayed at one of these positions, -1.0 is returned for this position. In order to retrieve this information, `disp_object_model_3d` must have been called with the generic parameter *'depth_persistence'* set to *'true'*.

The window coordinates `Row`, `Column` must be provided in respect to the current image part. Thereby they are understood in edge centered subpixel accurate coordinates, see Transformations / 2D Transformations. Given the current image part (`row1`,`column1`,`row2`,`column2`, in the HALCON standard coordinate system), the upper left corner corresponds to the coordinates (`row1` - 0.5, `column1` - 0.5). Accordingly, the bottom right corner corresponds to the coordinates (`row2` - 0.5, `column2` - 0.5). Use `get_mposition_sub_pix` or `get_mbutton_sub_pix` to obtain these coordinates directly. In case the window coordinates correspond to values outside the current image part, the operator behavior is undefined.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row** (input_control) ..... integer(-array) ⇝ *real* / integer
Row coordinates.

▷ **Column** (input_control) ..... integer(-array) ⇝ *real* / integer
Column coordinates.

▷ **Information** (input_control) ..... string(-array) ⇝ *string*
Information.
Default: 'depth'
List of values: `Information` ∈ {'depth', 'object_index'}

▷ **Value** (output_control) ..... integer(-array) ⇝ *integer* / real
Indices or the depth of the objects at (Row,Column).

Result

`get_disp_object_model_3d_info` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`disp_object_model_3d`, `get_mbutton`, `get_mbutton_sub_pix`, `get_mposition`, `get_mposition_sub_pix`

See also

`disp_object_model_3d`

Module

3D Metrology

get_os_window_handle (: : WindowHandle : OSWindowHandle, OSDisplayHandle)

Get the operating system window handle.

This operator does not work in an HDevelop graphics window opened with `dev_open_window`.

`get_os_window_handle` returns the operating system window handle of the HALCON window `WindowHandle` in `OSWindowHandle`. Under Unix-like systems, additionally the operating system display handle is returned in `OSDisplayHandle`. The operating system window handle can be used to access the window using functions from the operating system, e.g., to draw in a user-defined manner into the window. Under Windows, `OSWindowHandle` can be cast to a variable of type HWND. Under Unix-like systems, `OSWindowHandle` can be cast into a variable of type Window, while `OSDisplayHandle` can be cast into a variable of type Display.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **OSWindowHandle** (output_control) ..... pointer ⇝ *integer*
Operating system window handle.

▷ **OSDisplayHandle** (output_control) ..... pointer ⇝ *integer*
Operating system display handle (under Unix-like systems only).

Example

```
/* Draw a line into a HALCON window under Unix-like systems using X11 calls. */
```

```
#include "HalconC.h"
#include <X11/X.h>
#include <X11/Xlib.h>

int main(int argc, char **argv)
{
  Hlong hwin, win, disp;
  Display *display;
  Window window;
  GC gc;
  XGCValues values;
  static char dashes[] = { 20, 20 };

  open_window(0, 0, 500, 500, 0, "visible", "", &hwin);
  get_os_window_handle(hwin, &win, &disp);
  display = (Display *)disp;
  window = (Window)win;
  gc = XCreateGC(display, window, 0, &values);
  XSetFunction(display, gc, GXset);
  XSetLineAttributes(display, gc, 10, LineOnOffDash, CapRound, JoinRound);
  XSetDashes(display, gc, 0, dashes, 2);
  XSetForeground(display, gc, WhitePixel(display, DefaultScreen(display)));
  XSetBackground(display, gc, BlackPixel(display, DefaultScreen(display)));
  XDrawLine(display, win, gc, 20, 20, 480, 480);
  XFlush(display);
  XFreeGC(display, gc);
  wait_seconds(5);
  return 0;
}

/* Draw a line into a HALCON window under Windows using GDI calls. */
#include "HalconC.h"
#include "windows.h"

int main(int argc, char **argv)
{
  Hlong hwin, win, disp;
  HDC hdc;
  HPEN hpen;
  HPEN *hpen_old;
  LOGBRUSH logbrush;
  POINT point;
  static DWORD dashes[] = { 20, 20 };

  open_window(0, 0, 500, 500, 0, "visible", "", &hwin);
  get_os_window_handle(hwin, &win, &disp);
  logbrush.lbColor = RGB(255,255,255);
  logbrush.lbStyle = BS_SOLID;
  hpen =  ExtCreatePen(PS_USERSTYLE|PS_GEOMETRIC, 10, &logbrush, 2, dashes);
  hdc = GetDC((HWND)win);
  hpen_old = (HPEN *)SelectObject(hdc, hpen);
  MoveToEx(hdc, 20, 20, &point);
  LineTo(hdc, 480, 480);
  DeleteObject(SelectObject(hdc, hpen_old));
  ReleaseDC((HWND)win, hdc);
  wait_seconds(5);
  return 0;
}
```

Result

If the window is valid `get_os_window_handle` returns `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Module

Foundation

`get_window_attr` `( : : AttributeName : AttributeValue )`

Get window characteristics.

The operator `get_window_attr` can be used to read characteristics of graphics windows that were set using `set_window_attr`. The following parameters of a window may be queried:

'border_width' Width of the window border in pixels.

'border_color' Color of the window border.

'background_color' Background color of the window.

'window_title' Name of the window in the title bar.

Parameters

▷ **`AttributeName`** (input_control) ..... string ⇝ *string*
Name of the attribute that should be returned.
List of values: `AttributeName` ∈ {'border_width', 'border_color', 'background_color', 'window_title'}

▷ **`AttributeValue`** (output_control) ..... string ⇝ *string* / integer
Attribute value.

Result

If the parameters are correct `get_window_attr` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`, `set_draw`, `set_color`, `set_colored`, `set_line_width`

See also

`open_window`, `set_window_attr`

Module

Foundation

`get_window_extents` `( : : WindowHandle : Row, Column, Width, Height )`

Information about a window's size and position.

get_window_extents returns the position of the upper left corner, as well as width and height of the output window.

Attention

Size and position of a window may be modified by the window manager, without explicit instruction in the program. Therefore the values which are returned by get_window_extents may change cause of side effects.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Row** (output_control) ..... rectangle.origin.y ⇝ *integer*
  Row index of upper left corner of the window.
- ▷ **Column** (output_control) ..... rectangle.origin.x ⇝ *integer*
  Column index of upper left corner of the window.
- ▷ **Width** (output_control) ..... rectangle.extent.x ⇝ *integer*
  Window width.
- ▷ **Height** (output_control) ..... rectangle.extent.y ⇝ *integer*
  Window height.

Example

```
open_window(100,100,200,200,'root','visible','',WindowHandle)
fwrite_string(FileHandle, 'Move the window with the mouse!')
fnew_line(FileHandle)
repeat
get_mbutton(WindowHandle,_,_,Button)
get_window_extents(WindowHandle,Row,Column,Width,Height)
fwrite_string(FileHandle, ['(',Row,',',Column,')'])
fnew_line(FileHandle)
until(Button == 4)
```

Result

If the window is valid get_window_extents returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_draw, set_color, set_colored, set_line_width

See also

set_window_extents, open_window

Module

Foundation

get_window_pointer3 (: : WindowHandle : ImageRed, ImageGreen, ImageBlue, Width, Height)

Access to a window's pixel data.

This operator does not work in an HDevelop graphics window opened with dev_open_window.

get_window_pointer3 enables (in some window systems) the direct access to the bitmap. Result values are the three pointers on the color extracts of a 24-bit window (ImageRed, ImageGreen, ImageBlue), as well as the window size (Width, Height). In the language C the type of the image points is unsigned char.

Attention

get_window_pointer3 is usable only for window type *'pixmap'*.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.
▷ **ImageRed** (output_control) . . . . . integer ⇝ *integer*
Pointer on red channel of pixel data.
▷ **ImageGreen** (output_control) . . . . . integer ⇝ *integer*
Pointer on green channel of pixel data.
▷ **ImageBlue** (output_control) . . . . . integer ⇝ *integer*
Pointer on blue channel of pixel data.
▷ **Width** (output_control) . . . . . extent.x ⇝ *integer*
Length of an image line.
▷ **Height** (output_control) . . . . . extent.y ⇝ *integer*
Number of image lines.

Result

If a window of type *'pixmap'* exists and it is valid `get_window_pointer3` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

See also

`open_window`, `set_window_type`

Module

Foundation

get_window_type (: : WindowHandle : WindowType)

Get the window type.

`get_window_type` determines the type or the graphical software, respectively, of the output device for the window. You may query the available types of output devices with the operator `query_window_type`. A reasonable use for `get_window_type` might be in the field of the development of machine independent software. Possible values are:

'X-Window' X-Window Version 11.

'WIN32-Window' Microsoft Windows.

'pixmap' Windows are not shown, but managed in memory. By this means HALCON programs can be ported on computers without a graphical display. Note that windows of this type support only HALCON objects (images, regions and XLDs). Fonts, for example, are not available.

'PostScript' Objects are output to a PostScript File.

'default' Current window type.

'system_default' Default window type for current platform.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle* / string
Window handle.
Suggested values: `WindowHandle` ∈ {'default', 'system_default'}
▷ **WindowType** (output_control) . . . . . string ⇝ *string*
Window type

Example

```
open_window(100,100,200,200,'root','visible','',WindowHandle)
get_window_type(WindowHandle,WindowType)
fwrite_string(FileHandle, ['Window type: ',WindowType])
fnew_line(FileHandle)
```

Result

If the window is valid `get_window_type` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

See also

`query_window_type`, `set_window_type`, `get_window_pointer3`, `open_window`

Module

Foundation

new_extern_window (: : WINHWnd, Row, Column, Width, Height : WindowHandle)

Create a virtual graphics window under Windows.

`new_extern_window` opens a new virtual window. Note that this only works on Windows systems and drawing objects are not supported.

Virtual means that a new window will not be created, but the window whose window handle is given in the parameter `WINHWnd` is used to perform output of gray value data, regions, graphics as well as to perform textual output.

All operators that allow the user to define graphical primitives through mouse interactions, like for example `draw_rectangle1`, do not work on external windows. The following operators can be used:

- Output of gray values: `set_paint`, `set_lut`
- Regions: `set_color`, `set_rgb`, `set_hsi`, `set_gray`, `set_shape`, `set_line_width`, `set_insert`, `set_line_style`, `set_draw`
- Image part: `set_part`
- Text: `set_font`

You may query current set values by calling operators like `get_shape`. As some parameters are specified through the hardware (Resolution/Colors), you may query current available resources by calling operators like `query_color`.

The parameter `WINHWnd` is used to pass the window handle of the window in which output should be done.

The origin of the coordinate system of the window resides in the upper left corner (coordinates: (0,0)). The row index grows downward (maximum: `Height`-1), the column index grows to the right (maximal: `Width`-1).

You may use the value *-1* for parameters `Width` and `Height`. This means, that the corresponding value is chosen automatically. In particular, this is important if the aspect ratio of the pixels is not 1.0 (see `set_system`). If one of the two parameters is set to *-1*, it will be chosen through the size which results out of the aspect ratio of the pixels. If both parameters are set to *-1*, they will be set to the current image format.

The position and size of a window may change during runtime of a program, e.g., through external influences (window manager). With the operator `set_window_extents`, you can change the size of the (external) widow

via the program. Note that set_window_extents offers to change the position as well, but you cannot change the position of an external window under Windows.

Opening a window causes the assignment of a default font. It is used in connection with operators like write_string and you may change it by performing set_font after calling open_window. On the other hand, you have the possibility to specify a default font by calling set_system(::'default_font',<Fontname>:) before opening a window (and all following windows; see also query_font).

You may set the color of graphics and font, which is used for output operators like disp_region or disp_circle, by calling set_rgb, set_hsi or set_gray. Calling set_insert specifies how graphics is combined with the content of the image repeat memory. Thereto you may achieve by calling, e.g., set_insert(::'not':) to eliminate the font after writing text twice at the same position.

The content of the window is not saved, if other windows overlap the window. This must be done in the program code that handles the window in the calling program.

For graphical output (disp_image,disp_region, etc.) you may adjust the window by calling the operator set_part in order to represent a logical clipping of the image format. In particular this implies that only this part (appropriately scaled) of images and regions is displayed. Before you close your window, you have to close the HALCON-window.

Attention

Note that parameters as Row, Column, Width and Height are constrained through the output device, i.e., the size of the Windows NT desktop. Furthermore, be aware that all operators that allow the user to define graphical primitives through mouse interactions, like for example draw_rectangle1, do not work on external windows.

Parameters

- ▷ **WINHWnd** (input_control) ..... pointer ⇝ *integer*
  Windows window handle of a previously created window.
  Restriction: WINHWnd != 0
- ▷ **Row** (input_control) ..... rectangle.origin.y ⇝ *integer*
  Row coordinate of upper left corner.
  Default: 0
  Restriction: Row >= 0
- ▷ **Column** (input_control) ..... rectangle.origin.x ⇝ *integer*
  Column coordinate of upper left corner.
  Default: 0
  Restriction: Column >= 0
- ▷ **Width** (input_control) ..... rectangle.extent.x ⇝ *integer*
  Width of the window.
  Default: 512
  Restriction: Width > 0 || Width == -1
- ▷ **Height** (input_control) ..... rectangle.extent.y ⇝ *integer*
  Height of the window.
  Default: 512
  Restriction: Height > 0 || Height == -1
- ▷ **WindowHandle** (output_control) ..... window ⇝ *handle*
  Window handle.

Example

```
// Needs to be embedded into an application framework,
// see examples/cpp/mfc/MatchingExtWin.
```

Result

If the values of the specified parameters are correct new_extern_window returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

reset_obj_db

Possible Successors

set_color, query_window_type, get_window_type, set_window_type, get_mposition, set_tposition, set_tshape, set_window_extents, get_window_extents, query_color, set_check, set_system

Alternatives

open_window

See also

open_window, disp_region, disp_image, disp_color, set_lut, query_color, set_color, set_rgb, set_hsi, set_pixel, set_gray, set_part, set_part_style, query_window_type, get_window_type, set_window_type, get_mposition, set_tposition, set_window_extents, get_window_extents, set_window_attr, set_check, set_system

Module

Foundation

open_window (: : Row, Column, Width, Height, FatherWindow, Mode, Machine : WindowHandle)

Open a graphics window.

open_window opens a new window, which can be used to perform output of gray value data, regions, graphics as well as to perform textual output. All output (disp_region, disp_image, etc.) is redirected to this window, if the same logical window number WindowHandle is used.

The background of the created window is set to black in advance and it has a white border, which is 2 pixels wide (see also set_window_attr(::'border_width',<Width>:)).

Certain parameters used for the editing of output data are assigned to a window. These parameters are considered during the output itself (e.g., with disp_image or disp_region). They are not specified by an output operator, but by "configuration operators". If you want to set, e.g., the color red for the output of regions, you have to call set_color(::WindowHandle,'red':) before calling disp_region. These parameters are always set for the window with the logical window number WindowHandle and remain assigned to a window as long as they will be overwritten. You may use the following configuration operators:

- Output of gray values: set_paint, set_lut
- Regions: set_color, set_rgb, set_hsi, set_gray, set_shape, set_line_width, set_insert, set_line_style, set_draw
- Image clipping: set_part
- Text: set_font

You may query current set values by calling operators like get_shape. As some parameters are specified through the hardware (Resolution/Colors), you may query current available resources by calling query_color.

The origin of the coordinate system of the window resides in the upper left corner (coordinates: (0,0)). The row index grows downward (maximal: Height-1), the column index grows to the right (maximal: Width-1). You have to keep in mind, that the range of the coordinate system is independent of the window size. It is specified only through the image format (see reset_obj_db).

The parameter Machine indicates the name of the computer, which has to open the window. In case of a X-window, TCP-IP only sets the name, DEC-Net sets in addition a colon behind the name. The "server" resp. the "screen" are not specified. If the empty string is passed the environment variable DISPLAY is used. It indicates the target computer. At this the name is indicated in common syntax

```
<Host>:0.0
```

.

For windows of type *'WIN32-Window'* and *'X-Window'* the parameter FatherWindow can be used to determine the father window for the window to be opened. In case the control 'father' is set via set_check, FatherWindow must be the ID of a HALCON window, otherwise (set_check(::'~father':)) it can also be the ID of an operating system window. If FatherWindow is passed the value 0 or 'root', then under Windows and Unix-like systems the desktop and the root window become the father window, respectively. In this case, the value of the control 'father' (set via set_check) is irrelevant. The caller must ensure that FatherWindow is a valid handle and not destroyed as long as the embedded HALCON window is used.

You may use the value "-1" for parameters Width and Height. This means, that the according value has to be specified automatically. In particular this is of importance, if the proportion of pixels is not 1.0 (see set_system): Is one of the two parameters set to "-1", it will be specified through the size which results out of the proportion of pixels. Are both parameters set to "-1", they will be set to the maximum image format, which is currently used (further information about the currently used maximum image format can be found in the description of get_system using "width" or "height").

Position and size of a window may change during runtime of a program. This may be achieved by calling set_window_extents, but also through external interferences (window manager). In the latter case the operator set_window_extents is provided.

Opening a window causes the assignment of a called default font. It is used in connection with operators like write_string and you may overwrite it by performing set_font after calling open_window. On the other hand you have the possibility to specify a default font by calling set_system(::'default_font',<Fontname>:) before opening a window (and all following windows; see also query_font).

You may set the color of graphics and font, which is used for output operators like disp_region or disp_circle, by calling set_rgb, set_hsi or set_gray. Calling set_insert specifies how graphics is combined with the content of the image repeat memory. Thereto you may achieve by calling, e.g., set_insert(::'not':) to eliminate the font after writing text twice at the same position.

Normally every output (e.g., disp_image, disp_region, disp_circle, etc.) in a window is terminated by a called "flush". This causes the data to be fully visible on the display after termination of the output operator. But this is not necessary in all cases, in particular if there are permanently output tasks or if there is a mouse procedure active. Therefore it is more favorable (i.e., more rapid) to store the data until sufficient data is available. You may stop this behavior by calling set_system(::'flush_graphic','false':).

The content of windows is saved (in case it is supported by special driver software); i.e., it is preserved, also if the window is hidden by other windows. But this is not necessary in all cases: If the content of a window is built up permanently new (copy_rectangle), you may suppress the security mechanism for that and hence you can save the necessary memory. This is done by calling set_system(::'backing_store','false':) before opening a window. In doing so you save not only memory but also time to compute. This is significant for the output of video clips (see copy_rectangle).

For graphical output (disp_image,disp_region, etc.) you may adjust the window by calling the operator set_part in order to represent a logical clipping of the image format. In particular this implicates that you obtain this clipping (with appropriate enlargement) of images and regions only.

Difference: graphical window - textual window

- Using graphical windows the layout is not as variable as concerned to textual windows.
- You may use textual windows for the input of user data only (read_string).
- During the output of images, regions and graphics a "zooming" is performed using graphical windows: Independent on size and side ratio of the window images are transformed in that way, that they are displayed in the window by filling it completely. On the opposite side using textual windows the output does not care about the size of the window (only if clipping is necessary).
- Using graphical windows the coordinate system of the window corresponds to the coordinate system of the image format. Using textual windows, its coordinate system is always equal to the display coordinates independent on image size.

The parameter Mode determines the mode of the window. It may have following values:

'visible': Normal mode for graphical windows: The window is created according to the parameters and all input and output are possible.

'invisible': Invisible windows are not displayed in the display. Parameters like Row, Column and FatherWindow do not have any meaning. Output to these windows has no effect. Input (read_string, mouse, etc.) is not possible. You may use these windows to query representation parameter for an output device without opening a (visible) window. Common queries are, e.g., query_color and get_string_extents.

'transparent': These windows are transparent: the window itself is not visible (edge and background), but all the other operations are possible and all output is displayed. A common use for this mode is the creation of mouse sensitive regions.

'buffer': These are also not visible windows. The output of images, regions and graphics is not visible on the display, but is stored in memory. Parameters like Row, Column and FatherWindow do not have any meaning. You may use buffer windows, if you prepare output (in the background) and copy it finally with copy_rectangle in a visible window. Another usage might be the rapid processing of image regions during interactive manipulations. Textual input and mouse interaction are not possible in this mode.

Attention

You may keep in mind that parameters as Row, Column, Width and Height are constrained by the output device. If you specify a father window (FatherWindow <> 'root') the coordinates are relative to this window.

Parameters

▷ **Row** (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Row index of upper left corner.
Default: 0
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column index of upper left corner.
Default: 0
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Width** (input_control) . . . . . rectangle.extent.x ⇝ *integer*
Width of the window.
Default: 256
(lin)
Minimum increment: 1
Recommended increment: 1
Restriction: 0 <= Width <= 32768 || Width == -1

▷ **Height** (input_control) . . . . . rectangle.extent.y ⇝ *integer*
Height of the window.
Default: 256
(lin)
Minimum increment: 1
Recommended increment: 1
Restriction: 0 <= Height <= 32768 || Height == -1

▷ **FatherWindow** (input_control) . . . . . pointer ⇝ *integer* / string
Logical number of the father window. To specify the display as father you may enter 'root' or 0.
Default: 0
Restriction: FatherWindow >= 0

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Window mode.
Default: 'visible'
List of values: Mode ∈ {'visible', 'invisible', 'transparent', 'buffer'}

▷ **Machine** (input_control) . . . . . string ⇝ *string*
Name of the computer on which you want to open the window. Otherwise the empty string.
Default: ''

▷ **WindowHandle** (output_control) . . . . . window ⇝ *handle*
Window handle.

Example

```
open_window(0,0,400,-1,'root','visible','',WindowHandle)
read_image(Image,'fabrik')
disp_image(Image,WindowHandle)
write_string(WindowHandle,'File, fabrik')
new_line(WindowHandle)
get_mbutton(WindowHandle,_,_,_)
set_lut(WindowHandle,'temperature')
set_color(WindowHandle,'blue')
write_string(WindowHandle,'temperature')
new_line(WindowHandle)
write_string(WindowHandle,'Draw Rectangle')
new_line(WindowHandle)
draw_rectangle1(WindowHandle,Row1,Column1,Row2,Column2)
set_part(WindowHandle,Row1,Column1,Row2,Column2)
disp_image(Image,WindowHandle)
new_line(WindowHandle)
```

Result

If the values of the specified parameters are correct `open_window` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`reset_obj_db`

Possible Successors

`set_color`, `query_window_type`, `get_window_type`, `set_window_type`, `get_mposition`, `set_tposition`, `set_tshape`, `set_window_extents`, `get_window_extents`, `query_color`, `set_check`, `set_system`

See also

`disp_region`, `disp_image`, `disp_color`, `set_lut`, `query_color`, `set_color`, `set_rgb`, `set_hsi`, `set_pixel`, `set_gray`, `set_part`, `set_part_style`, `query_window_type`, `get_window_type`, `set_window_type`, `get_mposition`, `set_tposition`, `set_window_extents`, `get_window_extents`, `set_window_attr`, `set_check`, `set_system`

Module

Foundation

query_window_type (: : : WindowTypes)

Query all available window types.

`query_window_type` returns a tuple which contains all devices or software systems, respectively, which are used to display image objects. You may use `query_window_type` usefully while developing machine independent programs. Possible values are:

'X-Window' X-Window Version 11.

'WIN32-Window' Microsoft Windows.

'pixmap' Windows are not displayed, but managed in memory. In this manner it is possible to port HALCON programs to computers without graphical display.

'PostScript' Objects are output to a PostScript File.

Parameters

▷ **WindowTypes** (output_control) . . . . . string-array ⇝ *string*
Names of available window types.

Result

`query_window_type` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`reset_obj_db`

Module

Foundation

set_window_attr (: : AttributeName, AttributeValue :)

Set window characteristics.

You may use `set_window_attr` to set specific characteristics of graphics windows. With it you may modify the following default parameters of a window:

'border_width' Width of the window border in pixels. Is not implemented under Windows.

'border_color' Color of the window border. Is not implemented under Windows.

'background_color' Background color of the window.

'window_title' Name of the window in the title bar.

Attention

You have to call `set_window_attr` *before* calling `open_window`.

Parameters

▷ **AttributeName** (input_control) . . . . . string ⇝ *string*
Name of the attribute that should be modified.
List of values: `AttributeName` ∈ {'border_width', 'border_color', 'background_color', 'window_title'}

▷ **AttributeValue** (input_control) . . . . . string ⇝ *string* / integer
Value of the attribute that should be set.
List of values: `AttributeValue` ∈ {0, 1, 2, 'white', 'black', 'MyName', 'default'}

Result

If the parameters are correct `set_window_attr` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_draw, set_color, set_colored, set_line_width

See also

open_window, get_window_attr, set_window_param

Module

Foundation

set_window_dc (: : WindowHandle, WINHDC :)

Set the device context of a virtual graphics window (Windows NT).

set_window_dc sets the device context of a window previously opened with new_extern_window. All output (disp_region, disp_image, etc.) is done in the window with this device context.

The parameter WINHDC contains the device context of the window in which HALCON should output its data. This device context is used in all output routines of HALCON.

Attention

The window WindowHandle has to be created with new_extern_window beforehand.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **WINHDC** (input_control) ..... pointer ⇝ *integer*
device context of WINHWnd.
Restriction: WINHDC != 0

Result

If the values of the specified parameters are correct, set_window_dc returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

new_extern_window

Possible Successors

disp_image, disp_region

See also

new_extern_window, disp_region, disp_image, disp_color, set_lut, query_color, set_color, set_rgb, set_hsi, set_pixel, set_gray, set_part, set_part_style, query_window_type, get_window_type, set_window_type, get_mposition, set_tposition, set_window_extents, get_window_extents, set_window_attr, set_check, set_system

Module

Foundation

set_window_extents (: : WindowHandle, Row, Column, Width, Height :)

Modify position and size of a window.

set_window_extents positions the upper left corner of the output window at (Row,Column) and changes the size of the window to Width and Height at the same time. Negative values for Width and Height are ignored.

Adapting the size of the window to the size of the image part to be displayed will prevent slowing down the display due to necessary interpolations. Thus, the window preferably has the same size as the image part to be displayed, or otherwise half its size, quarter its size, etc.

Attention

Modifying the size of the window does not automatically redraw the window contents. This has to be done by the program by redisplaying the desired data.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row** (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Row index of upper left corner in target position.
Default: 0
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column index of upper left corner in target position.
Default: 0
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Width** (input_control) . . . . . rectangle.extent.x ⇝ *integer*
Width of the window.
Default: 512
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Height** (input_control) . . . . . rectangle.extent.y ⇝ *integer*
Height of the window.
Default: 512
(lin)
Minimum increment: 1
Recommended increment: 1

Result

If the window is valid and the parameters are correct `set_window_extents` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

See also

`get_window_extents`, `open_window`

Module

Foundation

set_window_type (: : WindowType :)

Specify a window type.

set_window_type determines on which type of output device the output is going to be displayed. This specification is going to be used by the operator open_window while opening the windows. You may open different windows on different types of output devices. Therefore you have to specify the wanted type before opening. You may request the available types of output devices by calling the operator query_window_type. Possible values are:

'X-Window' X-Window Version 11.

'WIN32-Window' Microsoft Windows.

'pixmap' Windows are not displayed, but managed in memory only. In this manner you may port HALCON programs to computers without graphical display.

'PostScript' Objects are output to a PostScript File.

'system_default' Default for current platform.

A useful usage of set_window_type could be the development of machine independent programs.

Parameters

▷ **WindowType** (input_control) . . . . . string ⇝ *string*
Name of the window type which has to be set.
Default: 'X-Window'
List of values: WindowType ∈ {'X-Window', 'WIN32-Window', 'pixmap', 'PostScript', 'system_default'}

Result

If the type of the output device is available, then set_window_type returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

See also

open_window, query_window_type, get_window_type

Module

Foundation

unproject_coordinates (Image : : WindowHandle, Row, Column : ImageRow, ImageColumn, Height)

Calculates image coordinates for a point in a 3D plot window.

If Image is displayed in WindowHandle using a 3D plot (i.e., *'3d_plot'* was set using set_paint), unproject_coordinates calculates the image coordinates ImageRow, ImageColumn, and the Height for a given point Row,Column in window coordinates. The window coordinates Row, Column must be provided in respect to the current image part. As a consequence, this coordinates are subpixel coordinates. Given the current image part (row1,column1,row2, column2), the upper left corner corresponds to the coordinates (row1 - 0.5, col1 - 0.5). Accordingly, the bottom right corner corresponds to the coordinates (row2 - 0.5, col2 - 0.5). Use get_mposition_sub_pix or get_mbutton_sub_pix to obtain these coordinates directly.

One of the window parameters *'save_depth_buffer'* or *'interactive_plot'* must be set to *'true'* using set_window_param. Otherwise, unproject_coordinates cannot be used.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / vector_field
Displayed image.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row** (input_control) ..... integer ⇝ *real* / integer
Row coordinate in the window.

▷ **Column** (input_control) ..... integer ⇝ *real* / integer
Column coordinate in the window.

▷ **ImageRow** (output_control) ..... integer ⇝ *integer*
Row coordinate in the image.

▷ **ImageColumn** (output_control) ..... integer ⇝ *integer*
Column coordinate in the image.

▷ **Height** (output_control) ..... real ⇝ *integer* / real
Height value.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_mbutton_sub_pix`, `get_mposition_sub_pix`, `disp_image`

See also

`disp_obj`, `update_window_pose`, `set_paint`

Module

Foundation

update_window_pose (: : WindowHandle, LastRow, LastCol, CurrentRow, CurrentCol, Mode :)

Modify the pose of a 3D plot.

The operator `update_window_pose` provides an easy way to modify the pose of the height field displayed by `disp_obj` if the paint mode is set to *'3d_plot'* using `set_paint`.

Two window coordinates (`LastRow`,`LastCol`) and (`CurrentRow`, `CurrentCol`) are transformed into rotation, scaling, or movement depending on `Mode`. This window coordinates must be provided with respect to the current image part. As a consequence, this coordinates are subpixel coordinates. Given the current image part (row1,column1,row2,column2), the upper left corner corresponds to the coordinate (row1 - 0.5, col1 - 0.5). Accordingly, the bottom right corner corresponds to the coordinate (row2 - 0.5, col2 - 0.5). Use `get_mposition_sub_pix` or `get_mbutton_sub_pix` to obtain this coordinates directly.

If `Mode` is set to *'rotate'*, the height field is rotated using a virtual trackball model. Both points are projected on a sphere centered in the center of the window `WindowHandle`. The circular arc between this two projections corresponds to the rotation applied to the height field.

If `Mode` is set to *'scale'*, zooming is reduced if `CurrentRow` is greater than `LastRow` and increased if `CurrentRow` is smaller than `LastRow`.

If `Mode` is set to *'move'*, the input points are projected onto the plane through the center of the height field parallel to the viewing plane. The center of the height field is moved in this plane by the distance between both projections.

If `Mode` is set to *'move_plane'*, the input points are projected on the plane given by height zero. The height field is moved in this plane by the distance between both projections.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **LastRow** (input_control) ..... point.y ⇝ *real* / integer
Row coordinate of the first point.

▷ **LastCol** (input_control) ..... point.x ⇝ *real* / integer
Column coordinate of the first point.

▷ **CurrentRow** (input_control) ..... point.y ⇝ *real* / integer
Row coordinate of the second point.

▷ **CurrentCol** (input_control) ..... point.x ⇝ *real* / integer
Column coordinate of the second point.

▷ **Mode** (input_control) ..... string ⇝ *string*
Navigation mode.
Default: 'rotate'
List of values: Mode ∈ {'rotate', 'scale', 'move', 'move_plane'}

Example

```
* Interactive display of a height field
dev_set_paint ('3d_plot')
while (1)
  dev_set_check ('~give_error')
  get_mposition_sub_pix (WindowHandle, Row, Column, Button)
  dev_set_check ('give_error')
  if (ButtonDown and (Button == 0))
    ButtonDown := false
  endif
  if (not(Button == 0))
    if (ButtonDown)
      if (Button == 1)
        mode := 'rotate'
      endif
      if (Button == 4)
        mode := 'scale'
      endif
      if (Button == 5)
        mode := 'move'
      endif
      update_window_pose (WindowHandle, lastRow, lastCol, Row, Column, mode)
    else
      if (Button == 2)
        break
      endif
      ButtonDown := true
    endif
    lastCol := Column
    lastRow := Row
  endif
  dev_display (Image)
endwhile
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_paint, open_window, get_mposition_sub_pix, get_mbutton_sub_pix

Possible Successors

disp_image

See also

unproject_coordinates

Module

Foundation

Chapter 14

Identification

14.1 Bar Code

This chapter provides an introduction to bar codes and the most important terms used when finding and reading bar codes in HALCON.

Structure of linear (1D) bar codes

A bar code consists of several dark bars and bright spaces. For every bar code there is a smallest possible element, the narrowest possible bar or the narrowest possible space, which is called module. In general, the width of the bars and spaces is different, but always a multiple of the module. Several bars and spaces together form a pattern. This pattern encodes a symbol defined in a decoding table. The conventions how such a binary pattern encodes a symbol and the corresponding decoding table lead to the respective bar code type. In addition to the patterns encoding the symbols, there are also patterns with special tasks. Depending on the type there are further patterns which can or must be part of the bar code:

- Quiet-zone: Zone in which no edges may occur. This zone is required before the start character and after the stop character to find the code.

Schematic representation of the quiet zone (orange) by means of an example bar code of type Code 128.

- Start and stop pattern: Define start, stop and direction. These patterns make it possible to read and decode the code from left to right as well as from right to left (and thus upside down).

Schematic representation of the start and stop pattern (orange) by means of an example bar code of type Code 128.

- Checksum: A digit which serves as a security check to ensure the code has been read correctly.

Schematic representation of the checksum (orange) by means of an example bar code of type Code 128.

- Guard pattern: Specific pattern, which can occur on the left, right, or in the center as additional symbol.

Schematic representation of a guard pattern (orange) by means of an example bar code of type UPC-A.

- Finder patterns: Pattern used to locate the symbol.

Schematic representation of a finder pattern (orange) by means of an example bar code of type GS1 Databar.

- Add-on symbol: Additional symbols for encoding of supplementary information. These symbols are encoded as bar codes with restricted length.

Schematic representation of an add-on symbol (orange) by means of an example bar code of type EAN-8 Add-On 2.

Structure of a composite bar code

Composite bar codes consist of the following elements:

- Linear bar code: As described above.
- 2D Code: For more information about 2D codes we refer to the "`Solution Guide II-C - 2D Data Codes`".
- Linking pattern: Combines the two codes mentioned before.

The following figure shows a schematic representation of a composite bar code.

Schematic representation of a composite bar code by means of an example bar code of type UPC-A Composite: Linear bar code (black), 2D code (light blue), and the linking pattern (orange).

Reading of a bar code

To be able to read a bar code in an image, it must be found first. The regions in which HALCON assumes a bar code are called candidates.

To read a candidate, different lines are laid through the region, called scanlines. Although theoretically a single scanline would be sufficient, one tries to decode the bar code along all scanlines. This increases the chance of decoding the code correctly. In addition, several successfully decoded scanlines allow additional plausibility checks, i.e., reading several scanlines equally decreases the chance of false positives.

Schematic representation of different scanlines of a candidate. Certain scanlines were read successfully (green), while the reading failed for others (red).

The workflow and various examples showing how to read bar codes in HALCON can be found in the "'Solution Guide I - Basics.

clear_bar_code_model (: : BarCodeHandle :)

Delete a bar code model and free the allocated memory

The operator clear_bar_code_model deletes a bar code model that was created by create_bar_code_model. All memory used by the model is freed. The handle of the model is passed in BarCodeHandle, which is invalid after the operator call.

Parameters

▷ **BarCodeHandle** (input_control) ..... barcode(-array) ⇝ *handle*
Handle of the bar code model.

Result

The operator clear_bar_code_model returns the value 2 (H_MSG_TRUE) if a valid handle was passed and the referred bar code model can be freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- BarCodeHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

find_bar_code

Module

Bar Code

create_bar_code_model (: : GenParamName,
GenParamValue : BarCodeHandle)

Create a model of a bar code reader.

The operator create_bar_code_model creates a generic model for reading all types of supported bar code symbols. The result of this operator is a handle to the bar code model (BarCodeHandle), which is used for all further operations on the bar code, like modifying the model, reading a symbol, or accessing the results of the symbol search.

In general, bar codes will be found and decoded without any additional adjustment of the parameters. Therefore, GenParamName and GenParamValue are empty tuples by default. In the case of poor image quality or abnormal geometric characteristics of the bar code, which requires special parameter settings for a successful decoding of the bar code symbols, parameters can be adjusted already while creating the bar code model. Alternatively, parameters can be changed later on as well by applying the operator set_bar_code_param

or set_bar_code_param_specific. For a detailed description of the available model parameters see set_bar_code_param.

Parameters

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that can be adjusted for the bar code model.
Default: []
List of values: GenParamName ∈ {'check_char', 'composite_code', 'barcode_height_min', 'barcode_width_min', 'barcode_width_max', 'element_size_max', 'element_size_min', 'meas_thresh', 'meas_thresh_abs', 'min_identical_scanlines', 'num_scanlines', 'orientation', 'orientation_tol', 'persistence', 'start_stop_tolerance', 'stop_after_result_num', 'upce_encodation', 'upce1_enable', 'timeout', 'train', 'quiet_zone', 'element_size_variable', 'min_code_length', 'max_code_length'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
Values of the generic parameters that can be adjusted for the bar code model.
Default: []
Suggested values: GenParamValue ∈ {0, 0.1, 1, 1.5, 2, 8, 32, 45, 'present', 'absent', 'none', 'CC-A/B', 'auto', 'high', 'low', 'true', 'false'}

▷ **BarCodeHandle** (output_control) . . . . . barcode ⇝ *handle*
Handle for using and accessing the bar code model.

Result

The operator create_bar_code_model returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

find_bar_code

See also

clear_bar_code_model

References

International Standard ISO/IEC 15420: "Information technology - Automatic identification and data capture techniques - Bar code symbology specification - EAN/UPC"; Reference number ISO/IEC 15420:2000 (E); ISO/IEC 2000.

International Standard ISO/IEC 16390: "Information technology - Automatic identification and data capture techniques - Bar code symbology specification - Interleaved 2 of 5"; Reference number ISO/IEC 16390:1999 (E); ISO/IEC 1999.

International Standard ISO/IEC 16388: "Information technology - Automatic identification and data capture techniques - Bar code symbology specification - Code 39"; Reference number ISO/IEC 16388:1999 (E); ISO/IEC 1999.

American National Standards Institute, Inc.(ANSI): "Uniform Symbology Specification Code 93"; ANSI/AIM-BC5-2000; AIM 2000.

International Standard ISO/IEC 15417: "Information technology - Automatic identification and data capture techniques - Bar code symbology specification - Code 128"; Reference number ISO/IEC 15417:2000 (E); ISO/IEC 2000.

American National Standards Institute, Inc.(ANSI): "Uniform Symbology Specification Codabar"; ANSI/AIM-BC3-2000; AIM 2000.

International Standard ISO/IEC 24724: "Information technology - Automatic identification and data capture techniques - Reduced Space Symbology (RSS) bar code symbology specification"; Reference number ISO/IEC 24724: 2006 (E); ISO/IEC 2006.

International Standard ISO/IEC 24723: “Information technology - Automatic identification and data capture techniques - EAN.UCC Composite bar code symbology specification”; Reference number ISO/IEC 24723:2006 (E); ISO/IEC 2006.

Module

Bar Code

decode_bar_code_rectangle2 (Image : : BarCodeHandle, CodeType, Row, Column, Phi, Length1, Length2 : DecodedDataStrings)

Decode bar code symbols within a rectangle.

The operator decode_bar_code_rectangle2 uses the bar code model specified by BarCodeHandle to decode a bar code at a given position within the image Image. BarCodeHandle must be created with create_bar_code_model, its parameters can be set with set_bar_code_param. The position of the bar code is given as an arbitrarily oriented rectangle. Contrary to find_bar_code, where the decoding is preceded by a time consuming search for candidate regions, decode_bar_code_rectangle2 scans the provided region directly for bar codes. The rectangular region is defined by the parameters Row and Column for the center, Phi for the orientation and Length1 and Length2 for the half edge lengths (see gen_rectangle2). The angle Phi also determines the reading direction and is defined as the angle between the reading direction of the bar code and the horizontal image axis. The angle is positive in counter clockwise direction and is given in radians as in gen_rectangle2. Note that the angle unit deviates from the conventions in get_bar_code_result and set_bar_code_param where angles are given in degrees. Phi can be in the range of $[-\pi, +\pi]$. The reading direction is perpendicular to the bars of the bar code. Bar codes with a reading direction Phi $+ \pi$ are also returned. Multiple regions for decoding can be given by supplying tuples for Row, Column, Phi, Length1 and Length2.

The rectangle should cover completely the bar code and the quiet zones. Regions that are too big will be decoded, if there is no disturbing pattern within these regions and the height (Length2) is small enough such that scanlines can be placed sufficiently dense. Rectangles that are too short in reading direction (Length1) cannot be decoded.

decode_bar_code_rectangle2 can be used if the position of the bar code is already known in advance. For example, the candidate regions found by find_bar_code and get_bar_code_object could be reused with a different code type (see the following example).

Multiple bar code types can be specified for CodeType. See section Autodiscrimination for find_bar_code.

Further aspects of the actual decoding are explained with the operator find_bar_code.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte
Input image.

▷ **BarCodeHandle** (input_control) ..... barcode ⇝ *handle*
Handle of the bar code model.

▷ **CodeType** (input_control) ..... string(-array) ⇝ *string*
Type of the searched bar code.
Default: ’EAN-13’
List of values: CodeType ∈ {’2/5 Industrial’, ’2/5 Interleaved’, ’Codabar’, ’Code 39’, ’Code 93’, ’Code 128’, ’EAN-13’, ’EAN-13 Add-On 2’, ’EAN-13 Add-On 5’, ’EAN-8’, ’EAN-8 Add-On 2’, ’EAN-8 Add-On 5’, ’UPC-A’, ’UPC-A Add-On 2’, ’UPC-A Add-On 5’, ’UPC-E’, ’UPC-E Add-On 2’, ’UPC-E Add-On 5’, ’MSI’, ’PharmaCode’, ’GS1 DataBar Omnidir’, ’GS1 DataBar Truncated’, ’GS1 DataBar Stacked’, ’GS1 DataBar Stacked Omnidir’, ’GS1 DataBar Limited’, ’GS1 DataBar Expanded’, ’GS1 DataBar Expanded Stacked’, ’GS1-128’, ’auto’}

▷ **Row** (input_control) ..... rectangle2.center.y(-array) ⇝ *real* / integer
Row index of the center.
Default: 50.0

▷ **Column** (input_control) ..... rectangle2.center.x(-array) ⇝ *real* / integer
Column index of the center.
Default: 100.0

▷ **Phi** (input_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real* / integer
Orientation of rectangle in radians.
Default: 0.0
Suggested values: Phi ∈ {0.0, 0.785398, 1.570796, 3.1415926}

▷ **Length1** (input_control) . . . . . rectangle2.hwidth(-array) ⇝ *real* / integer
Half of the length of the rectangle along the reading direction of the bar code.
Default: 200.0

▷ **Length2** (input_control) . . . . . rectangle2.hheight(-array) ⇝ *real* / integer
Half of the length of the rectangle perpendicular to the reading direction of the bar code.
Default: 100.0

▷ **DecodedDataStrings** (output_control) . . . . . string(-array) ⇝ *string*
Data strings of all successfully decoded bar codes.

Example

```
read_image (Image, 'barcode/ean13/ean1301.png')
create_bar_code_model ([], [], BHandle)
find_bar_code (Image, SymReg, BHandle, '2/5 Industrial', Dec)
if (|Dec| == 0)
*     A 2/5 Industrial code wasn't found. Try decoding an EAN-13 code.
    get_bar_code_object (CandReg, BHandle, 'all', 'candidate_regions')
    smallest_rectangle2 (CandReg, R, C, Phi, L1, L2)
    decode_bar_code_rectangle2 (Image, BHandle, 'EAN-13', R, C, Phi, \
                                L1, L2, Dec)
endif
```

Result

The operator decode_bar_code_rectangle2 returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- BarCodeHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_bar_code_model, set_bar_code_param, smallest_rectangle2

Possible Successors

get_bar_code_result, get_bar_code_object, clear_bar_code_model

Alternatives

find_bar_code

See also

gen_rectangle2

Module

Bar Code

deserialize_bar_code_model (
: : SerializedItemHandle : BarCodeHandle)

Deserialize a bar code model.

deserialize_bar_code_model deserializes a bar code model that was serialized by serialize_bar_code_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized model is defined by the handle SerializedItemHandle. The deserialized values are stored in a new bar code model with the handle BarCodeHandle.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **BarCodeHandle** (output_control) . . . . . barcode ⇝ *handle*
Handle of the bar code model.

Result

The operator deserialize_bar_code_model returns the value 2 (H_MSG_TRUE) if the bar code can be correctly deserialized. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_bar_code_model

Possible Successors

find_bar_code

Alternatives

create_bar_code_model

See also

serialize_bar_code_model, clear_bar_code_model

Module

Bar Code

find_bar_code (Image : SymbolRegions : BarCodeHandle,
CodeType : DecodedDataStrings)

Detect and read bar code symbols in an image.

The operator find_bar_code finds and reads bar code symbols in a given image (Image) and returns the decoded data. In one image an arbitrary number of bar codes of a single type can be read. The type of the desired bar code symbology is given by CodeType. The decoded strings are returned in DecodedDataStrings and the corresponding bar code regions in SymbolRegions. For a total of n successfully read bar codes, the indices from 0 to (n-1) can be used as candidate handle in the operators get_bar_code_object and get_bar_code_result in order to retrieve the desired data of one specific bar code result.

Bar codes are expected to appear dark on a light background. To read light bar codes on a dark background, invert the input image using the operator invert_image beforehand.

Before calling find_bar_code a bar code model must be created by calling create_bar_code_model. This operator returns a bar code model BarCodeHandle, which is input to find_bar_code.

The output value DecodedDataStrings contains the decoded string of the symbol for each bar code result. The contents of the strings are conform to the appropriate standard of the symbology. Typically, DecodedDataStrings contains only data characters. For bar codes with a mandatory check character the check character is not included in the string. For bar codes with an optional check character, like for example Code 39, Codabar, 2/5 Industrial or 2/5 Interleaved, the result depends on the value of the *'check_char'* parameter, which can be set in create_bar_code_model, set_bar_code_param or set_bar_code_param_specific: The default setting of *'absent'* assumes that no check character is present. In this case, no check is performed and all characters are returned as data. When set to *'present'*, a check character is expected and used to verify the correctness of the bar code. The bar code is graded as unreadable if the check sum does not match. Accordingly, the symbol region and the decoded string do not appear in the

list of resulting strings (DecodedDataStrings) and in the list of resulting regions (SymbolRegions). The check character itself is stripped from the data. If this stripping is undesired, the mode *'preserved'* allows to verify the bar code while still keeping the check character in the data.

The underlying decoded reference data, including start/stop and check characters, can be queried by using the get_bar_code_result operator with the option *'decoded_reference'*.

Following bar code symbologies are supported:

2/5 Industrial	EAN-8	GS1-128
2/5 Interleaved	EAN-8 Add-On 2	GS1 DataBar Omnidirectional
Codabar	EAN-8 Add-On 5	GS1 DataBar Truncated
Code 39	EAN-13	GS1 DataBar Stacked
Code 32 (converted from Code 39)	EAN-13 Add-On 2	GS1 DataBar Stacked Omnidirectional
Code 93	EAN-13 Add-On 5	GS1 DataBar Limited
Code 128	UPC-A	GS1 DataBar Expanded
MSI	UPC-A Add-On 2	GS1 DataBar Expanded Stacked
PharmaCode	UPC-A Add-On 5	
	UPC-E	
	UPC-E Add-On 2	
	UPC-E Add-On 5	

Note that the PharmaCode can be read in forward and backward direction, both yielding a valid result. Therefore, both strings are returned and concatenated into a single string in DecodedDataStrings by a separating comma.

Note that the GS1 DataBar bar codes may contain an additional composite code component. To find and decode this component, set the parameter *'composite_code'* to *'CC-A/B'* using the operator set_bar_code_param.

Note that also a bar code of type 'Code 32' can be read by using find_bar_code with CodeType set to *'Code 39'* in combination with the external procedure convert_decoded_string_code39_to_code32.

The list of available GS1 Application Identifiers can be updated using set_system (see the parameter *'gs1_syntax_dictionary'*).

Autodiscrimination

Autodiscrimination describes the simultaneous decoding of multiple bar code types in one call of find_bar_code. For this purpose a tuple of bar code types is specified for the parameter CodeType. Using the generic value *'auto'* all known bar code types are decoded - except *'PharmaCode'* and *'MSI'* because these codes don't have enough features to be reliably separated from other bar code types. The tuple can also contain bar code types with the tilde prefix (~) which won't be decoded. For example

['auto', '~EAN-8', '~EAN-8 Add-On 2', '~EAN-8 Add-On 5']

describes all bar code types without *'PharmaCode'*, *'MSI'* and all kinds of *'EAN-8'*. Please note that each additionally allowed bar code type increases the run-time of the operator. Using too many bar code types the reliability of the decoding could decrease because not all bar code types can be discriminated reliably. To improve autodiscrimination compatibility bar codes with a check character or check sum should be used.

The bar code reader tries to decode the bar code types in the following order:

'GS1 DataBar Omnidir', *'GS1 DataBar Truncated'*, *'GS1 DataBar Limited'*, *'GS1 DataBar Expanded'*, *'GS1 DataBar Expanded Stacked'*, *'GS1 DataBar Stacked Omnidir'*, *'GS1 DataBar Stacked'*, *'GS1-128'*, *'Code 128'*, *'EAN-13 Add-On 5'*, *'EAN-13 Add-On 2'*, *'EAN-13'*, *'UPC-A Add-On 5'*, *'UPC-A Add-On 2'*, *'UPC-A'*, *'EAN-8 Add-On 5'*, *'EAN-8 Add-On 2'*, *'EAN-8'*, *'UPC-E Add-On 5'*, *'UPC-E Add-On 2'*, *'UPC-E'*, *'Code 93'*, *'Code 39'*, *'Codabar'*, *'2/5 Interleaved'*, *'2/5 Industrial'*

Therefore you should exclude at least all definitely not occurring bar code types that are scanned before the first of the bar code types you expect to find or, better, just scan for the explicit list of bar code types you expect.

Note that *'UPC-A'* codes have the same structure as *'EAN-13'* codes, except that they always start with a zero. If one wants to read *'UPC-A'*, *'UPC-A Add-On 2'*, or *'UPC-A Add-On 5'* codes explicitly, then the corresponding EAN-13 bar code type must be excluded (*'EAN-13'*, *'EAN-13 Add-On 2'*, or *'EAN-13 Add-On 5'*, respectively). Otherwise, the results will be returned as EAN codes because of the decoding order. For UPC-A codes with Add-On, *'EAN-13'* must be excluded in addition to the corresponding EAN-13 type with Add-On.

Especially for the autodiscrimination there is the operator set_bar_code_param_specific. With it some parameters of the bar code model can be set specifically for certain bar code types.

Training

If the bar code reader is in training mode, the operator find_bar_code executes a training cycle. The training mode is described with the operator set_bar_code_param.

Timeout and Abort

The operator find_bar_code can be aborted by a timeout and dynamically. With the operator set_bar_code_param you can specify a timeout. If find_bar_code reaches this timeout, it returns all codes decoded so far. Alternatively, you can call set_bar_code_param with *'abort'* from another thread to abort find_bar_code dynamically.

The information whether the operator was aborted or not can be queried by calling get_bar_code_result with the parameter *'aborted'*.

Furthermore, the operator find_bar_code can be canceled, which means no result is returned but instead an error is returned. This can be done with the operator set_operator_timeout or interrupt_operator. If find_bar_code is canceled by set_operator_timeout, H_ERR_TIMEOUT (9400) is returned. If find_bar_code is canceled by interrupt_operator, H_ERR_CANCEL (22) is returned. Note: Both mentioned operators are only supported in cancel mode.

Advice on low resolution bar code imaging systems

This advice applies to bar code applications that use low resolution images, i.e., images in which a single bar is between 0.6 and 2 pixels wide.

In these cases, one should optimize the bar code imaging system in order to eliminate noise and focal blur because a combination of these optical distortions lead to problems in the decodability of low resolution bar codes.

See set_bar_code_param documentation entry on *'small_elements_robustness'* for more advice.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object : byte*
  Input image. If the image has a reduced domain, the bar code search is reduced to that domain. This usually reduces the runtime of the operator. However, if the bar code is not fully inside the domain, the bar code cannot be decoded correctly.
- ▷ **SymbolRegions** (output_object) ..... region(-array) ⇝ *object*
  Regions of the successfully decoded bar code symbols.
- ▷ **BarCodeHandle** (input_control) ..... barcode ⇝ *handle*
  Handle of the bar code model.
- ▷ **CodeType** (input_control) ..... string(-array) ⇝ *string*
  Type of the searched bar code.
  Default: 'auto'
  List of values: CodeType ∈ {'auto', '2/5 Industrial', '2/5 Interleaved', 'Codabar', 'Code 39', 'Code 93', 'Code 128', 'EAN-13', 'EAN-13 Add-On 2', 'EAN-13 Add-On 5', 'EAN-8', 'EAN-8 Add-On 2', 'EAN-8 Add-On 5', 'UPC-A', 'UPC-A Add-On 2', 'UPC-A Add-On 5', 'UPC-E', 'UPC-E Add-On 2', 'UPC-E Add-On 5', 'MSI', 'PharmaCode', 'GS1 DataBar Omnidir', 'GS1 DataBar Truncated', 'GS1 DataBar Stacked', 'GS1 DataBar Stacked Omnidir', 'GS1 DataBar Limited', 'GS1 DataBar Expanded', 'GS1 DataBar Expanded Stacked', 'GS1-128'}
- ▷ **DecodedDataStrings** (output_control) ..... string(-array) ⇝ *string*
  Data strings of all successfully decoded bar codes.

Example

```
* Small example for aborting bar code search:
read_image (Image, 'barcode/ean13/tea_box_10.png')

* Decode normally:
create_bar_code_model ([], [], BarCodeHandle)
find_bar_code (Image, SymbolRegions, BarCodeHandle, 'auto', \
    DecodedDataStrings)
stop ()
```

```
* Start bar code search in separate thread.
par_start<ThreadID>: find_bar_code (Image, SymbolRegions1, BarCodeHandle, \
    'auto', DecodedDataStrings1)

* Let some time pass.
wait_seconds (0.002)

* Abort the data code search.
set_bar_code_param (BarCodeHandle, 'abort', 'true')

* Wait for the thread, it should finish very fast.
par_join(ThreadID)

* We can determine that the search in fact was aborted.
get_bar_code_result (BarCodeHandle, 'all', 'aborted', Aborted)
```

Result

The operator `find_bar_code` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- BarCodeHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_bar_code_model`, `set_bar_code_param`

Possible Successors

`get_bar_code_result`, `get_bar_code_object`, `clear_bar_code_model`

Alternatives

`decode_bar_code_rectangle2`

Module

Bar Code

get_bar_code_object (: BarCodeObjects : BarCodeHandle, CandidateHandle, ObjectName :)

Access iconic objects that were created during the search or decoding of bar code symbols.

With the operator `get_bar_code_object`, iconic objects created during the last call of the operator `find_bar_code` can be accessed. Besides the name of the object (`ObjectName`), the bar code model (`BarCodeHandle`) must be passed to `get_bar_code_object`. In addition, in `CandidateHandle` an index to a single decoded symbol or a single symbol candidate must be passed. Alternatively, `CandidateHandle` can be set to *'all'* and then all objects of the decoded symbols or symbol candidates are returned.

Depending on the option selected in `ObjectName` parameter the following objects are returned:

'symbol_regions': The regions of successfully decoded symbols are returned. When choosing *'all'* as `CandidateHandle`, the regions of all decoded symbols are retrieved. The order of the returned objects is the same as in `find_bar_code`.

'candidate_regions': The regions of potential bar codes are returned. If there is a total of n decoded symbols out of a total of m candidates then CandidateHandle can be chosen between 0 and (m-1). With CandidateHandle between 0 and (n-1) the original segmented region of the respective decoded symbol is retrieved. With CandidateHandle between n and (m-1) the region of the potential or undecodable symbol is returned. In addition, CandidateHandle can be set to *'all'* to retrieve all candidate regions at the same time.

'scanlines_all', *'scanlines_valid'*, *'scanlines_all_plain'*, *'scanlines_valid_plain'*, *'scanlines_merged_edges'*: XLD contours representing the particular detected bars in the scanlines applied on the candidate regions are returned. Note that the scanlines can be only be returned if the bar code was decoded by find_bar_code or decode_bar_code_rectangle2 in *'persistence'* mode (see set_bar_code_param for further details).

'scanlines_all' represents all scanlines that find_bar_code could potentially create in order to decode a bar code.

'scanlines_valid' represents only those scanlines that could be successfully decoded. The number of scanlines that has to be valid can be adjusted with set_bar_code_param with the parameter *'min_identical_scanlines'*. There will be no *'scanlines_valid'* if the symbol was not decoded. For stacked bar codes (e.g., *'GS1 DataBar Stacked'* and *'GS1 DataBar Expanded Stacked'*) this rule applies similarly, but on a per-symbol-row basis rather then per-symbol.

'scanlines_merged_edges' shows the scanlines and edges which were used for the merged scanlines (see *'merge_scanlines'* in set_bar_code_param).

get_bar_code_object returns each scanline as separate XLD contour. Please note that the XLD contours returned by *'scanlines_all'*, *'scanlines_valid'* and *'scanlines_merged_edges'* are meant for visualization purposes and hence consist of many XLD points. Double lines are used between edges that the bar code reader recognized as bar, whereas single lines are used for white spaces. Additional information about each scanline can be obtained with get_bar_code_result with the parameter *'status'*.

The *'_plain'* variants *'scanlines_all_plain'* and *'scanlines_valid_plain'* return the described scanlines in a plain format. Every edge found is returned as a point of an XLD contour. For every scanline a new XLD contour is created. Scanlines for which no edges could be found are omitted.

Parameters

▷ **BarCodeObjects** (output_object) ..... object(-array) ⇝ *object*
Objects that are created as intermediate results during the detection or evaluation of bar codes.

▷ **BarCodeHandle** (input_control) ..... barcode ⇝ *handle*
Handle of the bar code model.

▷ **CandidateHandle** (input_control) ..... integer ⇝ *string* / integer
Indicating the bar code results respectively candidates for which the data is required.
Default: 'all'
Suggested values: CandidateHandle ∈ {0, 1, 2, 'all'}

▷ **ObjectName** (input_control) ..... string ⇝ *string*
Name of the iconic object to return.
Default: 'candidate_regions'
List of values: ObjectName ∈ {'symbol_regions', 'candidate_regions', 'scanlines_all', 'scanlines_valid', 'scanlines_merged_edges', 'scanlines_all_plain', 'scanlines_valid_plain'}

Result

The operator get_bar_code_object returns the value 2 (H_MSG_TRUE) if the given parameters are correct and the requested objects are available for the last symbol search. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

find_bar_code

See also

get_bar_code_result

Module

Bar Code

```
get_bar_code_param ( : : BarCodeHandle,
    GenParamName : GenParamValue )
```

Get one or several parameters that describe the bar code model.

The operator `get_bar_code_param` allows to query parameters of a bar code model, which are of relevance for a successful search and decoding of a respective class of bar codes.

The names of the desired parameters are passed in the generic parameter `GenParamName` and the corresponding values are returned in `GenParamValue`. All of these parameters can be set and changed at any time with the operator `set_bar_code_param`.

Parameters that have been set bar code type specifically by `set_bar_code_param_specific` or by automatic parameter training cannot be read with `get_bar_code_param`, but must be read with `get_bar_code_param_specific`. All parameters listed with the operator `get_bar_code_param_specific` can always be set specifically. Which parameters are actually set specifically can be determined at runtime with the operator `query_bar_code_params`.

The following parameters can be queried – ordered by different categories:

Size of the bar code elements:

'element_size_min': Minimal size of the base bar code elements, i.e., the minimal width of the narrowest bars and spaces for the specific bar code type.

'element_size_max': Maximal size of the base bar code elements, i.e., the maximal width of the narrowest bars and spaces for the specific bar code type.

'element_size_variable': Specifies if the element size varies across a single bar code.

'barcode_height_min': Minimal height of the bar code.

'barcode_width_min': Minimal bar code width.

'barcode_width_max': Maximal bar code width.

Scanning settings:

'num_scanlines': Maximal number of scans per (candidate) bar code.

'min_identical_scanlines': Minimal number of decoded scanlines which return identical data to read the bar code successfully.

'majority_voting': Decode result selection mode. Specifies whether majority voting is used in the selection of different scanline results.

'stop_after_result_num': Number of successfully decoded bar codes after which the decoding will stop. *0* is the default where the reader tries to decode all candidates.

'orientation': Accepted orientation of the decoded bar codes.

'orientation_tol': Tolerance of the accepted orientation.

'quiet_zone': Quiet zone verification mode.

'start_stop_tolerance': Start/Stop search criteria tolerance.

'min_code_length': Minimal number of decoded characters.

'max_code_length': Maximum number of decoded characters.

'merge_scanlines': Determines whether merged scanlines are computed if not enough scanlines could be decoded individually. Thus, occluded or damaged bar codes can be read.

Appearance of the bar code in the image:

'meas_thresh': Relative threshold for the detection of edges in the bar code region.

'meas_thresh_abs': Absolute threshold for the detection of edges in the bar code region.

'contrast_min': Minimal contrast between the foreground and the background of the bar code elements.

Print quality inspection-specific values:

'quality_isoiec15416_reflectance_reference': Reference gray value to normalize the reflectances used to assess the quality grades Symbol Contrast, Minimal Reflectance, and Minimal Edge Contrast.

Values that only apply to certain code types:

'check_char': Handling of an optional check character.

'composite_code': Presence and type of a 2D composite code appended to the bar code.

'upce_encodation': Output format for UPC-E bar codes (with number system 0 or 1).

'upce1_enable': Enables the decoding of UPC-E bar codes with number system 1.

Miscellaneous:

'timeout': Timeout for `find_bar_code`

'persistence': Persistence mode of the bar code model.

'train': Returns the *names* of the trained parameters. The returned tuple can be passed as `GenParamName` to a further call to `get_bar_code_param` to get the *values* of the trained parameters.

Further details on these parameters can be found with the description of the operator `set_bar_code_param`.

Parameters

▷ **`BarCodeHandle`** (input_control) . . . . . barcode ⇝ *handle*
Handle of the bar code model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that are to be queried for the bar code model.
Default: 'element_size_min'
List of values: `GenParamName` ∈ {'check_char', 'composite_code', 'barcode_height_min', 'barcode_width_min', 'barcode_width_max', 'element_size_max', 'element_size_min', 'contrast_min', 'meas_thresh', 'meas_thresh_abs', 'min_identical_scanlines', 'majority_voting', 'num_scanlines', 'orientation', 'orientation_tol', 'persistence', 'quality_isoiec15416_reflectance_reference', 'start_stop_tolerance', 'stop_after_result_num', 'timeout', 'train', 'upce_encodation', 'upce1_enable', 'quiet_zone', 'element_size_variable', 'min_code_length', 'max_code_length', 'merge_scanlines', 'small_elements_robustness'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.name(-array) ⇝ *real* / integer / string
Values of the generic parameters.

Result

The operator `get_bar_code_param` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_bar_code_model`, `set_bar_code_param`

Possible Successors

`set_bar_code_param`

Alternatives

`get_bar_code_param_specific`

Module

Bar Code

get_bar_code_param_specific (: : BarCodeHandle, CodeTypes, GenParamName : GenParamValue)

Get parameters that are used by the bar code reader when processing a specific bar code type.

The operator get_bar_code_param_specific allows to query parameters of a bar code model, which are of relevance for a successful search and decoding of a respective class of bar codes. Contrary to get_bar_code_param, get_bar_code_param_specific allows a bar code type specific query for the following parameters. This is useful when searching different code types in one image. The bar code types are specified in CodeTypes. The names of the desired parameters are passed in the generic parameter GenParamName and the corresponding values are returned in GenParamValue. All of these parameters can be set and changed selectively with the operator set_bar_code_param_specific or for all bar code types with the operator set_bar_code_param. Which parameters are set specifically can be determined at runtime with the operator query_bar_code_params. These parameters cannot be read with get_bar_code_param, but must be read with get_bar_code_param_specific.

The following parameters can be queried:

'num_scanlines': Maximal number of scanlines used during the scanning of a (candidate) bar code.

'min_identical_scanlines': Minimal number of decoded scanlines which return identical data to read the bar code successfully.

'stop_after_result_num': Number of successfully decoded bar codes of the given code type after which the decoding will stop. *0* is the default where the reader tries to decode all candidates.

'orientation': Accepted orientation of the decoded bar codes.

'orientation_tol': Tolerance of the accepted orientation.

'quiet_zone': Quiet zone verification mode.

'start_stop_tolerance': Start/stop search criteria tolerance. Currently it is implemented only for Code 128 and GS1-128.

'check_char': Handling of an optional check character.

'composite_code': Presence and type of a 2D composite code appended to the bar code.

'min_code_length': Minimal number of decoded characters.

'max_code_length': Maximum number of decoded characters.

Further details on the above parameters can be found with the description of set_bar_code_param operator.

Parameters

▷ **BarCodeHandle** (input_control) ..... barcode ⇝ *handle*
Handle of the bar code model.

▷ **CodeTypes** (input_control) ..... string(-array) ⇝ *string*
Names of the bar code types for which parameters should be queried.
Default: 'EAN-13'
List of values: CodeTypes ∈ {'2/5 Industrial', '2/5 Interleaved', 'Codabar', 'Code 39', 'Code 93', 'Code 128', 'EAN-13', 'EAN-13 Add-On 2', 'EAN-13 Add-On 5', 'EAN-8', 'EAN-8 Add-On 2', 'EAN-8 Add-On 5', 'UPC-A', 'UPC-A Add-On 2', 'UPC-A Add-On 5', 'UPC-E', 'UPC-E Add-On 2', 'UPC-E Add-On 5', 'MSI', 'PharmaCode', 'GS1 DataBar Omnidir', 'GS1 DataBar Truncated', 'GS1 DataBar Stacked', 'GS1 DataBar Stacked Omnidir', 'GS1 DataBar Limited', 'GS1 DataBar Expanded', 'GS1 DataBar Expanded Stacked', 'GS1-128'}

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Names of the generic parameters that are to be queried for the bar code model.
Default: 'check_char'
List of values: GenParamName ∈ {'check_char', 'composite_code', 'min_identical_scanlines', 'num_scanlines', 'orientation', 'orientation_tol', 'start_stop_tolerance', 'stop_after_result_num', 'quiet_zone', 'min_code_length', 'max_code_length'}

▷ **GenParamValue** (output_control) ..... attribute.name(-array) ⇝ *real* / integer / string
Values of the generic parameters.

Result

The operator get_bar_code_param_specific returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_bar_code_model, set_bar_code_param, set_bar_code_param_specific

Alternatives

get_bar_code_param

Module

Bar Code

get_bar_code_result (: : BarCodeHandle, CandidateHandle, ResultName : BarCodeResults)

Get the alphanumerical results that were accumulated during the decoding of bar code symbols.

The operator get_bar_code_result returns alphanumeric results of the reading process in BarCodeResults. In order to obtain a result, the bar code model (BarCodeHandle) and the index of the resulting symbol or candidate (CandidateHandle) are needed. ResultName determines what kind of result is to be returned.

CandidateHandle refers to the read candidates in the order as returned by the operator find_bar_code. This implies, the possible values for CandidateHandle depend on the queried result:

- Referring to decoded symbols, values from 0 to (n-1) are possible (with n as the total number of successfully decoded symbols).
- Referring to candidates, values from 0 to (m-1) are possible (with m as the total number of candidates).

In case the asked result is single valued, CandidateHandle can also be set to *'all'* to return the results of all symbols or candidates.

The following values are supported for ResultName:

'decoded_strings': returns the decoded result as a string in a human-readable format. Note that only data characters are contained in the decoded string. Start, stop, and check characters are excluded. For optional check characters the behavior depends on the value of the *'check_char'* (see set_bar_code_param for details) parameter.
The returned result is a single value.

'decoded_types': returns the bar code type of the decoded result as a string. This is especially important in the context of autodiscrimination (see find_bar_code).
The returned result is a single value.

'decoded_data': returns the decoded result as a tuple of byte values. In contrast to *'decoded_reference'*, this contains the same high level processed data that is also returned for *'decoded_strings'*. However, if the data contains binary non-printable characters, this parameter is more convenient. Also, for Code 128 or Code 93, the decoded data may contain NULL characters. In this case, the data can only be fully retrieved using this parameter, as the NULL character will otherwise terminate the string.

'decoded_reference': returns the underlying decoded reference data. It comprises all original characters of the symbol, i.e., data characters, potential start or stop characters and check characters if present. The specific result depends on the code type:

- For codes taking only numeric data, like, e.g., the EAN/UPC codes, the GS1 DataBar codes (except the GS1 DataBar Expanded variants), or the 2/5 codes, the decoded reference data takes the same values as the decoded string data including check characters.

- For codes with alphanumeric data, like for example Code 128 or Code 39 the decoded reference data are the indices of the respective decoding table.
- For GS1-128, the decoded reference data are the indices of the respective decoding table. The special character FNC1 appears with value 102.
- For GS1 DataBar Expanded and GS1 DataBar Expanded Stacked, the reference values are the ASCII codes of the decoded data (see also Tuple / String Operations), where the special character FNC1 appears with value 29.
- Furthermore, for all codes from the GS1 DataBar family the first reference value represents a linkage flag with value of 1 if the flag is set and 0 otherwise.

The result is returned as a tuple.

'composite_strings': returns the decoded string of a GS1 Composite component. For further details see the description of the parameter *'composite_code'* of `set_bar_code_param`.

The returned result is a single value.

'composite_reference': returns the decoded string of a GS1 Composite component as a tuple of byte values. If the data contains binary non-printable characters, this parameter is more convenient than *'composite_strings'*.

'orientation': returns the orientation for the specified result. The *'orientation'* of a bar code is defined as the angle between its reading direction and the horizontal image axis. The angle is positive in counter clockwise direction and is given in degrees. Note that the reading direction is perpendicular to the bars of the bar code. In cases where the orientation of the bars can not be determined reliably, e.g., for distorted codes, the orientation of the scanlines is returned.

The returned result is a single value.

Value range: *-180.0 ... 180.0*.

'element_size': returns the size of bar code elements for the specified result.

The returned result is a single value.

'aborted': returns a value indicating whether `find_bar_code` was aborted or not:

0 - *completed* `find_bar_code` completed.

1 - *aborted* `find_bar_code` was aborted by a timeout, see `set_bar_code_param`.

2 - *aborted* `find_bar_code` was aborted using `set_bar_code_param` with *'abort'*.

The returned result is a single value.

'quality_isoiec15416': returns a tuple with the assessment of print quality in compliance with the international standard ISO/IEC 15416:2016. Note that the print quality of a bar code can only be evaluated if the bar code was decoded by `find_bar_code` in *'persistence'* mode (see `set_bar_code_param` for further details).

The first tuple element always contains the overall print quality of the symbol. The length of the tuple and the interpretation of the remaining elements depend on whether a simple 1D bar code is evaluated (e.g., EAN-13, Code 128, GS1 DataBar) or a composite bar code (e.g., GS1 DataBar Omnidirectional CC-A, GS1 DataBar Limited CC-B). The names of the grades can be queried by setting `ResultName` to *'quality_isoiec15416_labels'*, see below.

For the print quality assessment a set of scan reflectance profiles (scan lines) is generated across the symbol. For each scan line a grade for each of the evaluated parameters is allocated.

For compatibility reasons these grades are numbers from 0 to 4, where 0 is the worst and 4 the best possible grade. Setting `ResultName` to *'quality_isoiec15416_float_grades'*, the grades are returned in accordance with the standard with one decimal place. The worst of all parameter grades of a scan line is defined as the scan reflectance profile grade. Thus, each scan line has an own grade representing the respective minimal grade of all parameter grades of this scan line. The arithmetic mean of all those scan line grades is then returned as the overall symbol grade.

If a grade is not defined for the bar code type of the decoded symbol, *'N/A'* is returned. In this case, the overall symbol grade is also *'N/A'*.

It is important to note that, even though the implementation is strictly based on the standard, the computation of the print quality grades depends on the decoding algorithm used. Thus, different bar code readers (of different vendors) can potentially produce slightly different results in the print quality assessment.

The tuple describing the print quality depends on whether the bar code is simple or composite:

simple 1D bar codes:

The valuation of the print quality is described in a tuple with the following nine elements:

0 - *overall quality*: The arithmetic mean of the scan reflectance profile grade of each scan line. Note, that this value is, in most cases, not the minimum of the other symbol grades, because it depends on the individual scan reflectance profile grades.

1 - *decode*: Set to *4* when the reflectance profile of the symbol could be decoded according to the reference decode algorithm for the symbology and *0*, otherwise. Note that HALCON's decode algorithm differs from the reference decode algorithm. Thus, in many cases HALCON can decode the symbol although the *decode* grade according to the standard is *0*.

2 - *symbol contrast*: The range between the minimal and the maximal value in the reflectance profile. A strong contrast results in a good grading.

3 - *minimal reflectance*: Set to *4* if the lowest reflectance value in the scan reflectance profile is lower or equal to 0.5 of the maximal reflectance value. Otherwise a value of *0* is assigned.

4 - *minimal edge contrast*: The contrast between any two adjacent elements, both bar-to-space or space-to-bar. The *minimal edge contrast* grades the minimum of the edge contrast values measured in the reflectance profile.

5 - *modulation*: Indicates how strong the amplitudes of the bar code elements are. Big amplitudes make the assignment of the elements to bars or spaces more certain, resulting in a high modulation grade.

6 - *defects*: Reflects irregularities in the gray value profile found within elements and quiet zones.

7 - *decodability*: Reflects deviations of the element widths from the nominal widths defined for the corresponding symbology.

8 - *additional requirements*: A grade concerning symbology specific requirements of the bar code. It mostly regards the required quiet zones, but sometimes it can also be related to e.g., wide/narrow ratio, inter-character gaps, guarding patterns. For the following code types the conformity to the corresponding ISO/IEC standard cannot be guaranteed. This is mainly because the computation of the *additional requirements* requires the availability of metrical information.

Code 39: the grade of the inter-character gap depends on the measured narrow element width given in millimeters. Since in `find_bar_code` neither metric information is available nor can be calculated, `get_bar_code_result` will only return the grade for the inter-character gap assuming that the measured narrow element width is less than 0.287mm.

EAN-13, EAN-8, UPC-A, UPC-E (including all add-on variants): ISO/IEC 15420:2009 specifies the magnification factors as an additional criterion. A computation of this criterion is not possible since this would require knowledge of the metrical size of the bar code.

While *overall quality* is the final symbol grade to be reported, the rest of the grades give information for possible causes for poor quality of a symbol. A detailed list of frequently appearing defects and their effect on the individual grades can be found with the ISO/IEC 15416:2016 standard.

composite bar codes:

The valuation of the print quality is described in a tuple with 24 elements, which we organize for better clarity in three groups: `OVERALL`, `LINEAR`, and `COMPOSITE`, the latter one including a sub group. The overall grades that are listed in the group `OVERALL`. The grades in the groups `LINEAR` and `COMPOSITE` give information of possible causes for poor quality of the symbol.
In the following, the individual tuple elements with their corresponding index and definition are listed in groups.

- The group `OVERALL`:
  These grades represent the respective minimum of all individual grades of each group.

  0 - *overall quality*: The final symbol grade to be reported. It returns the minimum of all grades.

  1 - *overall linear*: The overall grade for the symbols of the group `LINEAR`, the linear 1D symbols. This grade returns the minimum of all grades belonging to the group.

  2 - *overall composite*: The overall grade for the symbols of the group `COMPOSITE`, the composite 2D symbols. This grade returns the minimum of all grades belonging to the group.

- The group `LINEAR`:
  The interpretation of the grades in the `LINEAR` group correspond to those for simple 1D bar codes described above.

  3 - *decode*

  4 - *symbol contrast*

  5 - *minimal reflectance*

  6 - *minimal edge contrast*

7 - *modulation*

8 - *defects*

9 - *decodability*

10 - *additional requirements*

- The group COMPOSITE:
  these grades are equivalent to the quality grades for PDF 417 data code symbols, thus of two-dimensional symbols according to ICO/IEC 15415:2011 (see *'quality_isoiec15415'* in get_data_code_2d_results). However, there is a difference: For PDF 417 data code symbols a start/stop pattern is used for the evaluation concerning the quality of the reflectance profile. In contrast, for composite bar codes so-called Raw Address Patterns (RAP) are used instead of a start/stop pattern. The grades of the respective RAP symbols are organized in the sub group COMPOSITE RAP and are consistent with the grades for simple 1D bar codes.

  11 - *decode*: Grade for the decodability of the composite symbol part. Its meaning and determination is equivalent to the one for simple 1D bar codes described above.

  12 - *RAP overall*: Minimum of all individual grades of the sub group COMPOSITE RAP.

  13 - *RAP contrast*: Grade of the COMPOSITE RAP sub group, corresponds to the one for simple 1D bar codes as described above.

  14 - *RAP minimal reflectance*: Grade of the COMPOSITE RAP sub group, corresponds to the one for simple 1D bar codes as described above.

  15 - *RAP minimal edge contrast*: Grade of the COMPOSITE RAP sub group, corresponds to the one for simple 1D bar codes as described above.

  16 - *RAP modulation*: Grade of the COMPOSITE RAP sub group, corresponds to the one for simple 1D bar codes as described above.

  17 - *RAP defects*: Grade of the COMPOSITE RAP sub group, corresponds to the one for simple 1D bar codes as described above.

  18 - *RAP decodability*: Grade of the COMPOSITE RAP sub group, corresponds to the one for simple 1D bar codes as described above.

  19 - *codeword yield*: Counts and evaluates the relative number of correct decoded words acquired by the set of scan profiles.

  20 - *unused error correction*: Counts and evaluates the relative number of false decoded words within the error correction blocks.

  21 - *modulation*: Grades the ratio of the minimum edge contrast to the symbol contrast. indicates how strong the amplitudes, i.e., the extremal intensities, of the bars and spaces are.

  22 - *decodability*: Grades the uniformity of reflectance of the dark and light modules, respectively.

  23 - *defects*: Refers to a measurement of how perfect the reflectance profiles of bars and spaces are.

'quality_isoiec15416_float_grades': returns a tuple with the same assessment of print quality like *'quality_isoiec15416'*. In compliance with the international standard ISO/IEC 15416:2016 the grades are returned with one decimal place. The names of the grades can be queried by setting ResultName to *'quality_isoiec15416_labels'*, see below.

Note that the print quality of a bar code can only be evaluated if the bar code was decoded by find_bar_code in *'persistence'* mode (see set_bar_code_param for further details).

'quality_isoiec15416_values': returns a tuple with the raw values for all *'directly measurable'* grades (returned with ResultName to *'quality_isoiec15416'*). These are the grades, whose definition in the ISO/IEC 15416:2016 standard is a *'direct derivative'* of the reflectance (i.e., the gray values) properties of the symbol or grades that are the result of a *'direct counting'*. Additionally the three overall grades *overall quality*, *overall linear*, and *overall composite* are returned (see their explanation above in *'quality_isoiec15416'*). Note that the print quality of a bar code can only be evaluated if the bar code was decoded by find_bar_code in *'persistence'* mode (see set_bar_code_param for further details).

The names of the tuple elements can be queried by setting ResultName to *'quality_isoiec15416_labels'*, see below.

All values (not grades) are normalized between *0.0* and *1.0*. Hence, for example, a *symbol contrast* value of *0.75* will correspond to a gray value of *191.25* (for byte images).

Which values make up the tuple depends on whether the bar code is a simple 1D or a composite bar code:

simple 1D bar codes:

The following values are returned by the respective index:

0 - *overall quality*
2 - *symbol contrast*
3 - *minimal reflectance*
4 - *minimal edge contrast*
5 - *modulation*
6 - *defects*
7 - *decodability*

composite bar codes:
The following values are returned by the respective index:
0 - *overall quality*
1 - *overall linear*
2 - *overall composite*
4 - *symbol contrast*
5 - *minimal reflectance*
6 - *minimal edge contrast*
7 - *modulation*
8 - *defects*
9 - *decodability*
13 - *RAP contrast*
14 - *RAP minimal reflectance*
15 - *RAP minimal edge contrast*
16 - *RAP modulation*
17 - *RAP defects*
18 - *RAP decodability*
19 - *codeword yield*
20 - *unused error correction*

These tuples have the same length as the tuple with the grades returned when setting `ResultName` to *'quality_isoiec15416'* and *'quality_isoiec15416_float_grades'*, respectively. For the entries not listed above, the operator reports *'N/A'*. In case of simple 1D bar codes, the grades *decode* and *additional requirements* do not have any interpretation in the reflectance profile and as a consequence the value *'N/A'* is returned. For composite bar codes this is the case for the grades *decode* (`LINEAR`), *additional requirements* (`LINEAR`), *decode* (`COMPOSITE`), and *RAP overall* (`COMPOSITE`). Note that although the grades *modulation* (`COMPOSITE`), *decodability* (`COMPOSITE`), and *defects* (`COMPOSITE`) are grading the gray value reflectance profile of a composite symbol, *'N/A'* is reported for them as well. This is because they are computed in a complicated scheme involving the symbology decoding routine and the error correction mechanism. As a consequence they do not have a direct raw measurement interpretation.

If a grade is not defined for the bar code type of the decoded symbol, *'N/A'* is returned as its value. In this case, the value of the overall symbol grade is also *'N/A'*.

'quality_isoiec15416_labels': returns convenience grade labels of the elements of the tuple returned when calling `get_bar_code_result` with *'quality_isoiec15416'*. Note, that in order to be able to discriminate the composite from the linear grading case, the operator needs a handle of a valid result to be passed in `CandidateHandle`.

'gs1_lint_passed': If the symbol contains a GS1 formatted string, additional linting can be performed on the contained string. *'gs1_lint_passed'* will return *'true'*, if this is successful. Otherwise, *'false'* is returned. Linting checks that

- one AI (Application Identifier) does not occur multiple times with different contents,
- AIs required by other AIs are contained,
- AIs excluded by other AIs are not included, and that
- specific formatting rules for AIs are respected.

'gs1_lint_result': *'gs1_lint_result'* performs the linting tests documented in *'gs1_lint_passed'*. If the linting fails, a list of human-readable error messages is returned, if it is successful, *'ok'* is returned.

'status': determines and returns additional information about scanlines of a given candidate region in a human-readable format. Note in order to evaluate the status of the bar code scanlines the preceding call of the operator `find_bar_code` or `decode_bar_code_rectangle2` needs to be done in the mode *'persistence'*, see `set_bar_code_param`.

For the calculation of additional information the given candidate region is scanned again. In doing so, the bar code reader exits only after evaluation of all scanlines and not usual after a successful decoding. This is computationally expensive and should be queried only if additional information is needed. For further information on setting scanline parameters, see `set_bar_code_param`.

The operator returns for every scanline of the candidate region a status message and possible warning message, which will be added to the string containing the status message. These messages are sorted in the same order as the scanlines themselves are returned by `get_bar_code_object` with the parameter *'scanlines_all'*. The possible messages of these categories are listed below.

- Status messages:
  The following list shows the possible messages grouped into stages in which the message can appear. The numbers are the corresponding status codes returned by *'status_id'* (see below).

  0 - *'unknown status.'* The status of this scanline is unknown. The scanline will be ignored.

  1 - *'success.'* The scanline could be decoded successfully.

  2 - *'edges: not enough edges detected.'* The number of edges in this scanline is too low for this bar code type.

  3 - *'edges: not enough edges for a start, a stop and at least one data character.'* The number of edges in this scanline is too low to find at least the start pattern, the stop pattern and a data character.

  4 - *'edges: too many edges detected.'* The number of edges in this scanline is too high for this bar code type.

  5 - *'edges: center of scanline not within image domain.'*

  6 - *'decoding: could not find stop character.'* The symbology specific stop character could not be found.

  7 - *'decoding: could not find start and stop characters.'* The symbology specific start and stop characters could not be found.

  8 - *'decoding: internal error when estimating the maximum string length.'* Internal error.

  9 - *'decoding: internal error when decoding a single character.'* Internal error.

  10 - *'decoding: number of wide bars of a single character is not equal to 2.'* For bar code types *'2/5 Industrial'* and *'2/5 Interleaved'*, the number of wide bars in a single character must be two.

  11 - *'decoding: invalid encoding pattern.'* The encoding pattern does not correspond to a character in the symbology specific decoding table.

  12 - *'decoding: invalid mix of character sets.'* For bar code types *'EAN-13'* and *'UPC-A'*, the left half of the symbol contains an invalid mix of the number sets A and B.

  13 - *'decoding: error decoding the reference to a human readable string.'* For example, this could happen if not enough characters (depending on whether a check character is expected) could be found.

  14 - *'decoding: could not detect center guard pattern.'* For bar code types *'EAN-13'*, *'EAN-8'*, and *'UPC-A'* (including add-on variants) the obligatory center guard pattern could not be found.

  15 - *'decoding: could not detect left and/or right guard patterns.'* For bar code types *'EAN-13'*, *'EAN-8'*, and *'UPC-A'* (including add-on variants) either the obligatory left or right normal guard patterns could not be found. For bar code types *'UPC-E'* (including add-on variants) either the left normal or right special guard patterns could not be found.

  16 - *'decoding: could not detect add-on guard pattern.'* For bar code types *'EAN-13'*, *'EAN-8'*, *'UPC-A'*, and *'UPC-E'* containing add-on symbols, the obligatory add-on guard pattern could not be found.

  17 - *'decoding: could not detect enough finder patterns.'* For bar code types of the GS1 DataBar family, not enough finder patterns could be found.

  18 - *'decoding: no segment found.'* For stacked bar code types no segment could be found.

  19 - *'check: checksum test failed.'* The check character test failed. For bar codes with an optional check character, it is possible to disable the check character testing with *'check_char'* in `set_bar_code_param`.

  20 - *'check: check of add-on symbol failed.'* For bar code types *'EAN-13'*, *'EAN-8'*, *'UPC-A'*, and *'UPC-E'* containing add-on symbols, the mix of the number sets A and B does not match the implicit check digit.

21 - *'check: detected EAN-13 bar code type instead of specified type.'* For the bar code type *'UPC-A'* and its add-on variants, the left half of the symbol must consist of six symbol characters of number set A, but a *'EAN-13'* compatible mix of number sets A and B has been found instead. Try decoding the bar code as *'EAN-13'*.

22 - *'check: symbol region overlaps with another symbol region'* The bar code could be decoded, but its symbol region intersects with the symbol region of another successfully decoded symbol.

23 - *'decoding: could not detect the bar code type.'* While scanning a candidate region in autodiscrimination mode (see `find_bar_code` with `CodeType`=*'auto'*), the decoder was unable to detect which bar code type the symbol belongs to.

24 - *'decoding: quiet zone is disturbed.'* The quiet zone check was not successful. See the section *'quiet_zone'* in `set_bar_code_param` for further information.

25 - *'check: symbol region detected outside of the candidate region.'* A bar code could be decoded, but the resulting symbol region does not intersect the original candidate region. This is an indication that random clutter or another candidate was detected by accident by the scanlines used to scan the original candidate.

26 - *'decoding: minimum code length not reached (see min_code_length).'* The number of characters in the found code is smaller than the required amount. See the section *'min_code_length'* in `set_bar_code_param` for further information about the minimal required code length.

27 - *'decoding: maximum code length exceeded (see max_code_length).'* The number of characters in the found code exceeds the maximum. See the section *'max_code_length'* in `set_bar_code_param` for further information about the maximum allowed code length.

28 - *'decoding: could not estimate bar width threshold.'* Specific to Pharmacode. It was not possible to estimate a threshold to distinguish between wide and narrow bars.

- Warning messages:
  The messages listed below are only warnings, not errors. They can appear in combination with a status message and are then added to the status message string. These messages are returned if the bar code reader detects possible quality issues in the input image. The messages 1000 to 1003 can appear only for successfully decoded scanlines and only for the following bar code types: *'Codabar'*, *'2/5 Industrial'*, *'2/5 Interleaved'*, *'Code 39'*, *'Code 93'*, *'GS1-128'*, *'Code 128'*, *'MSI'*, all *'EAN'*-, *'UPC-A'*- and *'UPC-E'*-variants. Note that in cases where scanlines were already decoded incorrectly but not recognized as invalid by the bar code reader, these warnings will be wrong and must be ignored.

  1000 - *'White spaces too wide.'* The measured width of the white spaces is bigger than internally expected. This is not a decoding error, but a warning that increasing white spaces could lead to undecodable symbols.

  1001 - *'White spaces too narrow.'* The measured width of the white spaces is smaller than internally expected. This is not a decoding error, but a warning that decreasing white spaces could lead to undecodable symbols.

  1002 - *'Bars too wide.'* The measured width of the bars is bigger than internally expected. This is not a decoding error, but a warning that increasing the width of the bars could lead to undecodable symbols.

  1003 - *'Bars too narrow.'* The measured width of the bars is smaller than internally expected. This is not a decoding error, but a warning that decreasing the width of the bars could lead to undecodable symbols.

  1004 - *'Possible saturation of gray values.'* Internal algorithms show that the white pixels in the symbol region could be saturated. This warning is not bar code type specific. This warning is returned for each scanline. For example, the combination of the status messages 1001/1004 or 1003/1004 is a hint, that the input images might be overexposed.

  1005 - *'No composite component found.'* The linkage flag in the linear bar code component indicates that there should be a composite component, but the composite component could not be found. The decoder returned the linear component only. This warning is returned for each scanline.

  1006 - *'Used for merging.'* This message is an information that some edges of the current scanline were used to compute the merged scanline.

'status_id': returns additional information about scanlines in a numeric format that can be easily parsed. The description of this functionality and the message numbers are described with the parameter *'status'* above. The numbers of the warnings can be returned together with other status numbers. The single status message and warning numbers for each scanline are returned as a string, separated by a semicolon, e.g., `'1;1000;1004'`.

'status_small_elements_robustness': determines the status of an optional decoding attempt, which is described in the section about *'small_elements_robustness'* in `set_bar_code_param`. Additionally to the messages listed for *'status'*, *'status_small_elements_robustness'* can return the status *'small_elements_robustness: no scan.'* if for some reason the algorithm was not executed or failed.

'decode_feature': returns the feature that is used for the successful symbol decoding.

The returned result is a single value.

Suggested values: *'standard'*, *'merge_scanlines'*, *'small_elements_robustness'*.

Parameters

▷ **BarCodeHandle** (input_control) . . . . . barcode ⇝ *handle*
Handle of the bar code model.

▷ **CandidateHandle** (input_control) . . . . . integer ⇝ *string* / integer
Indicating the bar code results respectively candidates for which the data is required.
Default: 'all'
Suggested values: `CandidateHandle` ∈ {0, 1, 2, 'all'}

▷ **ResultName** (input_control) . . . . . attribute.name ⇝ *string*
Names of the resulting data to return.
Default: 'decoded_types'
Suggested values: `ResultName` ∈ {'decoded_types', 'decoded_strings', 'decoded_data', 'decoded_reference', 'element_size', 'orientation', 'composite_strings', 'composite_reference', 'aborted', 'gs1_lint_result', 'gs1_lint_passed', 'quality_isoiec15416', 'quality_isoiec15416_labels', 'quality_isoiec15416_values', 'quality_isoiec15416_float_grades', 'status', 'status_id', 'status_small_elements_robustness', 'decode_feature'}

▷ **BarCodeResults** (output_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
List with the results.

Result

The operator `get_bar_code_result` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct and the requested results are available for the last symbol search. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`find_bar_code`

See also

`get_bar_code_object`

Module

Bar Code

```
query_bar_code_params ( : : BarCodeHandle,
    Properties : GenParamName )
```

Get the names of the parameters that can be used in set_bar_code and get_bar_code* operators for a given bar code model*

The operator `query_bar_code_params` returns parameter names of the bar code model that correspond to the selection given in `Properties`. To explain this, here first some background: After creating a bar code model, all parameters are initially set *'general'*, i.e., they have the same values set for each supported bar code type. During automatic parameter training or after calling the operator `set_bar_code_param_specific` some parameters might become bar code type specific at runtime. These parameters can not be read with the general operator `get_bar_code_param` but must be read for a given bar code type with `get_bar_code_param_specific`. To make this access easy and generic, the output `GenParamName` of

query_bar_code_params can be used as input parameter GenParamName in get_bar_code_param or get_bar_code_param_specific, depending on Properties='*general' or Properties='*specific'.

Please note that you could alternatively use a static tuple of parameter names for get_bar_code_param or get_bar_code_param_specific (e.g., the parameter names described with these operators), but this is inflexible and therefore not recommended.

Possible values for Properties are:

'all': Returns the names of all parameters supported by the bar code reader. The names are independent of the bar code model given in BarCodeHandle.

'general': Returns the names of the parameters that contain the same values for all supported bar code types, i.e., there are no specific values set for certain bar code types, e.g., with set_bar_code_param_specific. The values of only these parameters can be read with get_bar_code_param.

Please note that the parameter name *'train'* is explicitly excluded here. *'train'* in a following call to get_bar_code_param would return a multivalue tuple and therefore couldn't be used within one tuple in combination with other parameter names.

'specific': Returns the names of the parameters that contain specific values for certain bar code types. The values of these parameters must be read with get_bar_code_param_specific instead of get_bar_code_param. Bar code type specific values can occur in conjunction with bar code autodiscrimination, described with the operator find_bar_code or automatic parameter training.

'trained_general': Returns the names of parameters that have already been trained and contain the same values for all supported bar code types. The training mode is described with the operator set_bar_code_param.

'trained_specific': Returns the names of the parameters that have already been trained and contain specific values for certain bar code types. The training mode is described with the operator set_bar_code_param.

Parameters

▷ **BarCodeHandle** (input_control) ..... barcode ⇝ *handle*
Handle of the bar code model.

▷ **Properties** (input_control) ..... attribute.name ⇝ *string*
Properties of the parameters.
Default: 'trained_general'
List of values: Properties ∈ {'general', 'specific', 'trained_general', 'trained_specific', 'all'}

▷ **GenParamName** (output_control) ..... attribute.name(-array) ⇝ *string*
Names of the generic parameters.

Example

```
TrainParams := ['element_size_min', 'element_size_max', 'orientation']
Train3times := gen_tuple_const(|TrainParams|,'train')
create_bar_code_model (Train3times, TrainParams, BarCodeHandle)
find_bar_code (Image, SymbolRegions, BarCodeHandle, 'EAN-13', \
               DecodedDataStrings)
query_bar_code_params (BarCodeHandle, 'trained_general', NamesGen)
* returns NamesGen = ['element_size_max', 'element_size_min']
get_bar_code_param (BarCodeHandle, NamesGen, ValGen)
* returns e.g., ValGen = [4.0, 1.5]
query_bar_code_params (BarCodeHandle, 'trained_specific', NamesSpec)
* returns NamesSpec = ['orientation', 'orientation_tol']
get_bar_code_param_specific (BarCodeHandle, 'EAN-13', NamesSpec, ValSpec)
* returns e.g., ValSpec = [89.9127, 0.5]
```

Result

The operator query_bar_code_params returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

find_bar_code, decode_bar_code_rectangle2

Possible Successors

get_bar_code_param, get_bar_code_param_specific

Module

Bar Code

read_bar_code_model (: : FileName : BarCodeHandle)

Read a bar code model from a file and create a new model.

The operator read_bar_code_model reads the bar code model from the file FileName and creates a new model that is an identical copy of the saved model. The parameter BarCodeHandle returns the handle of the new model. The model file FileName must have been created by the operator write_bar_code_model. The default HALCON file extension for bar code model is 'bcm'.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the bar code model file.
Default: 'bar_code_model.bcm'
File extension: .bcm

▷ **BarCodeHandle** (output_control) ..... barcode ⇝ *handle*
Handle of the bar code model.

Result

The operator read_bar_code_model returns the value 2 (H_MSG_TRUE) if the named bar code file was found and correctly read. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

find_bar_code

Alternatives

create_bar_code_model

See also

write_bar_code_model, clear_bar_code_model

Module

Bar Code

serialize_bar_code_model (
: : BarCodeHandle : SerializedItemHandle)

Serialize a bar code model.

serialize_bar_code_model serializes a bar code model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by

write_bar_code_model is converted to a serialized item. The bar code model is defined by the handle BarCodeHandle. The serialized model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_bar_code_model.

Parameters

▷ **BarCodeHandle** (input_control) ..... barcode ⇝ *handle*
Handle of the bar code model.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

The operator serialize_bar_code_model returns the value 2 (H_MSG_TRUE) if the passed handle of the bar code model is valid and if the model can be serialized into the serialized item. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_bar_code_model

See also

create_bar_code_model, set_bar_code_param, find_bar_code

Module

Bar Code

set_bar_code_param (: : BarCodeHandle, GenParamName, GenParamValue :)

Set selected parameters of the bar code model.

The operator set_bar_code_param is used to set or change the different parameters of a bar code model in order to adapt to special properties of the bar codes or to a particular appearance in the image. All parameters can also be set while creating the bar code model with create_bar_code_model. The current configuration of the bar code model can be queried with get_bar_code_param.

The following overview lists the different generic parameters with the respective value ranges and default values:

Size of bar code elements:

The first two parameters, *'element_size_min'* and *'element_size_max'*, influence the search for candidate regions and the decoding of the bar code. But note that these values are not used as strict limits for the size of the elements of found bar codes, i.e., also bar codes with elements that are smaller than *'element_size_min'* or larger than *'element_size_max'* may be found.

Attention: When the size is changed by calling the operator set_bar_code_param, the consistency of the parameters *'element_size_min'* and *'element_size_max'* is verified. Both parameters should be set in the same call.

'element_size_min': Minimal size of the base bar code elements (also called 'modules' or 'narrow bars', depending on the specific bar code type), i.e., the minimal width of the narrowest bars and spaces for the specific bar code type. The value of this parameter is defined in pixels. For low resolution bar codes, the value should be reduced to *1.5*, and in some cases even as low as *0.6*. If the *'element_size_min'* is specified below *2.0* pixels an additional decoding attempt is made (see *'small_elements_robustness'* for more details on this attempt). In the case of huge bar codes, the value should be increased, which results in a shorter execution time and fewer candidates.

Suggested values: [*1.2*, ..., *10.0*]

Value range: [*0.6*, ..., *64*]

Default: *2.0*

'element_size_max': Maximal size of the base bar code elements (also called 'modules' or 'narrow bars', depending on the specific bar code type), i.e., the maximal width of the narrowest bars and spaces for the specific bar code type. The value of *'element_size_max'* is defined in pixels. It should be adequately low such that two neighboring bar codes are not fused into a single one. On the other hand the value should be sufficiently high in order to find the complete bar code region.

Suggested values: [*4.0*, ..., *60.0*]

Value range: [*1*, ..., *256*]

Default: *8.0*

'element_size_variable': In some bar code images, the smallest element size may vary across a given bar code object. These deformations might be caused by perspective projection or by deformations of the surface on which the bar code is printed (e.g., barrel distortion on a bottle). By default, the bar code reader cannot handle such distortions. However, if *'element_size_variable'* is set to *'true'*, the bar code reader tries to compensate for such distortions. Please note that in some situations it is still not possible to undistort the bar code image.

The parameter *'element_size_variable'* applies to the following bar code types only:

- GS1 DataBar Limited
- GS1 DataBar Expanded
- GS1 DataBar Expanded Stacked

Any other bar code type is unaffected by this parameter.

List of values: *'false'*, *'true'*

Default: *'false'*

'barcode_height_min': Minimal bar code height. The value of this parameter is defined in pixels. The default value is *-1*, meaning that the bar code reader automatically derives a reasonable height from the other parameters. Just for very flat and very high bar codes a manual adjustment of this parameter can be necessary. In the case of a bar code with a height of less than 16 pixels the respective height should be set by the user. Note, that the minimal value is 8 pixels. If the bar code is very high, i.e. 70 pixels and more, manually adjusting to the respective height can lead to a speed-up of the subsequent finding and reading operation.

Suggested values: *-1*, *8*, *64*

Default: *-1*

'barcode_width_min': Minimal bar code width. The value of this parameter is defined in pixels. The width of a bar code depends on many factors:

- resolution of the camera
- distance between camera and bar code
- bar code type
- number of encoded characters

If these properties are constant throughout the application, this parameter should be set in order to increase both speed and robustness. The default value is *-1*, meaning that the reader estimates a minimal bar code width based on symbology specifications and the parameter *'element_size_min'*.

Suggested values: *-1*, *40*, *50*, *60*

Default: *-1*

'barcode_width_max': Maximal bar code width. The value of this parameter is defined in pixels. The width of a bar code depends on many factors:

- resolution of the camera
- distance between camera and bar code
- bar code type
- number of encoded characters

If the maximal bar code width is known, this parameter can increase the robustness. Especially the unintended merging of bar codes lying close to each other can be avoided reliably. In general it should not be necessary to set this parameter. The default value is *-1*, meaning that the maximal bar code width is not restricted.

Suggested values: *-1*, *300*, *400*, *500*

Default: *-1*

Scanning settings:

'num_scanlines': Maximum number of scanlines used during the scanning of a (candidate) bar code. If *'num_scanlines'* is not set (the parameter has a value of *0*), the maximum number of scanlines is determined internally and it will be *10* for all single-row bar codes, *20* for GS1 DataBar Stacked and GS1 DataBar Stacked Omnidirectional, and *55* for GS1 DataBar Expanded Stacked. With this parameter, you can improve performance in two cases. In the first case, the image contains many false candidates. While the bar code itself is usually decoded after one or two scans (except for stacked bar codes, see below), a false candidate is scanned with the default value of 10 scanlines, which increases the runtime unnecessarily. Reducing the number of scanlines can therefore improve performance in images with many false candidates. As a rule of thumb, images with higher quality need less scanlines than images of lower quality. For an average image, a value between 2 and 5 should be sufficient. If a bar code can, however, not be detected after reducing the number of scanlines, the number has to be increased again. The second case concerns stacked bar codes (currently GS1 DataBar Stacked, GS1 DataBar Stacked Omnidirectional, and GS1 DataBar Expanded Stacked). Here, all scanlines are evaluated, in contrast to single-row bar codes (e.g., Code 128, EAN 13, or GS1 DataBar Limited) where the scanning stops after the code is decoded successfully. Since the scanning of the scanlines is actually one of the most time consuming parts of the `find_bar_code` algorithm, adjusting *'num_scanlines'* might result in essential performance improvements. This is especially valid for GS1 DataBar Expanded Stacked. A GS1 DataBar Expanded Stacked symbol might have up to 11 rows and therefore 55 scanlines are required to robustly detect all of them. If only symbols with smaller number of rows are expected, you can reduce *'num_scanlines'*, leaving 1.5 to 5 scanlines per row.

If this parameter is set to a value that is smaller than the specified *'min_identical_scanlines'* value, the value of the parameter *'min_identical_scanlines'* is automatically reduced to the value of the specified *'num_scanlines'*.

This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`.

Suggested values: *0*, *5*, *10*, *20*

Default: *0*

'min_identical_scanlines': With this parameter, the probability can be lowered to read bar codes wrongly or in places where no bar codes is present. The parameter specifies the minimal number of decoded scanlines which return identical data to read the bar code successfully.

If this parameter is set to 0 (which is the default for all code types, except for '2/5 Industrial' and '2/5 Interleaved'), a bar code is considered decoded with the first successfully decoded scanline (in the case of stacked codes, there must be a successful scanline per symbol row). If *'majority_voting'* is enabled, a different scheme applies (see the section on *'majority_voting'* for more details). Increasing this parameter to 2 or more is particularly useful, to prevent that a bar code is detected accidentally. This can typically happen if a scanline extracted erroneous or false edges out of a low quality image or in a very noisy image fragment. This parameter can be used to reduce the number of false detections also in cases where a specific bar code type is searched for in an image containing symbols from other bar code symbologies.

Note that the default value of this parameter is *0* for most code types. However, the default value is *2* for *'2/5 Industrial'* and *'2/5 Interleaved'* bar code types. The reason for this is that these bar code types are easily misread in other structures like clutter and text with a *'min_identical_scanlines'* setting of *1*.

This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`.

If merged scanlines (see *'merge_scanlines'*) are used, *'min_identical_scanlines'* specifies in how many scanlines each edge must be recognized successfully (see `get_bar_code_object`, *'scanlines_merged_edges'*).

For *'GS1 DataBar Expanded Stacked'*, *'min_identical_scanlines'* specifies in how many scanlines each segment must be recognized successfully.

Suggested values: *0*, *2*, *3*

Default:

Bar code Type	*'min_identical_scanlines'*
2/5 Industrial	2
2/5 Interleaved	2
all others	0

'majority_voting': This parameter controls the decode result selection mode. If this parameter is *'false'*, a successful decode result is returned if the minimal number of identically decoded scanlines are found (see section on *'min_identical_scanlines'* for more details). By setting this parameter to *'true'* a majority voting scheme is used to select between different scanline results. The result which is decoded by the majority of all scanlines is selected as the overall result. Please note that setting this parameter to *'true'* leads to a slightly increased runtime since nearly all scanlines have to be considered instead of just the minimal identical ones.

In order to reduce false decode results this parameter should be enabled.

This parameter is only supported for non-stacked bar code types.

List of values: *'false'*, *'true'*

Default: *'false'*

'stop_after_result_num': Number of successfully decoded bar codes after which the decoding will stop. If this parameter is not set (has a value of *0*), all bar code candidates are decoded. Typically, this parameter is set if the number of expected bar codes is known in advance. Then the bar code reader can abort further decoding of candidates after a certain number of bar codes has been found and the overall decoding time will decrease.

This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`.

Suggested values: *0*, *1*, *2*

Default: *0*

'orientation': Expected bar code orientation. A potential (candidate) bar code contains bars with similar orientation. The *'orientation'* and *'orientation_tol'* parameters are used to specify the range [*'orientation'*-*'orientation_tol'*, *'orientation'*+*'orientation_tol'*]. `find_bar_code` processes a candidate bar code only when the average orientation of its bars lies in this range. If the bar codes are expected to appear only in certain orientations in the processed images, you can reduce the orientation range adequately. This enables an early identification of false candidates and hence shorter execution times. This adjustment can be used for images with a lot of texture, which includes fragments tending to result in false bar code candidates.

The actual orientation angle of a bar code is explained with `get_bar_code_result`(...,*'orientation'*,...) with the only difference that for the early identification of false candidates the reading direction of the bar codes is ignored, which results in relevant orientation values only in the range [*-90.0 ... 90.0*]. The only exception to this rule constitutes the bar code symbol PharmaCode, which possesses a forward and a backward reading direction at the same time: here, *'orientation'* can take values in the range [*-180.0 ... 180.0*] and the decoded result is unique corresponding to just one reading direction.

This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`.

Suggested values: [*-90.0*, ..., *90.0*]

Default: *0.0*

'orientation_tol': Orientation tolerance. Please refer to the explanation of *'orientation'* parameter for further information. As explained there, relevant orientation values are only in the range of [*-90.0 ... 90.0*], which means that with *'orientation_tol'* = *90* the whole range is spanned. Therefore, valid values for *'orientation_tol'* are only in the range of [*0.0 ... 90.0*]. The default value *90.0* means that no restriction on the bar code candidates is performed.

This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`.

Suggested values: [*0.0*, ..., *90.0*]

Default: *90.0*

'quiet_zone': Enforces the verification of the quiet zones of a bar code symbol. When enabled, scanlines are rejected when unexpected bars are detected within the quiet zones both left or right of a detected bar code sequence. Possible values:

- *'false'*: The quiet zones verification is disabled.
- *'true'*: The quiet zones must be at least as wide as specified by the corresponding bar code standard. The following values apply (in X units, where X stands for "module width" and corresponds to the smallest width of a bar in the sequence):
- An integer value (>= *1*): The quiet zones must be at least as wide as *'quiet_zone'*×X.
- *'tolerant'*: A limited number of edges are allowed in the quiet zones, but at most 1 per 4 module widths. The intent of this is to prevent detecting only part of a bar code, while still allowing to read bar codes with simple quiet zone violations.

Bar code Type	Left QZ	Right QZ	Bar code Type	Left QZ	Right QZ
2/5 Industrial	10	10	UPC-A	9	9
2/5 Interleaved	10	10	UPC-A Add-On 2	9	5
Codabar	10	10	UPC-A Add-On 5	9	5
Code 39	10	10	UPC-E	9	7
Code 93	10	10	UPC-E Add-On 2	9	5
Code 128	10	10	UPC-E Add-On 5	9	5
MSI	10	10	GS1-128	10	10
PharmaCode	5	5	GS1 DataBar Omnidir	1	1
EAN-8	7	7	GS1 DataBar Truncated	1	1
EAN-8 Add-On 2	7	5	GS1 DataBar Stacked	1	1
EAN-8 Add-On 5	7	5	GS1 DataBar Stacked Omnidir	1	1
EAN-13	11	7	GS1 DataBar Limited	1	5
EAN-13 Add-On 2	11	5	GS1 DataBar Expanded	1	1
EAN-13 Add-On 5	11	5	GS1 DataBar Expanded Stacked	1	1

Note: Even for code types with a specified quiet zone smaller than four module widths, the edges are checked within a range of four module widths around the code. In this case, this criterion can be stricter than the verification with *'quiet_zone'='true'*.

The quiet zone verification is very useful when using the bar code reader in multi-type or 'auto' reading mode. It prevents that simple bar code types are detected inside of a bar sequence representing a longer bar code or inside another, typically more complex bar code type. Usually, values between *2* and *4* achieve optimal results by effectively suppressing false bar codes, but still tolerating small disturbances, textures, label edges, etc. next to the symbol.

This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`.

Note: The quiet zones serve as a check, that no bar is detected within this zones, but it is not required that the quiet zones are contained as a whole in the image. This means, if a set quiet zone is not contained as a whole, only the part within the image is assessed and the part outside the image is assumed to be free of defects.

Suggested values: *'false'*, *'true'*, *'tolerant'*, *1*, ..., *100*

Default: *'false'*

'start_stop_tolerance': Enforces a tolerant (*'high'*) or strict (*'low'*) searching criteria while inspecting a scanline for a start or stop pattern, respectively. A more tolerant criteria will generally increase the detection chances of a bar code, provided that a clear symbol is imaged in the processed image. On the other side, it might result in false detections in noisy images or images containing symbols from other symbologies. Less tolerant criteria increase the robustness against false detections, but might reduce the general detection rate. Currently, there are two distinct criteria implemented only for Code 128 and GS1-128.

This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`. As already mentioned, currently this makes a difference only for Code 128 and GS1-128.

List of values: *'high'*, *'low'*

Default: *'high'*

'min_code_length': Minimal number of decoded characters. If a bar code is found with a code length below *'min_code_length'*, the result is discarded and not returned by `find_bar_code` or `decode_bar_code_rectangle2`. This parameter is useful to reduce the number of false reads in applications where the minimal number of characters is known in advance.

Note that the default value of this parameter is *0* for most code types. However, the default value is *3* for *'2/5 Industrial'* and *'2/5 Interleaved'* bar code types. The reason for this is that these bar code types are easily misread in other structures like clutter and text with a code length of *2*.

This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`.

Suggested values: *0*, *1*, *2*

Default:

Bar code Type	*'min_code_length'*
2/5 Industrial	3
2/5 Interleaved	3
all others	0

'max_code_length': Maximum number of decoded characters. If a bar code is found with a code length exceeding *'max_code_length'*, the result is discarded and not returned by `find_bar_code` or `decode_bar_code_rectangle2`. This parameter is useful to reduce the number of false reads in applications where the maximum number of characters is known in advance.

Value range: *1* ... *2147483647*

Default: *2147483647*

'merge_scanlines': If not enough scanlines (see *'min_identical_scanlines'*) can be read successfully, i.e., because the bar code is partly occluded or damaged, an attempt is being made to merge the existing scanlines. Then, these merged scanlines are decoded again. This additional decoding step is only performed for non-stacked bar code types and can be disabled for performance reasons.

List of values: *'true'*, *'false'*

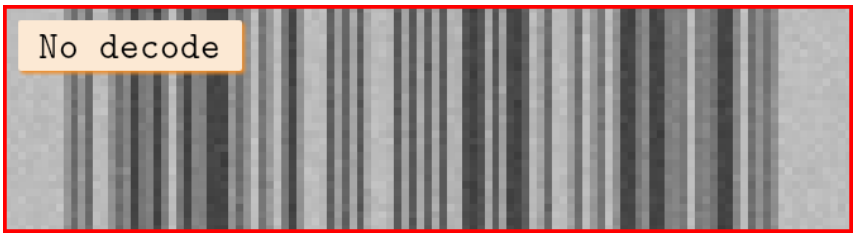
Default: *'true'*

'small_elements_robustness': If not enough scanlines (see *'min_identical_scanlines'*) can be read successfully, i.e., because the bar code contains small elements (smaller than *2.0* pixels in element size), an attempt is being made to read the code with an advanced algorithm which tries to solve the low resolution problem.

This additional decoding attempt is only performed for non-stacked bar code types and if the specified *'element_size_min'* is smaller than *2.0* pixels.

The status result of this attempt can be retrieved by calling the operator `get_bar_code_result` with *'status_small_elements_robustness'*.

In case of very small elements (smaller than *1.3* pixels in element size) it is beneficial if the camera sensor is rotated slightly with respect to the bar code (e.g., around 7 degrees). This is important because in such low resolution cases the bar code information is contained in multiple rows/columns of the perceived image only if it is rotated. See the following two figures for examples.



A bar code at *0.9* element size without sensor rotation. The relevant information of the bar code is not present in the image.

The same bar code as in the previous figure with sensor rotation. The rows of the image contain more information about the code and thus it can be decoded successfully.

Another important advice is to reduce the amount of blur introduced by the optical system.

The input image should not be preprocessed with filter operations such as `emphasize`, `zoom_image_size` or `mean_image`. Those operators would remove important information.

Note, if optical conditions are optimal (small amount of blur and slight rotation with respect to the bar code) it is possible to read bar codes as small as *0.6* pixels in element size.

This feature is only compatible with byte input images and can be disabled for performance reasons.

List of values: *'true'*, *'false'*
Default: *'true'*

Appearance of the bar code in the image:

'meas_thresh': *'meas_thresh'* defines a threshold which is a relative value with respect to the dynamic range of the scanline pixels. The bar-space-sequence of a bar code is determined with a scanline measuring the position of the edges. Finding these edges requires the mentioned threshold. In the case of disturbances in the bar code region or a high noise level, the value of *'meas_thresh'* should be increased.
Suggested values: [*0.05*, ..., *0.2*]
Value range: (*0.0*, ..., *1.0*] - without *0.0*
Default: *0.05*

'meas_thresh_abs': If a scanline is laid in an image region with no or just very small gray value dynamic range (e.g., in a white region with all gray values near 255), the edge detection threshold based on *'meas_thresh'* would be computed unreasonably small. This leads typically to the detection of a big amount of false edges. *'meas_thresh_abs'* is used to prevent such misdetections. If the threshold value based on *'meas_thresh'* gets smaller than the value of *'meas_thresh_abs'*, the latter is used as threshold instead. By default, *'meas_thresh_abs'* is set to *5.0*. A greater value might be more appropriate for images with high noise levels. On the other hand, in noise-free images with very weak contrast, this parameter might disturb the detection of real edges, so it might be necessary to reduce it or even completely disable it by setting it to *0.0*.
Suggested values: [*0.0*, ..., *10.0*]
Value range: [*0.0*, ..., *65535*]
Default: *5.0*

'contrast_min': Minimal contrast between the foreground and the background of the bar code elements. Setting this parameter to values greater than 5 will help the operator `find_bar_code` to optimize the candidate region search. `find_bar_code` will reject all candidate regions with a contrast value below *'contrast_min'*. Hence, setting a high *'contrast_min'* value will improve the runtime performance of `find_bar_code`. However, please note, that all bar codes with a contrast value below *'contrast_min'* will not be read. The calculated contrast value is an approximation in order to speed up the execution time. Try to set a lower threshold for *'contrast_min'* in order to find bar codes which might get rejected and have a contrast value that is close to the specified *'contrast_min'*.
Suggested values: *0*, *40*, *90*, *120*
Default: *0*

Values specific for print quality inspection:

'quality_isoiec15416_reflectance_reference': The reflectances used to assess the quality grades Symbol Contrast, Minimal Reflectance, and Minimal Edge Contrast is normalized with a reference gray value for reflectance. This value can be obtained with barium sulphate or magnesium oxide samples, i.a., see ISO/IEC 15416:2016 section 5.2. **Value range:** [*1* ... *255*]
Default: *255*

Values that only apply to certain code types:

'check_char': Bar codes with an optional check character are interpreted according to this parameter. These bar code types include, e.g., Code 39, Codabar, 2/5 Industrial and 2/5 Interleaved. The default setting of *'absent'* assumes that no check character is present. In this case, no check is performed and all characters are returned as data.
When set to *'present'*, a check character is expected and used to verify the correctness of the bar code. If the check sum matches, the check character itself is stripped from the data. If this stripping is undesired, the mode *'preserved'* allows to verify the bar code while still keeping the check character in the data. In the case the check sum does not match, no bar code result is returned.
For bar code types with a mandatory check character the parameter *'check_char'* is set to *'present'*. However, if the check character is removed from the data or stays, depends on the standard of the specific code type. Such code types include, e.g., Code 128, EAN-8, EAN-13, and UPC-A.
This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`.
List of values: *'absent'*, *'present'*, *'preserved'*
Default: *'absent'*

'composite_code': Most of the GS1 conform bar codes can have an additional 2D GS1 Composite code component appended. If *'composite_code'* is set to *'CC-A/B'* the composite component will be found and decoded. By default, *'composite_code'* is set to *'none'* and thus it is disabled. If the searched bar code symbol has no attached composite component, just the result of the bar code itself is returned by `find_bar_code`. Currently, composite codes are supported only for bar codes of the GS1 DataBar family.

This parameter can be set specifically for different types of bar codes by using the operator `set_bar_code_param_specific`.

List of values: *'none'*, *'CC-A/B'*

Default: *'none'*

'upce_encodation': For UPC-E bar codes, different output formats can be used. By default, *'upce_encodation'* is set to *'ucc-12'* and the decoded string will be returned in UCC-12 format (consisting of 12 digits). If *'upce_encodation'* is set to *'zero-suppressed'*, the result will be returned in zero-suppressed format (with suppressed zeros at defined places). This format consists of a leading zero (or a leading one, see *'upce1_enable'*), six encoded digits, and an implicitly encoded check digit. This corresponds to the format demanded by ISO/IEC 15420:2009 standard.

List of values: *'ucc-12'*, *'zero-suppressed'*

Default: *'ucc-12'*

'upce1_enable': The parameter *'upce1_enable'* enables the decoding of UPC-E bar codes with number system 1, if the parameter is set to *'true'*. UPC-E bar codes with number system 1 are an extension of UPC-E bar codes with number system 0, but are not documented in the underlying standard ISO/IEC 15420:2009. UPC-E bar codes with number system 1 also support the different output formats mentioned in *'upce_encodation'*, noting that the formats contain a leading one instead of a leading zero in UPC-E bar codes with number system 0.

List of values: *'true'*, *'false'*

Default: *'false'*

Training:

Besides setting the model parameters manually with `set_bar_code_param`, the model can also be trained with `find_bar_code` based on one or several sample images. For this, a bar code model must be created in training mode or an existing bar code model must be put into training mode by passing the generic parameter *'train'* in `GenParamName` of `set_bar_code_param`. The corresponding value passed in `GenParamValue` determines the model parameters that should be learned. The following values are possible:

'all': Train all possible model parameters.

'element_size_min': Minimal size of the base bar code elements. This setting activates the training mode for the parameter *'element_size_min'*.

'element_size_max': Maximal size of the base bar code elements. This setting activates the training mode for the parameter *'element_size_max'*.

'barcode_width_min': Minimal bar code width. This setting activates the training mode for the parameter *'barcode_width_min'*.

'barcode_width_max': Maximal bar code width. This setting activates the training mode for the parameter *'barcode_width_max'*.

'meas_thresh': Relative threshold for measuring the edge position within a scanline. This setting activates the training mode for the parameter *'meas_thresh'*.

'meas_thresh_abs': Absolute threshold for measuring the edge position within a scanline. This setting activates the training mode for the parameter *'meas_thresh_abs'*.

'orientation': Orientation of the bar code. This setting activates the training mode for the parameters *'orientation'* and *'orientation_tol'*. After ending the training the value for *'orientation_tol'* can be increased by a tolerance value to be able to find bar codes with orientations slightly beyond the trained orientations.

It is possible to train several of these parameters in one call of `find_bar_code` by passing the generic parameter *'train'* in `set_bar_code_param` in a tuple more than once in conjunction with the appropriate parameters: e.g., `GenParamName` = *['train','train']* and `GenParamValue` = *['element_size_min','element_size_max']*. Furthermore, in conjunction with *'train'* = *'all'* it is possible to exclude single parameters from training explicitly again by passing *'train'* more than once. The names of the parameters to exclude, however, must be prefixed by *'~'*: `GenParamName` = *['train','train']* and `GenParamValue` =

['all','~orientation'], e.g., trains all parameters except the orientation.

Afterwards, the operator find_bar_code has to be called for every image to be trained.

For training the model, the following aspects should be considered:

- Additionally needed, non-trained parameters (e.g., *'check_char'*) must be set manually before the training.
- To use several images for the training, the operator find_bar_code must be called once for every sample image.
- On a single training image there may be multiple bar codes of the selected code type visible.
- Please check the output of find_bar_code during training. Every decoded bar code will contribute to the trained bar code model. Hence, please make sure, that there are no false decodes. If you encounter false decodes, you may have to discard the image or set appropriate parameters in advance (e.g., you may need to increase the value of the parameter *'min_identical_scanlines'* or you may try to reduce the domain of the image appropriately).
- In an application with very similar images, *one* image for the training may be sufficient for the parameters *'element_size_min'*, *'element_size_max'*, *'meas_thresh'*, and *'meas_thresh_abs'*.

  For the training of the parameter *'orientation'*, at least two images are needed showing the bar code at its minimum and maximum orientations.
- In applications where the element size is not fixed, the smallest as well as the biggest symbols should be used for the training. If this can not be guaranteed, the limits for the element size should be adapted manually after the training or the element sizes should entirely be excluded from the training.
- During the first call of find_bar_code in the training mode, the trained model parameters are restricted to the properties of the detected bar code. Any successive training call will, where necessary, extend the parameter range to cover the already trained bar codes as well as the new bar code.
- The training can be stopped when enough images have been trained. Before this, the trained parameters should be retrieved. With the operator query_bar_code_params the names of the trained parameters - split into specific and general parameters - can be determined. The values of the parameters can then be retrieved with the operators get_bar_code_param_specific and get_bar_code_param.

  To stop the training, all trained parameters are excluded from the training by calling the operator set_bar_code_param with the parameters GenParamName = *'train'* and GenParamValue = *'~all'*.

  Alternatively, the existing bar code model can simply be cleared with the operator clear_bar_code_model.
- The value of any trained parameter should not be set explicitly with set_bar_code_param or set_bar_code_param_specific during training mode. If this is done, the complete internal training data is being reset and training will be started again at the beginning with the next call of find_bar_code.
- If find_bar_code is not able to read the bar code in the training image, this will produce no error or exception handling. It can simply be tested in the program by checking the output parameter DecodedDataStrings or SymbolRegions. These tuples will then be empty and the parameters of the model (especially the trained parameters) will not be modified.
- Timeouts are disabled during the training.

Miscellaneous:

'timeout': By the use of this parameter, it is possible to abort find_bar_code after a defined period in milliseconds. This is especially useful in cases where a maximum cycle time has to be ensured. All results gained before the timeout can be accessed by get_bar_code_result. Passing values less or equal zero implies a deactivation of the timeout (default).

The temporal accuracy of this timeout is about 10 ms. It depends on several factors including the speed of your computer and the *'timer_mode'* set via set_system.

find_bar_code does not raise an exception if a timeout occurs. To check whether find_bar_code has been aborted, check the parameter *'aborted'* in get_bar_code_result.

Note that the timeout is ignored if find_bar_code runs in training mode.

Suggested values: *'false'*, *-1*, *20*, *100*

Default: *'false'*

'abort': Using this option, it is possible to abort `find_bar_code` from another thread. When `set_bar_code_param` is called with *'abort'*, an instance of `find_bar_code` with the model `BarCodeHandle` running in another thread is requested to abort. If there is no `find_bar_code` running with this model, nothing happens.

The operator `find_bar_code` might not return immediately. It has to reach a cancellation point to ensure a proper cleanup. Depending on different factors like the computer performance this may take up to 10 ms.

All decoded results until this moment, are still returned. Note that the parameter is ignored if `find_bar_code` runs in training mode.

Note: This is the only action with a bar code handle, which can be used from different threads without requiring additional synchronization.

Default: *'true'* (The value is not processed.)

'persistence': Setting the model in persistence mode to *1*, makes it store some intermediate results during bar code decoding. These results are required if the bar code print quality has to be assessed (`get_bar_code_result` with *'quality_isoiec15416'*) or the decoding scanlines have to be inspected (`get_bar_code_object` with *'scanlines_all'* or *'scanlines_valid'*). Yet, enabling the *'persistence'* mode results in increased memory requirements for the bar code model structures.

List of values: *0*, *1*

Default: *0*

Parameters

▷ **BarCodeHandle** (input_control) . . . . . barcode ⇝ *handle*
Handle of the bar code model.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that shall be adjusted for finding and decoding bar codes.
Default: 'element_size_min'
List of values: `GenParamName` ∈ {'check_char', 'composite_code', 'barcode_height_min', 'barcode_width_min', 'barcode_width_max', 'element_size_max', 'element_size_min', 'contrast_min', 'meas_thresh', 'meas_thresh_abs', 'min_identical_scanlines', 'majority_voting', 'num_scanlines', 'orientation', 'orientation_tol', 'persistence', 'quality_isoiec15416_reflectance_reference', 'start_stop_tolerance', 'stop_after_result_num', 'upce_encodation', 'upce1_enable', 'timeout', 'train', 'quiet_zone', 'element_size_variable', 'min_code_length', 'max_code_length', 'merge_scanlines', 'small_elements_robustness', 'abort'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
Values of the generic parameters that are adjusted for finding and decoding bar codes.
Default: 8
Suggested values: `GenParamValue` ∈ {0, 0.1, 1, 1.5, 2, 8, 32, 45, 'true', 'false', 'present', 'absent', 'none', 'CC-A/B', 'auto', 'high', 'low', 'ucc-12', 'zero-suppressed'}

Result

The operator `set_bar_code_param` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- BarCodeHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_bar_code_model`

Possible Successors

`find_bar_code`

Alternatives

set_bar_code_param_specific

Module

Bar Code

set_bar_code_param_specific (: : BarCodeHandle, CodeTypes, GenParamName, GenParamValue :)

Set selected parameters of the bar code model for selected bar code types

The operator set_bar_code_param_specific is used to set or change the different parameters of a bar code model for selected bar code types in order to adapt to special properties of the bar codes. Contrary to set_bar_code_param, only parameters for selected bar code types will be changed with set_bar_code_param_specific. This is useful when searching different code types within one image (see section Autodiscrimination in find_bar_code).

The current configuration of the bar code model for a specified bar code type can be queried with get_bar_code_param_specific.

The following generic parameters can be set for specific bar code types:

'num_scanlines': Maximal number of scanlines used during the scanning of a (candidate) bar code.

Suggested values: [*0*, *5*, *10*, *20*, ...]

Default: *0*

'min_identical_scanlines': Minimal number of decoded scanlines which return identical data to read the bar code successfully.

Suggested values: [*0*, *2*, *3*, ...]

Default:

Bar code Type	*'min_identical_scanlines'*
2/5 Industrial	2
2/5 Interleaved	2
all others	0

'stop_after_result_num': Number of successfully decoded bar codes of the given code type after which the decoding will stop.

Suggested values: [*0*, *1*, *2*, ...]

Default: *0*

'orientation': Expected bar code orientation.

Suggested values: [*-90.0* ... *90.0*]

Default: *0.0*

'orientation_tol': Orientation tolerance.

Suggested values: [*0.0* ... *90.0*]

Default: *90.0*

'quiet_zone': Enforces the verification of quiet zones of a bar code symbol.

Suggested values: *'false'*, *'true'*, *1*, *2*, *3*, *4*, *5*

Default: *'false'*

'start_stop_tolerance': Start/stop search criteria tolerance. Currently this is implemented only for Code 128 and GS1-128.

List of values: *'high'*, *'low'*

Default: *'high'*

'check_char': Handling of an optional check character.

List of values: *'absent'*, *'present'*, *'preserved'*

Default: *'absent'*

'composite_code': Find and decode a composite component.

List of values: *'none'*, *'CC-A/B'*

Default: *'none'*

'min_code_length': Minimal number of decoded characters.

Default:

Bar code Type	*'min_code_length'*
2/5 Industrial	3
2/5 Interleaved	3
all others	0

'max_code_length': Maximum number of decoded characters.

Value range: [*1* ... *2147483647*]

Default: *2147483647*

Further details on these parameters can be found with the description of the operator `set_bar_code_param`.

It is possible to supply multivalue tuples for `CodeTypes`, `GenParamName`, and `GenParamValue`. The n-th entry of one multivalue tuple corresponds to the n-th entry of the other multivalue tuple. The following combinations are allowed:

CodeTypes	**GenParamName**	**GenParamValue**
single-valued	single-valued	single-valued
single-valued	multi-valued	multi-valued
multi-valued	single-valued	multi-valued
multi-valued	single-valued	single-valued

Parameters

▷ **BarCodeHandle** (input_control) ..... barcode ⇝ *handle*
Handle of the bar code model.

▷ **CodeTypes** (input_control) ..... string(-array) ⇝ *string*
Names of the bar code types for which parameters should be set.
Default: 'EAN-13'
List of values: `CodeTypes` ∈ {'2/5 Industrial', '2/5 Interleaved', 'Codabar', 'Code 39', 'Code 93', 'Code 128', 'EAN-13', 'EAN-13 Add-On 2', 'EAN-13 Add-On 5', 'EAN-8', 'EAN-8 Add-On 2', 'EAN-8 Add-On 5', 'UPC-A', 'UPC-A Add-On 2', 'UPC-A Add-On 5', 'UPC-E', 'UPC-E Add-On 2', 'UPC-E Add-On 5', 'MSI', 'PharmaCode', 'GS1 DataBar Omnidir', 'GS1 DataBar Truncated', 'GS1 DataBar Stacked', 'GS1 DataBar Stacked Omnidir', 'GS1 DataBar Limited', 'GS1 DataBar Expanded', 'GS1 DataBar Expanded Stacked', 'GS1-128'}

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Names of the generic parameters that shall be adjusted for finding and decoding bar codes.
Default: 'check_char'
List of values: `GenParamName` ∈ {'check_char', 'composite_code', 'min_identical_scanlines', 'num_scanlines', 'orientation', 'orientation_tol', 'start_stop_tolerance', 'stop_after_result_num', 'quiet_zone', 'min_code_length', 'max_code_length'}

▷ **GenParamValue** (input_control) ..... attribute.name-array ⇝ *real* / integer / string
Values of the generic parameters that are adjusted for finding and decoding bar codes.
Default: 'absent'
Suggested values: `GenParamValue` ∈ {0, 1, 2, 4, 45, 90, 'true', 'false', 'present', 'absent', 'none', 'CC-A/B', 'high', 'low'}

Result

The operator `set_bar_code_param_specific` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- BarCodeHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_bar_code_model

Possible Successors

find_bar_code

Alternatives

set_bar_code_param

Module

Bar Code

write_bar_code_model (: : BarCodeHandle, FileName :)

Write a bar code model to a file.

The operator write_bar_code_model writes the bar code model BarCodeHandle to the file FileName. The model can be read again with read_bar_code_model. The stored data contains all generic and all specific model parameters (see set_bar_code_param and set_bar_code_param_specific, respectively). If the model is in training mode (see set_bar_code_param with parameter *'train'*), the current training state is stored as well. After restoring the model, the training can be proceeded.

Besides the training state no other results of find_bar_code are stored in the file.

The default HALCON file extension for bar code model is 'bcm'.

Parameters

▷ **BarCodeHandle** (input_control) ..... barcode ⇝ *handle*
Handle of the bar code model.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Name of the bar code model file.
Default: 'bar_code_model.bcm'
File extension: .bcm

Example

```
*
* Create the bar code model in the training mode
create_bar_code_model ('train', 'all', BarCodeHandle)
*
* Set all additional, non-trained parameters in advance:
* Here, we specify that the training images have check characters
set_bar_code_param (BarCodeHandle, 'check_char', 'present')
*
* Train the model with several images
for I := 1 to 7 by 1
    FileName := 'barcode/25interleaved/25interleaved' + I$'.02'
    read_image (Image, FileName)
    *
    * Apply the training
    find_bar_code (Image, SymbolRegion, BarCodeHandle, '2/5 Interleaved', \
```

```
                    DecodedDataStrings)
endfor
*
* The training may be interrupted and the intermediate state
* of the model can be stored in a file
write_bar_code_model (BarCodeHandle, 'bar_code_model.bcm')
*
* RESTORE TRAINING:
* Later, when, e.g., new images are available, the training
* may be restored
read_bar_code_model ('bar_code_model.bcm', BarCodeHandle)
FileName := 'barcode/25interleaved/25interleaved08'
read_image (Image, FileName)
*
* Apply the training to the new image
find_bar_code (Image, SymbolRegion, BarCodeHandle, '2/5 Interleaved', \
               DecodedDataStrings)
*
* Finally, the training can be completed
set_bar_code_param (BarCodeHandle, 'train', '~all')
*
* The trained model can be stored for ONLINE use
write_bar_code_model (BarCodeHandle, 'trained_bar_code_model.bcm')
*
* ONLINE USE:
read_bar_code_model ('trained_bar_code_model.bcm', BarCodeHandle)
* ...
```

Result

The operator `write_bar_code_model` returns the value `2` (`H_MSG_TRUE`) if the passed handle is valid and if the model can be written into the named file. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_bar_code_model`, `set_bar_code_param`

See also

`create_bar_code_model`, `set_bar_code_param`, `find_bar_code`

Module

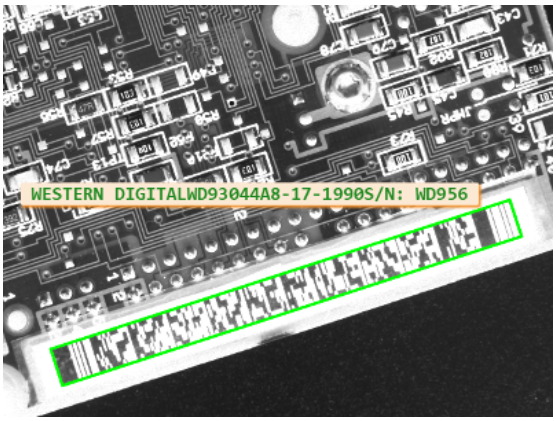
Bar Code

14.2 Data Code

This chapter contains operators for reading 2D data codes.

Concept of reading 2D data codes

2D data code symbols are a special kind of two-dimensional patterns that encode text and numbers. HALCON is able to read the most popular 2D data codes: Data Matrix ECC 200, QR Code, Micro QR Code, Aztec Code, PDF417, and DotCode. Except for DotCodes all these codes contain a finder pattern and a data pattern. The finder pattern is used to locate the pattern of the symbol and get basic information about the geometric properties, e.g., the orientation of the symbol. The data pattern contains the code itself and consists of multiple dots, bars, or small squares, the so-called modules. Because of the special design of the codes, they can be decoded even if some parts are disturbed.



Reading a 2D data code of type PDF417. This image is from the example program 2d_data_codes_default_settings.hdev.

In the following, the steps that are required to read 2D data codes are described briefly.

Create 2D data code model: First, a 2D data code model must be created using

- `create_data_code_2d_model`.

This model provides the reader with all necessary information about the structure of the code. For normal printed codes only the name needs to be provided and HALCON will select suitable default parameters. For special cases, you can modify the model parameters either when creating the 2D data code model or in a later step to adapt the model to a particular symbol appearance.

Modify the model parameters for non-standard codes: Using the default parameters, the 2D data code reader is able to read a wide range of codes. For non-standard codes the parameters can be modified using

- `set_data_code_2d_param`.

Here, you can either select an enhanced set of default parameters using the generic parameter *'default_parameters'*, e.g., with the value *'enhanced_recognition'*, or you specify the parameter values separately to adapt the model optimally to the conditions of the used print style. Note that `query_data_code_2d_params` can be used to query the parameters that are valid for the specific data code type. To obtain the currently set values of parameters, `get_data_code_2d_param` can be used.

Instead of modifying the model parameters manually, you can also let HALCON train the model using

- `find_data_code_2d`

with the generic parameter *'train'*. Then, HALCON will search for the best parameters needed to extract the given code. It is recommended to apply this to multiple example images to ensure that all variations are covered.

Read 2D data code: The 2D data code is located and its content is decoded using

- `find_data_code_2d`.

The operator returns for every successfully decoded symbol the surrounding XLD contour, a handle to a result structure, which contains additional information about the symbol as well as about the search and decoding process, and the string that is encoded in the symbol. With the result handles and the operators `get_data_code_2d_results` and `get_data_code_2d_objects`, additional data about the extraction process can be accessed that can be used both for process analysis and for displaying. In particular, `get_data_code_2d_results` allows to access several alphanumerical results that were calculated while searching and reading the symbols and `get_data_code_2d_objects` allows to access iconic objects that were created during the last call of `find_data_code_2d`.

Further operators

In addition to the operators mentioned above, write_data_code_2d_model allows to write the model into a file that can be used later to create (e.g., in a different application) an identical copy of the model. Such a model copy is created directly by read_data_code_2d_model (without calling create_data_code_2d_model). Furthermore, you can use serialize_data_code_2d_model and deserialize_data_code_2d_model to serialize and deserialize the 2D data code model.

Glossary

2D data code symbol Two-dimensional graphical symbol that encodes characters and numbers. It is constructed by dark and light dots, bars, or small squares that are called modules. There are different types of 2D data codes. Two common types are the so-called stacked codes and the so-called matrix codes.

stacked code Type of 2D data code symbol that contains a stack of 1D bar codes arranged in rows and columns. To ensure that the complete stack of 1D bar codes is processed, the symbol contains a start and a stop pattern. Additionally, the symbol is framed by a quiet zone.

matrix code Type of 2D data code symbol that uses graphical patterns consisting of dark and light modules arranged in two dimensions. The symbol consists of three components: a finder pattern, a data pattern, and a quiet zone.

modules Dark and light dots, bars, or small squares that are used to build a 2D data code symbol.

quiet zone Homogeneous frame around the symbol's border that makes the symbol better distinguishable from the background or from other objects in the image.

finder pattern Pattern that is used to find the symbol and its orientation in the image. The pattern differs depending on the used data code type.

Further Information

See also the "Solution Guide Basics" and the "Solution Guide on 2D Data Codes" for further details about 2D data codes.

clear_data_code_2d_model (: : DataCodeHandle :)

Delete a 2D data code model and free the allocated memory.

The operator clear_data_code_2d_model deletes a 2D data code model that was created by create_data_code_2d_model or read_data_code_2d_model. All memory used by the model is freed. The handle of the model is passed in DataCodeHandle. After the operator call it is invalid.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

Parameters

▷ **DataCodeHandle** (input_control) ..... datacode_2d ⇝ *handle*
Handle of the 2D data code model.

Result

The operator clear_data_code_2d_model returns the value 2 (H_MSG_TRUE) if a valid handle was passed and the referred 2D data code model can be freed correctly. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DataCodeHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

create_data_code_2d_model, read_data_code_2d_model

Module

Data Code

create_data_code_2d_model (: : SymbolType, GenParamName, GenParamValue : DataCodeHandle)

Create a model of a 2D data code class.

The operator create_data_code_2d_model creates a model for a certain class of 2D data codes. In DataCodeHandle the operator returns a handle to the 2D data code model, which is used for all further operations on the data code, like modifying the model, reading a symbol, or accessing the results of the symbol search.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

Supported symbol types

The parameter SymbolType is used to determine the type of data codes to process. Presently, six main types are supported: *'Data Matrix ECC 200'*, *'QR Code'*, *'Micro QR Code'*, *'PDF417'*, *'Aztec Code'*, and *'DotCode'*. Additionally, four GS1 types are supported: *'GS1 DataMatrix'*, *'GS1 QR Code'*,*'GS1 Aztec Code'*, and *'GS1 DotCode'*. Data Matrix codes of type ECC 000-140 are not supported. For the QR Code the older Model 1 as well as the new Model 2 can be read. The PDF417 can be read in its conventional as well as in its compact form ('Compact/Truncated PDF417'). The Aztec code can be read in compact, full-range, and rune (see Annex E of ISO/IEC 24778:2008 (E)) form. The structure of the four GS1 symbologies is basically identical to their non-GS1 counterparts - Data Matrix ECC 200, QR Code Aztec Code, and DotCode, respectively. In the following, all parameters, settings, and rules applying to any particular data code type apply to its GS1 variant as well. The GS1 symbologies enforce merely additional rules for the format of the data carried by the codes. It has to be organized in so called GS1 application element strings according to the GS1 General Specifications. To update the list of available GS1 Application Identifiers, see the parameter *'gs1_syntax_dictionary'* in set_system.

For all symbol types, the data code reader supports the Extended Channel Interpretation (ECI) protocol. If the symbol contains an ECI code, all characters with ASCII code 92 (backslash, '\') that occur in the normal data stream are, in compliance with the standard, doubled ('\\') for the output. This is necessary in order to distinguish data backslashes from the ECI sequence *'\nnnnnn'*.

The information whether the symbol contains ECI codes (and consequently doubled backslashes) or not is stored in the Symbology Identifier number that can be obtained for every successfully decoded symbol with the help of the operator get_data_code_2d_results passing the generic parameter *'symbology_ident'*. How the code number encodes additional information about the symbology and the data code reader, like the ECI support, is defined in the different symbology specifications. For more information see the corresponding standards and the operator get_data_code_2d_results.

The Symbology Indentifier code will not be preceded by the data code reader to the output data, even if the symbol contains an ECI code. If this is needed, e.g., by a subsequent processing unit, the *'symbology_ident'* number (obtained by the operator get_data_code_2d_results with the parameter *'symbology_ident'*) can be added to the data stream manually together with the symbology flag and the symbol code: ']d', ']Q', ']L', ']z', or ']J' for DataMatrix codes, QR Codes, PDF417 Codes, Aztec Codes, or DotCodes, respectively. In particular, for GS1 symbologies the parameter *'symbology_ident'* returns 2 for GS1 DataMatrix, 3 for GS1 QR Code, and 1 for GS1 Aztec Code and GS1 DotCode. This corresponds to symbol codes: ']d2', ']Q3', ']z1', and ']J1', respectively.

Standard default settings of the data code model

The default settings of the model were chosen to read a wide range of common symbols within a reasonable amount of time. However, for runtime reasons some restrictions apply to the symbol (see the following table). If the model was modified (as described later), it is at any time possible to reset it to these default settings by passing the generic parameter *'default_parameters'* together with the value *'standard_recognition'* to the operator set_data_code_2d_param.

For some model parameters the default values depend on the chosen recognition approach. Those parameters are listed in the table below. Further parameter details and the remaining model parameters can be found in the reference of set_data_code_2d_param.

Model parameter	*'standard_recognition'*	*'enhanced_recognition'*	*'maximum_recognition'*
'polarity'	'dark_on_light' (dark symbols on a light background)	'any' (dark symbols on a light background and vice versa)	'any' (dark symbols on a light background and vice versa)
'contrast_min' (Aztec Code, Micro QR Code, QR Code, PDF417)	30	10	10
'symbol_cols_min', *'symbol_cols_max'* (PDF417)	1 ... 20	1 ... 30	1 ... 30
'symbol_rows_min', *'symbol_rows_max'* (PDF417)	5 ... 45	3 ... 90	3 ... 90
'module_size_min', *'module_size_max'* (Aztec Code, Data Matrix ECC 200, Micro QR Code, QR Code)	6 ... 20 pixels	4 (2 for sharp images) ... 100 pixels	4 (1 for sharp images) ... 100 pixels
'module_size_min', *'module_size_max'* (DotCode)	4 ... 100 pixels	2 ... 100 pixels	2 ... 100 pixels
'module_width_min', *'module_width_max'* (PDF417)	3 ... 15 pixels	3 (2 for sharp images) ... 100 pixels	3 (1 for sharp images) ... 100 pixels
'module_aspect_min', *'module_aspect_max'* (PDF417)	1 ... 4	1 ... 10	1 ... 10
'small_modules_robustness' (All code types except DotCode)	'low'	'low'	'high'
'module_gap_max' (Aztec Code)	'small' (< 10% of the module size)	'big' (< 50% of the module size)	'big' (< 50% of the module size)
'module_gap_max' (Data Matrix ECC 200, Micro QR Code, QR Code)	'no'	'small' (< 10% of the module size)	'big' (< 50% of the module size)
'module_gap_max' (DotCode)	'no'	'small' (< 10% of the module size)	'small' (< 10% of the module size)
'slant_max' (Data Matrix ECC 200)	0.1745 (10 degree)	0.5235 (30 degree)	0.5235 (30 degree)
'module_grid' (Data Matrix ECC 200)	'fixed'	'any' (fixed or variable)	'any' (fixed or variable)
'finder_pattern_tolerance' (Aztec Code)	'low'	'high'	'high'
'finder_pattern_tolerance' (Data Matrix ECC 200)	'low'	'low'	'any'
'alternating_pattern_tolerance' (Data Matrix ECC 200)	'low'	'medium'	'high'
'contrast_tolerance' (Data Matrix ECC 200, Micro QR Code, QR Code)	'low'	'low'	'any'
'candidate_selection' (Data Matrix ECC 200)	'default'	'extensive'	'extensive'
'candidate_selection' (DotCode, Micro QR Code, QR Code)	'default'	'extensive'	'all'
'position_pattern_min' (QR Code)	3	2	2
'model_type' (QR Code)	2	2	'any'

Modify the data code model

If it is known that the symbol does not or may not comply with all of these restrictions (e.g., the symbol is brighter than the background or the contrast is very low), or if first tests show that some of the symbols cannot be read with the default settings, it is possible to adapt single model parameters – while others are kept to the default – or the whole model can be extended in a single step by setting the generic parameter *'default_parameters'* to the value *'enhanced_recognition'*. This will lead to a more general model that covers a wider range of 2D data code symbols. However, the symbol search with such a general model is more extensive, hence the runtime of the operator `find_data_code_2d` may increase significantly. This is true especially in the following cases: *no* readable data code is detected, the symbol is printed light on dark, or the modules are very small.

By setting the generic parameter *'default_parameters'* to the value *'maximum_recognition'* the model is further extended in comparison to *'enhanced_recognition'*, so that data codes with very small module sizes can be decoded more robustly (see *'small_modules_robustness'*). For the Data Matrix ECC 200, the mode *'maximum_recognition'* differs from the mode *'enhanced_recognition'* in a way that even symbols with defect or partially occluded finder patterns can be read (see *'finder_pattern_tolerance'*). Using this mode may lead to a further increased runtime and memory usage of the operator `find_data_code_2d` during symbol search and decoding.

For these reasons, the model should always be specified as exactly as possible by setting all known parameters. The model parameters can be set directly during the creation of the model or later with the help of the operator `set_data_code_2d_param`. Both operators provide the generic parameters `GenParamName` and `GenParamValue` for this purpose. For the Data Matrix ECC 200, for example, the symbol size should be specified as exactly as possible if the mode *'maximum_recognition'* was chosen. A detailed description of all supported generic parameters can be found with the operator `set_data_code_2d_param`.

Another way for adapting the model is to train it based on sample images. Passing the parameter *'train'* to the operator `find_data_code_2d` will cause the find operator to look for a symbol, determine its parameters, and modify the model accordingly. More details can be found with the description of the operator `find_data_code_2d`.

It is possible to query the model parameters with the operator `get_data_code_2d_param`. The names of all supported parameters for setting or querying the model are returned by the operator `query_data_code_2d_params`.

Parameters

▷ **`SymbolType`** (input_control) . . . . . string ⇝ *string*
Type of the 2D data code.
Default: 'Data Matrix ECC 200'
List of values: `SymbolType` ∈ {'Data Matrix ECC 200', 'QR Code', 'Micro QR Code', 'PDF417', 'Aztec Code', 'DotCode', 'GS1 DataMatrix', 'GS1 QR Code', 'GS1 Aztec Code', 'GS1 DotCode'}

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that can be adjusted for the 2D data code model.
Default: []
List of values: `GenParamName` ∈ {'default_parameters', 'strict_model', 'persistence', 'polarity', 'mirrored', 'contrast_min', 'candidate_selection', 'model_type', 'version', 'version_min', 'version_max', 'symbol_size', 'symbol_size_min', 'symbol_size_max', 'symbol_cols', 'symbol_cols_min', 'symbol_cols_max', 'symbol_rows', 'symbol_rows_min', 'symbol_rows_max', 'symbol_shape', 'module_size', 'module_size_min', 'module_size_max', 'small_modules_robustness', 'module_width', 'module_width_min', 'module_width_max', 'module_aspect', 'module_aspect_min', 'module_aspect_max', 'module_gap', 'module_gap_min', 'module_gap_max', 'slant_max', 'module_grid', 'position_pattern_min', 'strict_quiet_zone', 'timeout', 'finder_pattern_tolerance', 'contrast_tolerance', 'additional_levels'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Values of the generic parameters that can be adjusted for the 2D data code model.
Default: []
Suggested values: `GenParamValue` ∈ {'standard_recognition', 'enhanced_recognition', 'maximum_recognition', 'yes', 'no', 'any', 'dark_on_light', 'light_on_dark', 'square', 'rectangle', 'small', 'big', 'fixed', 'variable', 'low', 'high', 'default', 'extensive', 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 30, 50, 70, 90, 12, 14, 16, 18, 20, 22, 24, 26, 32, 36, 40, 44, 48, 52, 64, 72, 80, 88, 96, 104, 120, 132, 144}

▷ **`DataCodeHandle`** (output_control) . . . . . datacode_2d ⇝ *handle*
Handle for using and accessing the 2D data code model.

Example

```
* Two simple examples that show the use of create_data_code_2d_model
* to detect a Data matrix ECC 200 code and a QR Code.
```

```
* (1) Create a model for reading simple QR Codes
*     (only dark symbols on a light background will be read)
create_data_code_2d_model ('QR Code', [], [], DataCodeHandle)
* Read an image
read_image (Image, 'datacode/qrcode/qr_workpiece_01')
* Read the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], \
                   ResultHandles, DecodedDataStrings)
* Clear the model
clear_data_code_2d_model (DataCodeHandle)

* (2) Create a model for reading a wide range of Data matrix ECC 200 codes
*     (this model will also read light symbols on dark background)
create_data_code_2d_model ('Data Matrix ECC 200', 'default_parameters', \
                           'enhanced_recognition', DataCodeHandle)
* Read an image
read_image (Image, 'datacode/ecc200/ecc200_cpu_010')
* Read the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], \
                   ResultHandles, DecodedDataStrings)
* Clear the model
clear_data_code_2d_model (DataCodeHandle)
```

Result

The operator `create_data_code_2d_model` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`set_data_code_2d_param`, `find_data_code_2d`

Alternatives

`read_data_code_2d_model`

See also

`clear_data_code_2d_model`

References

International Standard ISO/IEC 16022: “Information technology - Automatic identification and data capture techniques - Data Matrix bar code symbology specification”; Reference number ISO/IEC 16022:2006 (E); ISO/IEC 2006.

International Standard ISO/IEC 15438: “Information technology - Automatic identification and data capture techniques - PDF417 bar code symbology specification”; Reference number ISO/IEC 15438:2006 (E); ISO/IEC 2006.

International Standard ISO/IEC 18004: “Information technology - Automatic identification and data capture techniques - QR Code 2005 bar code symbology specification”; Reference number ISO/IEC 18004:2006 (E); ISO/IEC 2006.

International Standard ISO/IEC 24778: “Information technology - Automatic identification and data capture techniques - Aztec Code bar code symbology specification”; Reference number ISO/IEC 24778:2008 (E); ISO/IEC 2008.

GS1 General Specifications; Version 12; Issue 1, Jan-2012; GS1.

Module

Data Code

deserialize_data_code_2d_model (
: : SerializedItemHandle : DataCodeHandle)

Deserialize a serialized 2D data code model.

deserialize_data_code_2d_model deserializes a 2D data code model, that was serialized by serialize_data_code_2d_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized 2D data code model is defined by the handle SerializedItemHandle. The deserialized values are stored in a new 2D data code model with the handle DataCodeHandle.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **DataCodeHandle** (output_control) ..... datacode_2d ⇝ *handle*
Handle of the 2D data code model.

Result

The operator deserialize_data_code_2d_model returns the value 2 (H_MSG_TRUE) if the 2D data code can be correctly deserialized. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_data_code_2d_model

Possible Successors

find_data_code_2d

Alternatives

create_data_code_2d_model

See also

serialize_data_code_2d_model, clear_data_code_2d_model

Module

Data Code

find_data_code_2d (Image : SymbolXLDs : DataCodeHandle,
GenParamName, GenParamValue : ResultHandles, DecodedDataStrings)

Detect and read 2D data code symbols in an image or train the 2D data code model.

The operator find_data_code_2d detects 2D data code symbols in the input image (Image) and reads the data that is encoded in the symbol. Before calling find_data_code_2d, a model of a class of 2D data codes that matches the symbols in the images must be created with create_data_code_2d_model or read_data_code_2d_model. The handle returned by these operators is passed to find_data_code_2d in DataCodeHandle. To look for more than one symbol in an image, the generic parameter *'stop_after_result_num'* can be passed to GenParamName together with the number of requested symbols as GenParamValue.

As a result the operator returns for every successfully decoded symbol the surrounding XLD contour (SymbolXLDs), a result handle, which refers to a candidate structure that stores additional information about the symbol as well as the search and decoding process (ResultHandles), and the string that is encoded in the symbol (DecodedDataStrings). Passing the candidate handle from ResultHandles together with the generic parameter *'decoded_data'*, get_data_code_2d_results returns a tuple with the ASCII code of all characters of the string (see also Tuple / String Operations).

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

The symbol structure of GS1 DataMatrix, GS1 QR Code, GS1 Aztec Code, and GS1 DotCode is identical to the structure of Data Matrix ECC 200, QR Code, Aztec Code, and DotCode, respectively. Therefore, all parameters, settings, and rules applying to Data Matrix ECC 200, QR Code, Aztec Code, or DotCode apply to their GS1 variants as well. The GS1 symbologies enforce merely additional rules for the format of the data carried by the codes. The data has to be organized in so called GS1 application element strings according to the GS1 General Specifications. For example, if the data code model `DataCodeHandle` is created as GS1 DataMatrix, then `find_data_code_2d` only returns a result if the underlying symbol is a valid Data Matrix ECC 200 symbol, and only if its data conforms to the GS1 application element strings rules. For the GS1 application element strings it is only checked if the application identifiers exist in the GS1 General Specifications. In contrast, `find_data_code_2d` will return no results for Data Matrix ECC 200 containing general data. The same is valid for GS1 QR Code, GS1 Aztec Code, and GS1 DotCode Code.

Adjusting the model

If there is a symbol in the image that cannot be read, it should be verified, whether the properties of the symbol fit the model parameters. Special attention should be paid to

- the correct polarity (*'polarity'*, light-on-dark or dark-on-light),
- the symbol size (*'symbol_size'* for Data Matrix ECC 200, Aztec Code, and DotCode),
- *'version'* for QR Code,
- *'format'* for Aztec Code,
- the module size (*'module_size'* for Data Matrix ECC 200, QR Code, Micro QR Code, Aztec Code, and DotCode, *'module_width'* and *'module_aspect'* for PDF417),
- the possibility of a mirroring of the symbol (*'mirrored'*),
- and the specified minimum contrast (*'contrast_min'* for Aztec Code, QR Code, and PDF417).

Further relevant parameters are the gap between neighboring foreground modules and, for Data Matrix ECC 200, the maximum slant of the L-shaped finder pattern (*'slant_max'*). The current settings for these parameters are returned by the operator `get_data_code_2d_param`. If necessary, the corresponding model parameters can be adjusted with `set_data_code_2d_param`.

It is recommended to adjust the model as well as possible to the symbols in the images also for runtime reasons. In general, the runtime of `find_data_code_2d` is higher for a more general model than for a more specific model. One should take into account that a general model leads to a high runtime especially if *no* valid data code can be found.

Train the model

Besides setting the model parameters manually with `set_data_code_2d_param`, the model can also be trained with `find_data_code_2d` based on one or several sample images. For this, the generic parameter *'train'* must be passed in `GenParamName`. The corresponding value passed in `GenParamValue` determines the model parameters that should be learned. The following values are possible:

- All data code types:

  'all': All model parameters that can be trained.

  'symbol_size': Size of the data code symbol.
  The model parameters trained for Data Matrix ECC200, PDF417, and DotCode are *'symbol_cols_min'*, *'symbol_cols_max'*, *'symbol_rows_min'*, and *'symbol_rows_max'*. Additionally, the model parameter *'symbol_shape'* (square or rectangular) is trained for Data Matrix ECC 200 only.
  For QR Code and Micro QR Code it is also possible to pass *'version'*. The model parameters trained hereby are *'symbol_size_min'*, *'symbol_size_max'*, *'version_min'*, and *'version_max'*.
  The model parameters trained for Aztec Code are *'symbol_size_min'* and *'symbol_size_max'*.

  'module_size': Size of the modules. The model parameters trained are *'module_size_min'* and *'module_size_max'*.
  For PDF417 this includes the module width and the module aspect ratio, so thereby the respective model parameters are *'module_width_min'*, *'module_width_max'*, *'module_aspect_min'*, and *'module_aspect_max'*.

  'polarity': Polarity of the symbols, i.e. they may appear dark on a light background or light on a dark background. The model parameter trained is *'polarity'*.

'mirrored': Whether the symbols in the image are mirrored or not. The model parameter trained is *'mirrored'*.

- Data Matrix ECC 200, Aztec Code, PDF417, QR Code, and Micro QR Code only:

  'small_modules_robustness': Robustness of the decoding of data codes with very small module sizes. This parameter is not trained for DotCode. The model parameter trained is *'small_modules_robustness'*. Note that the training of the parameter is deactivated internally if the training image exceeds the maximum image size (see `get_system` *'halcon_xl'*).

- Aztec Code, PDF417, QR Code, and Micro QR Code only:

  'contrast': Minimum contrast for detecting the symbols. The model parameter trained is *'contrast_min'*.

- Data Matrix ECC 200, Micro QR Code, and QR Code only:

  'contrast_tolerance': The tolerance of the symbol search with respect to strong local contrast variations. The model parameter trained is *'contrast_tolerance'*.

- All data code types except PDF417::

  'module_shape': Whether there is a gap between neighboring foreground modules or whether they are connected. The model parameters trained are *'module_gap_min'* and *'module_gap_max'*.

- Data Matrix ECC 200 and Aztec Code only:

  'finder_pattern_tolerance': The allowed tolerance of the symbol search with respect to a defect or partially occluded finder pattern. The model parameter trained is *'finder_pattern_tolerance'*.

- Data Matrix ECC 200, DotCode, Micro QR Code, and QR Code only:

  'candidate_selection': Selection of candidate regions to be processed. The model parameter trained is *'candidate_selection'*.

- Data Matrix ECC 200 only:

  'alternating_pattern_tolerance': The tolerance of the symbol search with respect to a variation of the module widths along the two sides with the alternating pattern. The model parameter trained is *'alternating_pattern_tolerance'*.

  'module_grid': Algorithm for calculating the module positions (fixed or variable grid). The model parameter trained is *'module_grid'*.

  'image_proc': Adjusting different internal image processing parameters. Until now, only the maximum slant of the L-shaped finder pattern of the Data Matrix ECC 200 symbols is set (more parameters may follow in future). The model parameter trained is *'slant_max'*.

- QR Code only:

  'model_type': Whether the QR Code symbols follow the Model 1 or Model 2 specification. The model parameter trained is *'model_type'*.

  'position_pattern_min': Number of position detection patterns that have to be visible for generating a new symbol candidate. The model parameter trained is *'position_pattern_min'*.

- Aztec Code only

  'additional_levels': The number of additional pyramid levels. The model parameter trained is *'additional_levels'*.

It is possible to train several of these parameters in one call of `find_data_code_2d` by passing the generic parameter *'train'* in a tuple more than once in conjunction with the corresponding parameters: e.g., `GenParamName` = *['train','train']* and `GenParamValue` = *['polarity','module_size']*. Furthermore, in conjunction with *'train'* = *'all'* it is possible to exclude single parameters from training explicitly again by passing *'train'* more than once. The names of the parameters to exclude, however, must be prefixed by *'~'*: `GenParamName` = *['train','train']* and `GenParamValue` = *['all','~contrast']*, e.g., trains all parameters except the minimum contrast.

For training the model, the following aspects should be considered:

- To use several images for the training, the operator `find_data_code_2d` must be called with the parameter *'train'* once for every sample image.
- It is also possible to train the model with several symbols in one image. Here, the generic parameter *'stop_after_result_num'* must be passed as a tuple to `GenParamName` together with *'train'*. The number of symbols in the image is passed in `GenParamValue` together with the training parameters.

- If the training image contains more symbols than the one that shall be used for the training the domain of the image should be reduced to the symbol of interest with reduce_domain.
- In an application with very similar images, *one* image for training may be sufficient if the following assumptions are true: The symbol size (in modules) is the same for all symbols used in the application, foreground and background modules are of the same size and there is no gap between neighboring foreground modules, the background has no distinct texture; and the contrast of all images is almost the same. Otherwise, several images should be used for training.
- In applications where the symbol size (in modules) is not fixed, the smallest as well as the biggest symbols should be used for the training. If this can not be guaranteed, the limits for the symbol size should be adapted manually after the training, or the symbol size should entirely be excluded from the training.
- During the first call of find_data_code_2d in the training mode, the trained model parameters are restricted to the properties of the detected symbol. Any successive training call will, where necessary, extend the parameter range to cover the already trained symbols as well as the new symbols. Resetting the model with set_data_code_2d_param to one of its default settings (*'default_parameters'* = *'standard_recognition'*, *'enhanced_recognition'*, or *'maximum_recognition'*) will also reset the training state of the model. With set_data_code_2d_param and *'trained'* the training state of parameters can be set to trained. Subsequent training of this parameter will not reset its value, but extend it, so symbols readable by the previous value can still be read after training.
- If find_data_code_2d is not able to read the symbol in the training image, this will produce no error or exception handling. This can simply be tested in the program by checking the results of find_data_code_2d: SymbolXLDs, ResultHandles, DecodedDataStrings. These tuples will be empty, and the model will not be modified.

Note that during training, a possibly set timeout is ignored (see set_data_code_2d_param).

Functionality of the symbol search

- All data code types:

  Depending on the current settings of the 2D data code model (see set_data_code_2d_param), the operator find_data_code_2d performs several passes for searching the data code symbols. The search starts at the highest pyramid level, where – according to the maximum module size defined in the data code model – the modules can be separated. In addition, in every pyramid level the preprocessing can vary depending on the presets for the module gap. If the data code model enables dark symbols on a light background as well as light symbols on a dark background, within the current pyramid level, the dark symbols are searched first. Then the passes for searching light symbols follow. A pass consists of two phases: The search phase is used to look for the finder patterns and to generate a symbol candidate for every detected finder pattern, and the evaluation phase, where in a lower pyramid level all candidates are investigated and – if possible – read.

  The operator call is either terminated after successfully decoding the requested number of symbols, after processing all search passes, or due to a timeout (see set_data_code_2d_param). The number of requested symbols can be specified via the generic parameter GenParamName = *'stop_after_result_num'*. Without specifying this number the search stops as soon as one symbol could be decoded.
- Data Matrix ECC 200, QR Code, and Micro QR Code:

  The complexity of the symbol search can be controlled with the generic parameter GenParamName = *'symbol_search'*. Per default, *'symbol_search'* is set to GenParamValue = *'default'*.

  – For simple images, i.e. images that:
    * only contain symbols with high contrast and a large quiet zone and
    * show a homogeneous background

    a less complex and (in simple cases) faster symbol search method can be used by setting *'symbol_search'* to *'rudimental'*. Please note that in this case the following parameters do not have any effect: *'module_gap_min'*, *'module_gap_max'*, *'module_gap'*, *'finder_pattern_tolerance'*,*'contrast_tolerance'*, and *'candidate_selection'*. Further, no parameters can be trained with this method.

  – For large images with many codes and a highly structured background, the number of code candidates can be very high for certain parameter settings, e.g. for *'maximum_recognition'*. For performance reasons, the ECC 200 reader, the QR Code reader, and the Micro QR Code reader stop by default after 10000 processed candidates.

    For cases, where more candidates need to be considered in order to decode all codes in the image the *'symbol_search'* can be set to *'exhaustive'*. This will stop the search only when

'stop_after_result_num' codes have been decoded or all candidates have been processed. Please note that this may increase the runtime accordingly. The actual number of candidates can be queried with `get_data_code_2d_results`.

Query results of the symbol search

With the result handles and the operators `get_data_code_2d_results` and `get_data_code_2d_objects`, additional data can be requested about the search process, e.g., the number of internal search passes or the number of investigated candidates, and – together with the `ResultHandles` – about the symbols, like the symbol and module size, the contrast, or the raw data coded in the symbol. In addition, these operators provide information about all investigated candidates that could not be read. In particular, this helps to determine if a candidate was actually generated at the symbol's position during the preprocessing and – by the value of a status variable – why the search or reading was aborted. Further information about the parameters can be found with the operators `get_data_code_2d_results` and `get_data_code_2d_objects`.

Timeout and Abort

The operator `find_data_code_2d` can be aborted by a timeout and dynamically. With the operator `set_data_code_2d_param` you can specify a timeout. If `find_data_code_2d` reaches this timeout, it returns all codes decoded so far. Alternatively, you can call `set_data_code_2d_param` with *'abort'* from another thread to abort `find_data_code_2d` dynamically.

The information whether the operator was aborted or not can be queried by calling `get_data_code_2d_results` with the parameter *'aborted'*.

Furthermore, the operator `find_data_code_2d` can be canceled, which means no result is returned but instead an error is returned. This can be done with the operator `set_operator_timeout` or `interrupt_operator`. If `find_data_code_2d` is canceled by `set_operator_timeout`, H_ERR_TIMEOUT (9400) is returned. If `find_data_code_2d` is canceled by `interrupt_operator`, H_ERR_CANCEL (22) is returned. Note: Both mentioned operators are only supported in cancel mode.

Special parallelization for Data Matrix ECC 200

The operator `find_data_code_2d` supports special parallelization features for Data Matrix ECC 200. This additional internal parallelization can lead to a significantly shorter execution time under certain circumstances. In general, faster calculation times can be expected if, e.g., `set_data_code_2d_param` is used to set the value *'any'* for one or more of the parameters *'polarity'*, *'contrast_tolerance'* and *'finder_pattern_tolerance'*, or if the set values for *'module_gap_min'* and *'module_gap_max'* differ. In all these cases, a series of parameter values must be taken into account. This applies in particular if *'default_parameters'* has been set to *'enhanced_recognition'* or *'maximum_recognition'*.

Please note, that the memory consumption increases with the number of parallel running threads. To reduce the memory consumption `set_system` can be used to either set the number of threads with the parameter *'thread_num'*, or to switch off the temporary memory cache with the parameter *'temporary_mem_cache'*. The Data Matrix ECC 200 specific parallelization can also be easily disabled by setting the generic parameter `GenParamName` = *'specific_parallelization'* to *'disable'*. Note that this will only switch off the additional internal parallelization for Data Matrix ECC 200. Certain sub-tasks might be still processed in parallel. For more information on how to disable the entire parallelization, see the operator reference of `set_system` with its parameters *'thread_num'* and *'parallelize_operators'*. In general, however, it should not be necessary to switch off the specific parallelization. Per default, *'specific_parallelization'* is set to *'enable'*.

Chinese characters

If a QR Code contains Chinese characters encoded according to the Chinese national standard GBT 18284-2000, `find_data_code_2d` returns these characters UTF-8 encoded in `DecodedDataStrings`, if the system parameter *'filename_encoding'* is set to *'utf8'*. The contents of *'decoded_data'*, which can be retrieved with `get_data_code_2d_results`, are never converted to UTF-8.

Parameters

▷ **`Image`** (input_object) ..... singlechannelimage ⇝ *object :* byte
Input image. If the image has a reduced domain, the data code search is reduced to that domain. This usually reduces the runtime of the operator. However, if the datacode is not fully inside the domain, the datacode might not be found correctly. In rare cases, data codes may be found outside the domain. If these results are undesirable, they have to be subsequently eliminated.

▷ **SymbolXLDs** (output_object) ..... xld_cont(-array) ⇝ *object*
XLD contours that surround the successfully decoded data code symbols. The order of the contour points reflects the orientation of the detected symbols. The contours begin in the top left corner (see *'orientation'* in the reference of `get_data_code_2d_results`) and continue clockwise.

Order of points of `SymbolXLDs`

▷ **DataCodeHandle** (input_control) ..... datacode_2d ⇝ *handle*
Handle of the 2D data code model.

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Names of (optional) parameters for controlling the behavior of the operator.
Default: []
List of values: `GenParamName` ∈ {'train', 'stop_after_result_num', 'symbol_search', 'specific_parallelization'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *integer* / real / string
Values of the optional generic parameters.
Default: []
Suggested values: `GenParamValue` ∈ {'all', 'model_type', 'symbol_size', 'version', 'module_size', 'small_modules_robustness', 'module_shape', 'polarity', 'mirrored', 'contrast', 'candidate_selection', 'module_grid', 'alternating_pattern_tolerance', 'finder_pattern_tolerance', 'contrast_tolerance', 'position_pattern_min', 'additional_levels', 'image_proc', 'rudimental', 'default', 'exhaustive', 1, 2, 3, 'enable', 'disable'}

▷ **ResultHandles** (output_control) ..... integer(-array) ⇝ *integer*
Handles of all successfully decoded 2D data code symbols.

▷ **DecodedDataStrings** (output_control) ..... string(-array) ⇝ *string*
Decoded data strings of all detected 2D data code symbols in the image.

Example

```
* Examples showing the use of find_data_code_2d.
* First, the operator is used to train the model, afterwards it is used to
* read the symbol in another image.

* Create a model for reading Data matrix ECC 200 codes
create_data_code_2d_model ('Data Matrix ECC 200', [], [], DataCodeHandle)
* Read a training image
read_image (Image, 'datacode/ecc200/ecc200_cpu_007')
* Train the model with the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, 'train', 'all', \
                   ResultHandles, DecodedDataStrings)
*
* End of training / begin of normal application
*

* Read an image
read_image (Image, 'datacode/ecc200/ecc200_cpu_010')
* Read the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], \
```

```
                    ResultHandles, DecodedDataStrings)

* Display all symbols, the strings encoded in them, and the module size
dev_set_color ('green')
for i := 0 to |ResultHandles| - 1 by 1
    select_obj (SymbolXLDs, SymbolXLD, i+1)
    dev_display (SymbolXLD)
    get_contour_xld (SymbolXLD, Row, Col)
    set_tposition (WindowHandle, max(Row), min(Col))
    write_string (WindowHandle, DecodedDataStrings[i])
    get_data_code_2d_results (DataCodeHandle, ResultHandles[i], \
                              ['module_height','module_width'], ModuleSize)
    new_line (WindowHandle)
    write_string (WindowHandle, 'module size = ' + ModuleSize[0] + 'x' + \
                                 ModuleSize[1])
endfor

* Clear the model
clear_data_code_2d_model (DataCodeHandle)
```

Result

The operator `find_data_code_2d` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- DataCodeHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_data_code_2d_model`, `read_data_code_2d_model`, `set_data_code_2d_param`

Possible Successors

`get_data_code_2d_results`, `get_data_code_2d_objects`, `write_data_code_2d_model`

See also

`create_data_code_2d_model`, `set_data_code_2d_param`, `get_data_code_2d_results`, `get_data_code_2d_objects`

References

GS1 General Specifications; Version 12; Issue 1, Jan-2012; GS1.

Module

Data Code

get_data_code_2d_objects (: DataCodeObjects : DataCodeHandle, CandidateHandle, ObjectName :)

Access iconic objects that were created during the search for 2D data code symbols.

The operator `get_data_code_2d_objects` facilitates to access iconic objects that were created during the last call of `find_data_code_2d` while searching and reading the 2D data code symbols. Besides the name of the object (`ObjectName`), the 2D data code model (`DataCodeHandle`) must be passed

to get_data_code_2d_objects. In addition, in CandidateHandle a handle of a result or candidate structure or a string identifying a group of candidates (see get_data_code_2d_results) must be passed. These handles are returned by find_data_code_2d for all successfully decoded symbols and by get_data_code_2d_results for a group of candidates. If these operators return several handles in a tuple, the individual handles can be accessed by normal tuple operations.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

If *'discard_undecoded_candidates'* was set to *'yes'* with set_data_code_2d_param, only results of successfully decoded candidates can be accessed.

Some objects are not accessible without setting the model parameter *'persistence'* to *1* (see set_data_code_2d_param). The persistence must be set before calling find_data_code_2d, either while creating the model with create_data_code_2d_model or with set_data_code_2d_param.

Currently, the following iconic objects can be retrieved:

Regions of the modules

- All data code types:

  'module_1_rois': All modules that were classified as foreground (set).

- All data code types except DotCode:

  'module_0_rois': All modules that were classified as background (not set).

These region arrays correspond to the areas that were used for the classification. The returned object is a region array. Hence it cannot be requested for a group of candidates. Therefore, a single result handle must be passed in CandidateHandle. The model persistence must be *1* for this object. In addition, requesting the module ROIs makes sense only for symbols that were detected as valid symbols. For other candidates, whose processing was aborted earlier, the module ROIs are not available.

XLD contour

'candidate_xld': An XLD contour that surrounds the candidate or decoded symbol.

This object can be requested for any group of results or for any single candidate or symbol handle. The persistence setting is of no relevance.

Pyramid images

'search_image': Pyramid image, in which the candidate was found.

'process_image': Pyramid image, in which the candidate was investigated more closely.

The persistence setting is also not relevant here.

Parameters

▷ **DataCodeObjects** (output_object) ..... object(-array) ⇝ *object*
Objects that are created as intermediate results during the detection or evaluation of 2D data codes.

▷ **DataCodeHandle** (input_control) ..... datacode_2d ⇝ *handle*
Handle of the 2D data code model.

▷ **CandidateHandle** (input_control) ..... integer ⇝ *integer* / string
Handle of the 2D data code candidate. Either an integer (usually the ResultHandle of find_data_code_2d) or a string representing a group of candidates.
Default: 'all_candidates'
Suggested values: CandidateHandle ∈ {0, 1, 2, 'all_candidates', 'all_results', 'all_undecoded', 'all_aborted'}

▷ **ObjectName** (input_control) ..... string ⇝ *string*
Name of the iconic object to return.
Default: 'candidate_xld'
List of values: ObjectName ∈ {'module_1_rois', 'module_0_rois', 'candidate_xld', 'search_image', 'process_image'}

Example

```
* Example demonstrating how to access the iconic objects of the data code
* search.

* Create a model for reading Data matrix ECC 200 codes
create_data_code_2d_model ('Data Matrix ECC 200', [], [], DataCodeHandle)
set_data_code_2d_param (DataCodeHandle, 'default_parameters', \
                        'maximum_recognition')
set_data_code_2d_param (DataCodeHandle, 'persistence', 1)
* Read an image
read_image (Image, 'datacode/ecc200/ecc200_disturbed_012')
* Read the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], \
                   ResultHandles, DecodedDataStrings)

* Get the handles of all candidates that were detected as a symbol but
* could not be read
get_data_code_2d_results (DataCodeHandle, 'all_undecoded', 'handle', \
                          HandlesUndecoded)

* For every undecoded symbol, get the contour and the classified
* module regions
for I := 0 to |HandlesUndecoded| - 1 by 1
  * Display image.
  dev_display (Image)
  dev_set_draw ('margin')
  * Get the contour of the symbol.
  dev_set_color ('blue')
  get_data_code_2d_objects (SymbolXLD, DataCodeHandle, \
                            HandlesUndecoded[I], 'candidate_xld')
  dev_display (SymbolXLD)
  * Get the module regions of the foreground modules
  dev_set_color ('green')
  get_data_code_2d_objects (ModuleFG, DataCodeHandle, \
                            HandlesUndecoded[I], 'module_1_rois')
  dev_display (ModuleFG)
  * Get the module regions of the background modules
  dev_set_color ('red')
  get_data_code_2d_objects (ModuleBG, DataCodeHandle, \
                            HandlesUndecoded[I], 'module_0_rois')
  dev_display (ModuleBG)
  * Stop for inspecting the image.
  stop ()
endfor

* Clear the model
clear_data_code_2d_model (DataCodeHandle)
```

Result

The operator `get_data_code_2d_objects` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct and the requested objects are available for the last symbol search. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

find_data_code_2d, query_data_code_2d_params

Possible Successors

get_data_code_2d_results

See also

query_data_code_2d_params, get_data_code_2d_results, get_data_code_2d_param, set_data_code_2d_param

Module

Data Code

get_data_code_2d_param (: : DataCodeHandle, GenParamName : GenParamValue)

Get one or several parameters that describe the 2D data code model.

The operator get_data_code_2d_param allows to query the parameters that are used to describe the 2D data code model. The names of the desired parameters are passed in the generic parameter GenParamName, the corresponding values are returned in GenParamValue. All these parameters can be set and changed at any time with the operator set_data_code_2d_param. A list with the names of all parameters that are valid for the used 2D data code type is returned by the operator query_data_code_2d_params.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

Note that the symbol structure of GS1 DataMatrix, GS1 QR Code, GS1 Aztec Code, and GS1 DotCode is identical to the structure of Data Matrix ECC 200, QR Code, Aztec Code, and DotCode, respectively. Therefore, all symbology specific parameters applying to Data Matrix ECC 200, QR Code, Aztec Code, or DotCode apply to their corresponding GS1 variant as well. In the following, the explicit enumeration of the parameters for any particular GS1 code is omitted for sake of readability. Instead, the relevant parameters names are to be inferred from the parameters for the corresponding non-GS1 code type or can be explicitly queried by query_data_code_2d_params with parameter *'get_model_params'*.

The following parameters can be queried – ordered by different categories and data code types:

Size and shape of the symbol:

- Data Matrix ECC 200 (including the finder pattern):

  'symbol_cols_min': Minimum number of module columns in the symbol.

  'symbol_cols_max': Maximum number of module columns in the symbol.

  'symbol_rows_min': Minimum number of module rows in the symbol.

  'symbol_rows_max': Maximum number of module rows in the symbol.

  'symbol_shape': Possible restrictions concerning the module shape (rectangle and/or square)
  List of values: *'square'*, *'rectangle'*, *'any'*.

- QR Code (including the finder pattern):

  'model_type': Type of the QR Code model specification: *1*, *2*, *0* (for *'any'*)

  'version_min': Minimum symbol version to be read: [*1*...*40*] (Model 1: [*1*...*14*])

  'version_max': Maximum symbol version to be read: [*1*...*40*] (Model 1: [*1*...*14*])

  'symbol_size_min': Minimum symbol size (this value is directly linked to the version *'version_min'*): [*21*...*177*] (Model 1: [*21*...*73*])

  'symbol_size_max': Maximum symbol size (this value is directly linked to the version *'version_max'*): [*21*...*177*] (Model 1: [*21*...*73*])

- Micro QR Code:

  'version_min': Minimum symbol version to be read: [*1*...*4*]

  'version_max': Maximum symbol version to be read: [*1*...*4*]

  'symbol_size_min': Minimum symbol size (this value is directly linked to the version *'version_min'*): [*11*...*17*]

'symbol_size_max': Maximum symbol size (this value is directly linked to the version *'version_max'*): [*11*...*17*]

- PDF417:

  'symbol_cols_min': Minimum number of data columns in the symbol in codewords, i.e., excluding the codewords of the start/stop pattern and of the two row indicators.

  'symbol_cols_max': Maximum number of data columns in the symbol in codewords, i.e., excluding the codewords of the start/stop pattern and of the two row indicators.

  'symbol_rows_min': Minimum number of module rows in the symbol.

  'symbol_rows_max': Maximum number of module rows in the symbol.

- Aztec Code (including the finder pattern):

  'format': Format of the Aztec Code: space separated list with the values *'compact'*, *'full_range'*, or *'rune'*

  'symbol_size_min': Minimum symbol size [*11*...*151*]

  'symbol_size_max': Maximum symbol size [*11*...*151*]

- DotCode:

  'symbol_cols_min': Minimum number of module columns in the symbol.

  'symbol_cols_max': Maximum number of module columns in the symbol.

  'symbol_rows_min': Minimum number of module rows in the symbol.

  'symbol_rows_max': Maximum number of module rows in the symbol.

Appearance of the modules in the image:

- All data code types:

  'polarity': Possible restrictions concerning the polarity of the modules, i.e., if they are printed dark on a light background or vice versa: *'dark_on_light'*, *'light_on_dark'*, *'any'*.

  'discard_undecoded_candidates' Controls whether candidates that could not be successfully decoded are stored in the model: *'yes'*, *'no'*.

  'mirrored': Describes whether the symbol is or may be mirrored (which is equivalent to swapping the rows and columns of the symbol): *'yes'*, *'no'*, *'any'*.

- All data code types except Data Matrix ECC 200 and DotCode:

  'contrast_min': Minimum contrast between the foreground and the background of the symbol (specified as gray value difference). This measure corresponds to the minimum gradient between the symbol's foreground and the background.

- Data Matrix ECC 200, Micro QR Code, and QR Code:

  'contrast_tolerance': Describes the tolerance of the search with respect to local contrast variations (e.g., in the presence of glare or reflections). Depending on the value of the parameter two different algorithms are applied. If *'contrast_tolerance'* is set to *'high'* the robustness in the presence of strong local contrast variations is improved. In the case where *'contrast_tolerance'* is set to *'low'* the algorithm used is less robust to strong local contrast variations, however, it is faster and still able to handle contrast variations under normal circumstances and therefore *'low'* should be used in most cases. If *'contrast_tolerance'* is set to *'any'* both algorithms are applied.

- All data code types except PDF417:

  'module_size_min': Minimum module size in the image in pixels.

  'module_size_max': Maximum module size in the image in pixels.

  With the following parameters it is possible to specify whether neighboring foreground modules are connected or whether there is or may be a gap between them (possible values are *'no'* (no gap) $<$ *'small'* $<$ *'big'*) (for DotCode, only *'no'* and *'small'* are available):

  'module_gap_min': Minimum gap.

  'module_gap_max': Maximum gap.

- All data code types except DotCode:

'small_modules_robustness': Robustness of the decoding of data codes with very small module sizes. Setting the parameter *'small_modules_robustness'* to *'high'* increases the likelihood of being able to decode data codes with very small module sizes. Additionally, in that case the minimum module size should also be adapted accordingly, thus *'module_size_min'* and *'module_width_min'* (PDF417) should be set to the expected minimum module size and width, respectively. Setting *'small_modules_robustness'* to *'high'* can significantly increase the internal memory usage of `find_data_code_2d`. Thus, in the default case *'small_modules_robustness'* should be set to *'low'*.
List of values: *'low'*, *'high'*
Default: *'low'* (enhanced: *'low'*, maximum: *'high'*)

- PDF417:

  'module_width_min': Minimum module width in the image in pixels.

  'module_width_max': Maximum module width in the image in pixels.

  'module_aspect_min': Minimum module aspect ratio (module height to module width).

  'module_aspect_max': Maximum module aspect ratio (module height to module width).

- Aztec Code:

  'finder_pattern_tolerance': Tolerance of the search with respect to a defect or partially occluded finder pattern. Depending on this parameter, different algorithms are used during the symbol search in `find_data_code_2d`. In one case (*'low'*), it is assumed that all rings of the finder pattern can be extracted. In the other case (*'high'*) it is assumed that at least one of the rings of the finder pattern can be extracted.

  'additional_levels': To increase the robustness of the Aztec Code reader, a number of additional search levels (in addition to the search levels derived from the minimum and maximum module dimensions) can be specified via this parameter. [*0*...*2*]

- Data Matrix ECC 200:

  'slant_max': Maximum slant of the L-shaped finder pattern (the angle is returned in radians and corresponds to the distortion that occurs when the symbol is printed or during the image acquisition).

  'finder_pattern_tolerance': Tolerance of the search with respect to a defect or partially occluded finder pattern. The finder pattern includes the L-shaped side as well as the opposite alternating side. Depending on this parameter, different algorithms are used during the symbol search in `find_data_code_2d`. In one case (*'low'*), it is assumed that the finder pattern is present to a high degree and shows almost no disturbances. In the other case (*'high'*), the finder pattern may be defect or partially occluded without influencing the recognition and the reading of the symbol. Note, however, that in this mode the parameters for the symbol search should be restricted as narrow as possible by using `set_data_code_2d_param` because otherwise the runtime of `find_data_code_2d` may increase significantly. Also note that the two algorithms slightly differ from each other in terms of robustness. This may lead to different results depending on the value of *'finder_pattern_tolerance'* even if the finder pattern of the symbol is not disturbed. For example, if *'high'* is chosen, only symbols with an equidistant module grid can be found (see below), and hence the robustness to perspective distortions is decreased. Finally, if *'finder_pattern_tolerance'* is set to *'any'* both algorithms are applied.

  'alternating_pattern_tolerance': Tolerance of the search with respect to the variation of the module widths along the two sides with the alternating pattern. The alternating pattern analysis is decisive if *'finder_pattern_tolerance'* is set to *'low'* (or *'any'* which includes *'low'*). Note, however, that this parameter has no effect if *'finder_pattern_tolerance'* has been set to *'high'*. A total of three different values are allowed for this parameter, with a higher setting always including the lower values: *'low'* allows only a small variation of the module widths along the sides with the alternating pattern. *'medium'* first tries to meet the strict requirements of *'low'*, and if this fails, a larger variation of the module widths along the sides with the alternating pattern is allowed. Finally, *'high'* includes everything that *'medium'* allowed and additionally expands the parameter space for the symbol's outer boundaries through further attempts. It also softens the relation between the number of edges detected and the symbol type used for subsequent decoding. The greater robustness in difficult cases comes with the disadvantage that *'high'* in particular can lead to a significant runtime increase. To counteract this, the parameter space should be reduced in total by using `set_data_code_2d_param`, e.g., by setting the expected symbol sizes as tight as possible.

  'module_grid': Describes whether the size of the modules may vary (in a specific range) or not. Dependent on the parameter different algorithms are used for the calculation of the module's center positions. If it is set to *'fixed'*, an equidistant grid is used. Allowing a variable module size (*'variable'*), the grid is

aligned only to the alternating side of the finder pattern. With *'any'* both approaches are tested one after the other. Please note that the value of *'module_grid'* is ignored if *'finder_pattern_tolerance'* is set to *'high'*. In this case, an equidistant grid is assumed.

- QR Code:

  'position_pattern_min': Number of position detection patterns that have to be visible for generating a new symbol candidate (2 or 3).

General model behavior:

- All data code types:

  'persistence': Controls whether certain intermediate results of the symbol search with `find_data_code_2d` are stored only temporarily or persistently in the model: *0* (temporary), *1* (persistent).

  'strict_model': Controls the behavior of `find_data_code_2d` while detecting symbols that could be read but that do not fit the model restrictions concerning the size of the symbols: *'yes'* (strict: such symbols are rejected), *'no'* (not strict: all readable symbols are returned as a result independent of their size and the size specified in the model). Please note that for DotCode symbols the module size restrictions (*'module_size_min'* and *'module_size_max'*) are not checked even if *'strict_model'* is set to *'yes'*.

  'string_encoding': Returns the expected encoding of the string that is encoded in the symbol: *'utf8'*, *'locale'*, *'latin1'*, *'shiftjis'*, or *'raw'*.

  'symbol_type': Returns symbol type which was set with `create_data_code_2d_model`.

  'timeout': Enables aborting `find_data_code_2d` after a defined period in milliseconds: *'false'*, *-1*, *20 ... 100*.

- All data code types except Aztec Code:

  'strict_quiet_zone': Controls the behavior of `find_data_code_2d` while detecting symbols that could be read but show defects in their quiet zone. Possible values:

  'yes': Decoded symbols with poor grades for their quiet zone are not returned as a result. The quiet zone is validated similar to the method used for print quality inspection, but regarded as being only one module large. The *'status'* for rejected symbols is set to *'quiet zone is missing'*.

  'no' **(default):** All readable symbols are returned as a result.

- All data code types except DotCode:

  'quality_isoiec15415_aperture_size': Defines the aperture sizes for ISO/IEC 15415:2011 print quality inspection as fraction of the module width (see ISO/IEC 15415:2011 section 7.3.3).

  'quality_isoiec15415_reflectance_reference': Defines the reference reflectance for ISO/IEC 15415:2011 print quality inspection as grayvalue (see ISO/IEC 15415:2011 section 7.3).

  'quality_isoiec15415_smallest_module_size': Defines the module size used to obtain the synthetic aperture for ISO/IEC 15415:2011 print quality inspection (see ISO/IEC 15415:2011 section 7.3.3).

- Data Matrix ECC 200, DotCode, Micro QR Code, and QR Code:

  'candidate_selection': Controls the selection of candidate regions that are used for symbol detection. Setting this parameter to *'extensive'* increases the number of generated candidate regions and thus the likelihood of detecting a code. When this parameter is set to *'all'*, all possible candidates are used. If *'candidate_selection'* is set to *'default'*, less candidate regions are used.

- DotCode:

  'max_allowed_error_correction': Controls the maximum allowed error correction. Due to the high error correction capacity, it is possible to successfully decode false positive DotCode candidates. Especially candidates which only cover a small part of a real DotCode may be decoded successfully because of the error correction capabilities. This is the case because DotCode symbols can consist of almost every size. So there are fewer criteria to decide whether the candidate is valid or not. In order to tackle this problem, the parameter *'max_allowed_error_correction'* can be used to specify the percentage of maximum allowed error correction. Per default the value is set to *'0.9'*, which means 90%. Setting the value to e.g., *'0.5'* means, only candidates which could be decoded with maximum 50% used error correction, will be returned as successfully decoded results.
  Value range: [*0.0 ... 1.0*]
  Default: *'0.9'*

It is possible to query the values of several or all parameters with a single operator call by passing a tuple containing the names of all desired parameters to GenParamName. As a result a tuple of the same length with the corresponding values is returned in GenParamValue.

Parameters

▷ **DataCodeHandle** (input_control) . . . . . datacode_2d ⇝ *handle*
Handle of the 2D data code model.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that are to be queried for the 2D data code model.
Default: 'polarity'
List of values: GenParamName ∈ {'strict_model', 'persistence', 'polarity', 'mirrored', 'contrast_min', 'candidate_selection', 'discard_undecoded_candidates', 'model_type', 'version_min', 'version_max', 'format', 'string_encoding', 'symbol_size_min', 'symbol_size_max', 'symbol_cols_min', 'symbol_cols_max', 'symbol_rows_min', 'symbol_rows_max', 'symbol_shape', 'module_size_min', 'module_size_max', 'module_width_min', 'module_width_max', 'small_modules_robustness', 'module_aspect_min', 'module_aspect_max', 'module_gap_min', 'module_gap_max', 'slant_max', 'module_grid', 'position_pattern_min', 'strict_quiet_zone', 'timeout', 'alternating_pattern_tolerance', 'finder_pattern_tolerance', 'additional_levels', 'contrast_tolerance', 'quality_isoiec15415_aperture_size', 'quality_isoiec15415_decode_algorithm', 'quality_isoiec15415_reflectance_reference', 'quality_isoiec15415_smallest_module_size', 'decoding_scheme', 'symbol_type', 'max_allowed_error_correction'}

▷ **GenParamValue** (output_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Values of the generic parameters.

Result

The operator get_data_code_2d_param returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_data_code_2d_params, set_data_code_2d_param, find_data_code_2d

Possible Successors

find_data_code_2d, write_data_code_2d_model

Alternatives

write_data_code_2d_model

See also

query_data_code_2d_params, set_data_code_2d_param, get_data_code_2d_results, get_data_code_2d_objects, find_data_code_2d

Module

Data Code

get_data_code_2d_results (: : DataCodeHandle, CandidateHandle, ResultNames : ResultValues)

Get the alphanumerical results that were accumulated during the search for 2D data code symbols.

The operator get_data_code_2d_results allows to access several alphanumerical results that were calculated while searching and reading the 2D data code symbols. These results describe the search process in general or one of the investigated candidates – independently of whether it could be read or not. The results are in most cases not related to the symbol with the highest resolution but depend on the pyramid level that was investigated when the reading process was aborted. To access a result, the name of the parameter (ResultNames) and the 2D data code model (DataCodeHandle) must be passed. In addition, in CandidateHandle a handle of a result

or candidate structure or a string identifying a group of candidates must be passed. These handles are returned by find_data_code_2d for all successfully decoded symbols and by get_data_code_2d_results for a group of candidates. If these operators return several handles in a tuple, the individual handles can be accessed by normal tuple operations.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

If *'discard_undecoded_candidates'* was set to *'yes'* with set_data_code_2d_param, only results of successfully decoded candidates can be accessed.

Most results consist of one value. Several of these results can be queried for a specific candidate in a single call. The values returned in ResultValues correspond to the appropriate parameter names in the ResultNames tuple. As an alternative, these results can also be queried for a group of candidates (see below). In this case, only one parameter can be requested per call, and ResultValues contains one value for every candidate.

Furthermore, there exists another group of results that consist of more than one value (e.g., *'bin_module_data'*), which are returned as a tuple. These parameters must always be queried exclusively: one result for one specific candidate.

Apart from the candidate-specific results there are a number of results referring to the search process in general. This is indicated by passing the string *'general'* in CandidateHandle instead of a candidate handle.

Note that the symbol structure of GS1 DataMatrix, GS1 QR Code, GS1 Aztec Code, and GS1 DotCode is identical to the structure of Data Matrix ECC 200, QR Code, Aztec Code, or DotCode, respectively. Therefore, all type specific results for Data Matrix ECC 200, QR Code, Aztec Code, and DotCode can be queried for their corresponding GS1 variant as well. In the following, the explicit enumeration of the type specific results for any particular GS1 code is omitted for sake of readability. Instead, the relevant result names are to be inferred from the result names for the corresponding non-GS1 code type and or be explicitly queried by query_data_code_2d_params with parameter *'get_result_params'*.

Candidate groups

The following candidate group names are predefined and can be passed as CandidateHandle instead of a single handle:

'general': This value is used for results that refer to the last find_data_code_2d call in general but not to a specific candidate.

'all_candidates': All candidates (including the successfully decoded symbols) that were investigated during the last call of find_data_code_2d.

'all_results': All symbols that were successfully decoded during the last call of find_data_code_2d.

'all_undecoded': All candidates of the last call of find_data_code_2d that were detected as 2D data code symbols, but could not be decoded. For these candidates the error correction detected too many errors, or there was an failure while decoding the error-corrected data because of inconsistent data.

'all_aborted': All candidates of the last call of find_data_code_2d that could not be identified as valid 2D data code symbols and for which the processing was aborted.

Supported results

Currently, the access to the following results, which are returned in ResultValues, is supported:

General results that do not depend on specific candidates – *'general'*:

- All data code types:

  'min_search_level': Lowest pyramid level that is searched for symbols. A pyramid level of 0 corresponds to the original image.

  'max_search_level': Highest pyramid level that is searched for symbols. A pyramid level of 0 corresponds to the original image.

  'result_num': Number of successfully decoded symbols.

  'candidate_num': Number of all investigated candidates.

  'undecoded_num': Number of candidates that were identified as symbols but could not be read.

  'aborted_num': Number of candidates that could not be identified as valid 2D data code symbols.

  'aborted': returns whether find_data_code_2d was aborted. This might be caused by a timeout (see *'timeout'* in set_data_code_2d_param) or an explicit abort (see *'abort'* in set_data_code_2d_param).

Value	Description
0	find_data_code_2d completed
1	find_data_code_2d was aborted by a timeout
2	find_data_code_2d was aborted using set_data_code_2d_param with *'abort'*

- Data Matrix ECC 200, Aztec Code, QR Code, and Micro QR Code:

  'quality_isoiec29158_labels' **and** *'quality_aimdpm_1_2006_labels'*: Labels of grades for all tuple elements returned when calling get_data_code_2d_results with *'quality_isoiec29158'* or *'quality_aimdpm_1_2006'*.

- Data Matrix ECC 200, Aztec Code, PDF417, QR Code, and Micro QR Code:

  'pass_num': Number of passes that were completed, see also find_data_code_2d (Functionality of the symbol search).

  'quality_isoiec15415_labels': Labels of grades for all tuple elements returned when calling get_data_code_2d_results with *'quality_isoiec15415'*. The labels are different depending on the type of the evaluated data code symbol.

- Data Matrix ECC 200:

  'quality_semi_t10_labels': Convenience value labels of the elements of the tuple returned when calling get_data_code_2d_results with *'quality_semi_t10_values'*.

Results associated with a specific candidate - one value results:

Results that contain exactly **one value** and hence can be applied to a **candidate group** or a **specific candidate**. Please consider that some of the following results will only be meaningful if the candidate could be successfully decoded (see *'status'*).

- All data code types:

  'handle': Handle to the candidate. This parameter is used to receive the handles of all candidates of the specified group.

  'status': Indicates whether the decoding was successful or why the processing was aborted.

  'search_level': Pyramid level on which the finder pattern was found.

  'process_level': Pyramid level on which the candidate was processed and decoded.

  'polarity': Polarity of the symbol. This is the assumption about the polarity that was used for searching the candidate.

  'mirrored': Indicates whether a successfully decoded symbol is mirrored or not. For ECC 200, PDF417, QR and Micro QR Codes *'no'* or *'yes'* is returned. For Aztec Codes *'false'* or *'true'* is returned. For candidates that could not be read, the parameter returns the mirroring specification of the model.

  'orientation': Indicates the orientation of a successfully decoded symbol. The orientation is an angle relative to the orientation described in the figure below. The angle is positive in counter clockwise direction and is given in degrees. It can be in the range of [-180.0 .. 180.0] degrees.

(1) (2) (3)

(4) (5)

Orientation at zero degree of a Aztec symbol (1), ECC200 symbol (2), QR code symbol (3), PDF417 symbol (4) and DotCode symbol (5). The respective finder patterns or characteristic points are highlighted.

'symbol_rows', *'symbol_cols'*: Data Matrix ECC 200, QR Code, Micro QR Code, Aztec Code, and DotCode:
detected size of the symbol in modules: number of rows and columns including the finder pattern;
PDF417:
detected number of rows and data columns (each 17 modules wide) within the symbol (excluding the start/stop patterns and the row indicators).

'module_height', *'module_width'*: Height and width of the modules in pixels.

'decoded_string': Result string that is encoded in the symbol – this query is useful only for successfully decoded strings. It returns the same string as `find_data_code_2d`.

'decoding_error': Decoding error – for successfully decoded symbols this is the number of errors that were detected and corrected by the error correction. The number of errors corresponds here to the number of code words that lead to errors when trying to read them. For PDF417 this number includes erasures, i.e. errors with known locations. If the error correction failed, a negative error code is returned.

- Data Matrix ECC 200, Aztec Code, PDF417 QR Code, and Micro QR Code:

  'pass': Number of the pass in which the candidate was generated and processed, see also `find_data_code_2d` (Functionality of the symbol search).

  'small_modules_robustness': Robustness of the decoding of data codes with very small module sizes. For *'low'* the data code was decoded with a method that has only a small robustness regarding very small module sizes. For *'high'* the data code was decoded with a method that has a high robustness in this regard.

  'contrast': Estimation of the symbol's contrast. This value is based on the gradient of the edge between the finder pattern and the background.

- Data Matrix ECC 200, Micro QR Code, and QR Code:

  'contrast_tolerance': This parameter informs about the algorithm that was used to correct local contrast variations in the symbol search. For *'high'* the symbol was found using an algorithm to correct strong local contrast variations. For *'low'* the symbol was found with the faster algorithm, which however is less robust with respect to local contrast variations.

- Data Matrix ECC 200, Aztec Code, QR Code, Micro QR Code, and DotCode:

  'module_gap': Assumption about the module gap that was used for searching the candidate.

- Data Matrix ECC 200, Aztec Code, QR Code, PDF417, and DotCode:

  'symbology_ident': The *Symbology Identifier* is used to indicate that the data code contains the FNC1 and/or ECI characters.
  FNC1 (*Function 1 Character*) is used if the data formatting conforms to specific predefined industry standards.
  The ECI protocol (*Extended Channel Interpretation*) is used to change the default interpretation of the encoded data. A 6-digit code number after the ECI character switches the interpretation of the following characters from the default to a specific code page like an international character set. In the output stream the ECI switch is coded as *'\nnnnnn'*. Therefore all backslashes (*'\'*, ASCII code 92), that occur in the normal output stream have to be doubled.
  The *'symbology_ident'* parameter returns only the actual identifier value m ($m \in [0,6]$ (QR-Code), $m \in [0,12]$ (Data Matrix ECC 200 and Aztec Code), $m \in [0,2]$ (PDF417) bzw. $m \in [0,5]$ (DotCode)) according to the specification of Data Matrix, QR Codes, Aztec Code, PDF417, and DotCode but not the identifier prefixes ']d', ']Q', ']z', ']L', and ']J' for Data Matrix, QR Codes, Aztec Codes, PDF417, and DotCode, respectively. If required, this *Symbology Identifier* composed of the prefix and the value m has to be preceded the decoded string (normally only if $m > 1$) manually (for Data Matrix ECC 200 and Aztec Codes, the values 10, 11, and 12 have to be converted to A, B, and C, respectively).
  GS1 symbologies have the following identifiers: ']d2' for GS1 DataMatrix, ']Q3' for GS1 QR Code, ']z1' for GS1 Aztec Code, and ']J1' for GS1 DotCode. Therefore, the *'symbology_ident'* parameter returns the values 2, 3, and 1, respectively.
  Symbols that contain ECI codes (and hence doubled backslashes) can be recognized by the following identifier values: Data Matrix ECC 200: 4, 5, 6, 10, 11 and 12, QR Code: 2, 4, and 6, PDF417: 1, Aztec Code: 3, 4, 5, 9, 10, and 11, DotCode: 3, 4, and 5.

- Data Matrix ECC 200 and DotCode:

  'reader_programming': If the symbol contains a Reader Programming character that indicates that the symbol encodes a message used to program the reader system, this parameter returns *'yes'*, otherwise this parameter returns *'no'*. The decoded message does not contain the Reader Programming character.

- Data Matrix ECC 200:

  'slant': Slant of the L-shaped finder pattern in radians. This is the difference between the angle of the 'L' and the right angle.

  'finder_pattern_tolerance': This parameter informs about the algorithm that found the symbol. For *'low'* the symbol was found with the algorithm that has only a small tolerance with respect to a defect or partially occluded finder pattern. For *'high'* the symbol was found with the algorithm that has a high tolerance in this regard. Please note that when using both algorithms simultaneously it may happen that symbols with an undisturbed finder pattern are found from the algorithm of high tolerance.

  'alternating_pattern_tolerance': This parameter informs about the algorithm that was used to detect and analyze the two sides with the alternating pattern. Based on this analysis, the associated symbol type is derived and the grid for the module classification is created. For *'low'*, the analysis was performed with a small tolerance to variations in the module widths. *'medium'*, when successfully decoded, signals that an acceptance of greater variations with respect to the module widths along the sides with the alternating pattern was necessary. Finally, *'high'* means that not only a larger variation has to be accepted, but additionally the parameter space for the symbol's outer boundaries had to be expanded or the relation between the number of edges and the symbol type had to be softened. In addition, *'undefined'* signals that the candidate either did not reach the alternating pattern analysis or was part of the search pass that is based on *'finder_pattern_tolerance'* set to *'high'*. In the latter case, the *'alternating_pattern_tolerance'* parameter has no effect.

  'module_grid': For symbols that could be decoded, this parameter informs about the algorithm that was used for calculating the module grid: If a variable grid was used it returns *'variable'*, and otherwise *'fixed'*. For symbols that could not be decoded, it returns the method that was used during the last decoding trial or, if the candidate was rejected before the decoding, the corresponding model setting.

- QR Codes:

  'version': Version number that corresponds to the size of the symbol (version 1 = 21 $\times$ 21, version 2 = 25 $\times$ 25, ..., version 40 = 177 $\times$ 177).

  'symbol_size': Detected size of the symbol in modules.

  'model_type': Type of the QR Code Model. In HALCON the older, original specification for QR Codes Model 1 as well as the newer, enhanced form Model 2 are supported.

  'error_correction_level': If a candidate is recognized as a QR Code, the first step is to read the format information encoded in the symbol. This includes the level of error correction (*'error_correction_level'* $\in$ [*'L'* (Low), *'M'* (Medium), *'Q'* (Quartile), *'H'* (High)]).

  'mask_pattern_ref': Additionally, the format information encoded in the symbol include a code for the pattern that was used for masking the data modules ($0 \leq$ *'mask_pattern_ref'* ≤ 7).

- Micro QR Codes:

  'version': Version number that corresponds to the size of the symbol (version M1 = 11 $\times$ 11, version M2 = 13 $\times$ 13, version M3 = 15 $\times$ 15, version M4 = 17 $\times$ 17).

  'symbol_size': Detected size of the symbol in modules.

  'error_correction_level': If a candidate is recognized as a Micro QR Code, the first step is to read the format information encoded in the symbol. This includes the level of error correction (*'error_correction_level'* $\in$ [*'N'* (None), *'L'* (Low), *'M'* (Medium), *'Q'* (Quartile)]).

  'mask_pattern_ref': Additionally, the format information encoded in the symbol include a code for the pattern that was used for masking the data modules ($0 \leq$ *'mask_pattern_ref'* ≤ 7).

- PDF417:

  'module_aspect': Module aspect ratio; this corresponds to the ratio of *'module_height'* to *'module_width'*.

  'error_correction_level': If a candidate is recognized as a PDF417 the first step is to read the format information encoded in the symbol. This includes the error correction level, which was used during encoding (*'error_correction_level'* $\in$ [*0*, *8*]).

'macro_exist': Symbols that are part of a group of symbols are called *"Macro PDF417"* symbols. These symbols contain additional information within a control block. For macro symbols *'macro_exist'* returns the value *1* while for conventional symbols *0* is returned.

'macro_segment_index': Returns the index of the symbol in the group. For macro symbols this information is obligatory.

'macro_file_id': Returns the group identifier as a string. For macro symbols this information is obligatory.

'macro_segment_count': Returns the number of symbols that belong to the group. For macro symbols this information is optional.

'macro_time_stamp': Returns the time stamp on the source file expressed as the elapsed time in seconds since 1970:01:01:00:00:00 GMT as a string. For macro symbols this information is optional.

'macro_checksum': Returns the CRC checksum computed over the entire source file using the CCITT-16 polynomial. For macro symbols this information is optional.

'macro_last_symbol': Returns *1* if the symbol is the last one within the group of symbols. Otherwise *0* is returned. For macro symbols this information is optional.

'quality_isoiec15415_float_grades': returns a tuple with the same assessment of print quality like *'quality_isoiec15415'*. For a PDF417 some grades are assessed with the international standard ISO/IEC 15416:2016. In compliance with the standard the grades are returned with one decimal place.

- Aztec Codes:

  'format': String that corresponds to the format of the decoded symbol: *'compact'*, *'full_range'*, or *'rune'*.

  'symbol_size': Detected size of the symbol in modules.

  'layer_num': The number of layers as encoded in the mode message of the symbol.

  'codeword_num': The number of codewords as encoded in the mode message of the symbol.

Results associated with a specific candidate - tuple of values results:

Results that return a **tuple of values** and hence can be requested only separately and only for a **single candidate**. Please consider that some of the following results will not return a useful value if the investigation of the candidate was aborted.

- All data code types:

  'bin_module_data': Binary symbol data that corresponds to the classification results of the individual modules – a value of 0 means that the module was classified as background and 100 indicates that the module belongs to the foreground. Values between 0 and 100 can be interpreted as foreground or background. The model persistence must be 1 for this result (see `set_data_code_2d_param`). For DotCodes the model persistence does not matter. It shall be noted that the order of *'bin_module_data'* does not necessarily match the order of the modules, which can be retrieved using *'module_1_rois'* and *'module_0_rois'* via `get_data_code_2d_objects`. Instead, the order for successfully decoded symbols depends on the actual *'mirrored'* status of the result, or the *'mirrored'* setting specified via `set_data_code_2d_param` for failed candidates.

  'raw_coded_data': Data obtained by mapping the binary data to data words according to the particular coding scheme of the symbol type. Single bits may still be erroneous, and the words that are used for the error correction are still included.

  'corr_coded_data': Data obtained after applying the error correction: erroneous bits are corrected and all redundant words are removed, but the words are still encoded according to the coding scheme that is specific for the data code type.

  'decoded_data': Tuple with the decoded data words (= characters of the decoded data string) as Latin-1 or ASCII code (see also Tuple / String Operations). Additionally, decoded words can be encoded as JIS8 or Shift JIS characters (QR Code and Micro QR Code only) or GB2312 characters (QR Code only). Note that for QR Code and Micro QR Code the Kanji characters are not returned as single (Shift JIS) code points by `get_data_code_2d_results`, but split into individual bytes.

- All data code types except DotCode:

  'quality_isoiec15415': Tuple with the assessment of print quality in compliance with the international standard ISO/IEC 15415:2011. The first element always contains the overall print quality of the symbol; the length of the tuple and the denotation of the remaining elements depend on the specific data code type. According to the standard, the grades are whole numbers from 0 to 4, where 0 is the lowest and 4 the

highest grade. It is important to note what, even though the implementation is based on the parts of the standard which are applicable to software, the computation of the print quality grades depends on the preceding decoding algorithm. Thus, different data code readers (of different vendors) can potentially produce slightly different results in the print quality assessment. See `set_data_code_2d_param` for how to set the algorithm used to determine the module grid for *Data Matrix ECC 200* codes with *'quality_isoiec15415_decode_algorithm'*.

Using these values requires a thorough understanding of the underlying algorithms, and we recommend to read ISO/IEC 15415:2011 along with this documentation.

We recommend to use high quality images without artifacts like defocus, noise, overexposure, or inhomogeneous illumination. Such artifacts influence the result of the print quality grading. Further, to obtain robust grading results, an effective resolution of at least ten pixels per module in width and height is required (see ISO/IEC 15415:2011, Chapter 7.3.3). The quality grades are only computed when the symbol region including the quiet zone is fully in the image, elsewise -1 is returned for the quality grades. For the reference decoding algorithm, more cases lead to quality grades equal to -1. Please refer to *'quality_isoiec15415_decode_algorithm'* for more information.

For the 2D data codes *ECC 200*, *Aztec Code*, *QR Code*, and *Micro QR Code* the print quality is described in a tuple with twelve elements:(*overall quality, contrast, modulation, fixed pattern damage, decode, axial non-uniformity, grid non-uniformity, unused error correction, reflectance margin, print growth, contrast uniformity, aperture*). For *QR Code* and *Micro QR Code* also the *format information* and *version information* are returned as additional grading parameters.

The definition of the respective elements is as follows: The *overall quality* is the minimum of all individual grades. The *contrast* is the range between the minimal and the maximal pixel intensity in the data code domain, and a strong contrast results in a good grading. The *modulation* indicates how strong the amplitudes of the data code modules are. Big amplitudes make the assignment of the modules to black or white more certain, resulting in a high modulation grade. It is to note that the computation of the modulation grade is influenced by the specific level of error correction capacity, meaning that the modulation degrades less for codes with higher error correction capacity. The *contrast uniformity* is the minimum modulation value found in any module. This value is no grade, therefore, it can have real values. It does not affect the *overall quality*. According to ISO/IEC15415:2011, the *print growth* is no grade and is therefore not used for calculating the *overall quality*. Instead, it can be used as an additional information to find out, if the graphical features comprising the symbol have not shrunk or grown from their nominal size. This means that the *print growth* gives some indication to which extent the dark and light modules fill out their module boundaries. The calculation of the print growth does not follow the scheme as specified by ISO/IEC 15415:2011, but instead it considers the specifications as stated by the ISO/IEC standard of the 2D data code to be examined. This means that the computation of the *print growth* for *Aztec Codes* is implemented as described in ISO/IEC 24778:2008. For *ECC 200* the corresponding standard is ANSI/AIM International Specification Data Matrix and for *QR Codes* as well as for *Micro QR Codes* the implementation follows ISO/IEC 18004:2006. For all 2D data codes described above, the *print growth* is calculated in horizontal and vertical direction. The reported grade is the lower rated of the pair.

The *reflectance margin* also indicates (like the modulation) how strong the amplitudes of the data code modules are. The difference to *modulation* is that *reflectance margin* assessed the correct classification of the modules. The fixed pattern of both ECC 200, Aztec Code, QR Code, and Micro QR Code is of high importance for detecting and decoding the codes. Degradation or damage of the fixed pattern, or the respective quiet zones, is assessed with the *fixed pattern damage* quality, which is based on the modulation values.

The *decode* grade indicates if a code was successfully decoded. The *decode* quality is graded *4* when the code could be decoded according to the reference decode algorithm defined in the standard, and *0*, otherwise. Note that HALCON's decode algorithm differs from the reference decode algorithm. Thus, in many cases HALCON can decode the symbol although the decode grade according to the standard is *0*.

Originally, data codes have squared modules, i.e., the width and height of the modules are the same. Due to a potentially oblique view of the camera onto the data code or a defective fabrication of the data code itself, the width to height ratio can be distorted. This deterioration results in a degraded *axial non-uniformity*. If apart from an affine distortion the data code is subject to perspective or any other distortions too this degrades the *grid non-uniformity* quality. As data codes are redundant codes, errors in the modules or codewords can be corrected. The amount of error correcting capacities which is not already used by the present data code symbol is expressed in the *unused error correction* quality. In a way, this grade reflects the reliability of the decoding process. Note, that even codes with an unused

error correction grading of 0, which could possibly mean a false decoding result, can be decoded by the find_data_code_2d operator in a reliable way, because the implemented decoding functionality is more sophisticated and robust compared to the reference decode algorithm proposed by the standard.
For *QR Codes* and *Micro QR Codes* the additional grading parameters *format information* and *version information* are graded in a similar way to *fixed pattern damage*. If no *version information* exists, *'N/A'* is returned.
For the 2D stacked code *PDF417* the print quality is described in a tuple with eight elements: (overall quality, start/stop pattern, codeword yield, unused error correction, modulation, decodability, defects, aperture).
The definition of the respective elements is as follows: The *overall quality* is the minimum of all individual grades. As the PDF417 data code is a stacked code, which can be read by line scan devices as well, print quality assessment is mainly based on techniques for linear bar codes: a set of scan reflectance profiles is generated across the symbol followed by the evaluation of the respective print qualities within each scan, which are finally subsumed as overall print qualities. For more details, the user is referred to the standard for linear symbols ISO/IEC 15416:2016. In *start/stop pattern*, the start and stop patterns are assessed concerning the quality of the reflectance profile and the correctness of the bar and space sequence. The grade *codeword yield* counts and evaluates the relative number of correct decoded words acquired by the set of scan profiles. For the grade *unused error correction*, the relative number of false decoded words within the error correction blocks are counted. As for 2D data codes, the *modulation* grade indicates how strong the amplitudes, i.e., the extremal intensities, of the bars and spaces are. The grade *decodability* measures the deviation of the nominal length of bars and spaces with respect to their reference length. And finally, the grade *defects* refers to a measurement of how perfect the reflectance profiles of bars and spaces are.
The *aperture* is the size of the synthesized aperture in units of the module size of the symbol. This aperture is used to obtain the reference gray scale image during the grading procedure. It is defined in ISO/IEC 15415:2011, Chapter 7.3.3.
To be able to compute the quality values for QR Code, Micro QR Code, PDF417, and Aztec Code, it is necessary that the parameter *'persistence'* (see set_data_code_2d_param) is set to a value greater than or equal to *0*. For Data Matrix ECC 200, quality grading can also be performed with a *'persistence'* value of *-1*.

'quality_isoiec15415_values': returns a tuple with the raw values for all *'directly measurable'* grades (reported by *'quality_isoiec15415'*). These are grades, whose definition in the ISO/IEC 15415:2011 standard is a *'direct derivative'* of the reflectance (i.e., the gray values) or of geometrical properties of the symbol, or grades that are the result of a *'direct counting'*.
The returned tuple has the same order of elements as the corresponding result with *'quality_isoiec15415'*. For the grades, which are excluded from these lists, the operator reports *'N/A'*. For the 2D data codes *ECC 200*, *Aztec Code*, *QR Code*, and *Micro QR Code* the excluded grades are *overall quality*, *modulation*, *fixed pattern damage*, *decode* and *reflectance margin*. For the 2D stacked code *PDF417* the excluded grades are *overall quality*, *start/stop pattern*, *modulation*, *decodability*, *defects*.
Although the grades *modulation* (for *ECC 200*, *QR Code*, and *Micro QR Code*) and *modulation*, *decodability*, *defects* (for *PDF417*) generally are grading the reflectance properties of the symbol, the standard procedures for their computation involve the symbology decoding routine and error correction mechanism. Therefore there is no *'direct'* raw measurement underlying these grades. The grade *start/stop pattern* (for *PDF417*) is excluded because it is a complex grade, which does not correspond to a single raw measurement value.
All values except the *print growth* are normalized between *0.0* and *1.0*. Hence, for example, a *contrast* value of *0.75* will correspond to a gray value of *191.25* (for BYTE images). For the *print growth*, values between approximately *-0.9* and *0.9* are possible as an output. A negative value corresponds to a *print shrinkage* whereas a positive value reflects a *print growth*. The values can be interpreted as the percentage share that a dark module occupies too little (*print shrinkage*) or too much (*print growth*) of its nominal module size. A value of *-0.50* indicates for example that a dark module is 50% smaller as the nominal module size. If the *print growth* could not be calculated, the value is set to *'N/A'*.
To be able to compute the quality values for QR Code, Micro QR Code, PDF417, and Aztec Code, it is necessary that the parameter *'persistence'* (see set_data_code_2d_param) is set to a value greater than or equal to *0*. For Data Matrix ECC 200, quality grading can also be performed with a *'persistence'* value of *-1*.

- Data Matrix ECC 200:

'quality_isoiec15415_additional_reflectance_check': Returns the result of the additional reflectance check

described in ISO/IEC 15415:2011 7.7. If the check is successful or one of the grades for Modulation, Decode, or Fixed Pattern Damage is smaller than 1, *'passed'* is returned. If the reflectance interval in the extended area is within the reflectance interval of the code area including the quiet zone, *'passed'* is returned, as well. If the extended region required for this check is not completely contained within the image, *'fov_too_small'* is returned. If the check fails, *'failed'* is returned. In the exceptional case that the quality assessment fails and all grades are -1, *'unchecked'* will be returned.

'quality_isoiec15415_intermediate': Tuple with intermediate results that are determined during the assessment of print quality for ECC 200 codes in compliance with the international standard ISO/IEC 15415: 2011 and ISO/IEC 16022:2006. Using these values requires a thorough understanding of the underlying algorithms, and we strongly recommend to read ISO/IEC 15415:2011 and ISO/IEC 16022:2006 along with this documentation.

A description of the intermediate results is given below. The returned intermediate grades (*'quality_isoiec15415_intermediate'*) and values (*'quality_isoiec15415_intermediate_values'*) are the minimal grades and values for the code under inspection. The names of the available intermediate results can be queried with *'quality_isoiec15415_intermediate_labels'*.

- *'Rmin'* and *'Rmax'*: The minimum and maximum reflectance value in the sample area of the symbol (see Chapter 7.6, ISO/IEC 15415:2011). There are no corresponding grades for these values.
- *'L1'* and *'L2'*: The grades of the vertical and horizontal portions of the outside L of the fixed pattern as defined in Chapter M.1.2, ISO/IEC 16022:2006. The corresponding number of damaged modules is returned as value.
- *'QZL1'* and *'QZL2'*: The grades of the vertical and horizontal portions of the quiet zone adjacent to L1 and L2, respectively (see Chapter M.1.2, ISO/IEC 16022:2006). The corresponding number of damaged modules is returned as value.
- *'Transition Ratio'*: The grade for the transition ratio test described in Chapter M.1.3 b), ISO/IEC 16022:2006. The transition ratio TR is returned as value. It measures the ratio between the number of transitions on the clock track and the associated solid area.
- *'Clock track regularity'*: The grade for the clock track regularity test as described in Chapter M.1.3 e), ISO/IEC 16022:2006. If for any group of five adjacent modules more than two modules are considered as errors, this grade is 0, otherwise its 4. There is no corresponding value.
- *'Clock track damage'*: The grade for the clock track damage (see Chapter M.1.3 f), ISO/IEC 16022: 2006). The corresponding number of damaged modules is returned as value.
- *'Solid fix pattern'*: The grade for the solid fixed pattern (see Chapter M.1.3 g), ISO/IEC 16022: 2006). The corresponding number of damaged modules is returned as value.
- *'Clock track and adjacent solid pattern'*: The overall grade for the clock track and adjacent solid pattern (see Chapter M.1.3 k), ISO/IEC 16022:2006). There is no corresponding value.
- *'Average grade'*: The average grade for the for the fixed pattern. It is based on the grades for the L1, L2, QZL1, QZL2, and the overall grade for the clock track and adjacent solid pattern (see Chapter M.1.4, ISO/IEC 16022:2006). There is no corresponding value.

The grade for *fixed pattern damage* for the symbol is given by the minimum of the grades for *'L1'*, *'L2'*, *'QZL1'*, *'QZL2'*, *'Clock track and adjacent solid pattern'* and *'Average grade'* (see Chapter M1.4, ISO/IEC 16022:2006).

'quality_semi_t10_values': Tuple with the assessment of the print quality in compliance with the international standard SEMI T10-0701.

We recommend to use high quality images without artifacts like defocus, noise, overexposure, or inhomogeneous illumination. Such artifacts influence the result of the print quality grading. Further, to obtain robust grading results, an effective resolution of at least ten pixels per module in width and height is required. The quality grades are only computed when the symbol is fully in the image, elsewise -1 is returned for the quality grades.

The direct mark quality is described in a tuple with 21 elements: [P1 Row, P1 Column, P2 Row, P2 Column, P3 Row, P3 Column, P4 Row, P4 Column, Rows, Columns, Symbol Contrast, Symbol Contrast SNR, Horizontal Mark Growth, Vertical Mark Growth, Data Matrix Cell Width, Data Matrix Cell Height, Horizontal Mark Misplacement, Vertical Mark Misplacement, Cell Defects, Finder Pattern Defects, Unused Error Correction]. Note that the *Unused Error Correction* is returned for each Reed-Solomon block. Therefore, the actual length of the returned tuple depends on the number of Reed-Solomon blocks.

The definition of the respective elements is as follows: The first eight entries contain the coordinates of the four corners. The first corner is located at the vertical finder pattern. The remaining corners follow contour clockwise (for non mirrored codes). With *Rows* and *Columns* the numbers of rows and columns of the code are reported.

The value for *Symbol Contrast* reports the contrast between light and dark classified symbol pixels with respect to the full gray value range (255 for byte images) in percent. *Symbol Contrast SNR* is the corresponding signal-to-noise ratio. If the value is infinite, *'N/A'* is returned.
The values for *Horizontal Mark Growth* and *Vertical Mark Growth* represent the width and height, respectively, of marked modules with respect to the sum of the width and height, respectively, of a marked module and a space module in percent. A value of 50% is optimal. If some parts of the alternating pattern are hidden, *'N/A'* is returned.
The values for *Data Matrix Cell Width* and *Data Matrix Cell Height* report the average module width and height. They are computed from the four corner points and the number of rows and columns of the code.
The values for *Horizontal Mark Misplacement* and *Vertical Mark Misplacement* report the displacement of the alternating pattern marks' centers in horizontal and vertical direction, respectively. These values are given in percent with respect to *Data Matrix Cell Width* and *Data Matrix Cell Height*, respectively. If some parts of the alternating pattern are hidden, *'N/A'* is returned.
The value for *Cell Defects* reports the percentage of incorrectly classified symbol pixels.
The value for *Finder pattern Defects* reports the percentage of finder pattern pixels that would be classified incorrectly.
The value for *Unused Error Correction* reports the error correction capacities that are not already used by each Reed-Solomon block.

'quality_semi_t10_labels': Convenience value labels of the elements of the tuple returned when calling `get_data_code_2d_results` with *'quality_semi_t10_values'*. If the ECC 200 code has more than one Reed-Solomon block, the label tuple is extended by the corresponding values.

'quality_isoiec15415_reflectance_margin_module_grades': Tuple with the reflectance margin module grades that are determined during the assessment of print quality according to the ISO/IEC 15415:2011 standard Chapter 7.8.4.3. This includes the finder pattern modules and the 4 quiet zones adjacent to the symbol (2 horizontal and 2 vertical).
If the symbol size is $Rows \cdot Columns$, the tuple size is $(Rows + 2) \cdot (Columns + 2)$ as the 4 quiet zones are included. The tuple is sorted row by row, where the first row is the upper quiet zone and the last row is the bottom one (or QZL2 as described in ISO/IEC 16022:2006 Annex M.1.2, Figure M.1).
The computed grades for each module will be in the range 0 to 4. For unused data modules the module grade is not computed, in this case the corresponding tuple value will be -1.

'quality_isoiec15415_rows': Tuple with the image row coordinates of the points within the symbol modules for which the module grades are computed with the parameter *'quality_isoiec15415_reflectance_margin_module_grades'*.
The returned tuple has the same number and order of elements as the corresponding result with *'quality_isoiec15415_reflectance_margin_module_grades'*.

'quality_isoiec15415_cols': Tuple with the image column coordinates of the points within the symbol modules for which the module grades are computed with the parameter *'quality_isoiec15415_reflectance_margin_module_grades'*.
The returned tuple has the same number and order of elements as the corresponding result with *'quality_isoiec15415_reflectance_margin_module_grades'*.

'quality_isoiec29158_reflectance_margin_module_grades': Similar to *'quality_isoiec15415_reflectance_margin_module_grades'* but in compliance with the ISO/IEC TR 29158 (AIM DPM-1-2006) print quality standard.

'quality_aimdpm_1_2006_reflectance_margin_module_grades': See the entry for *'quality_isoiec29158_reflectance_margin_module_grades'*.

'quality_isoiec29158_reflectance_margin_module_float_grades': Similar to *'quality_isoiec29158_reflectance_margin_module_grades'* but in compliance with the ISO/IEC 29158: 2020 print quality standard. The underlying grades are returned with one decimal place.

'quality_isoiec29158_rows' **and** *'quality_aimdpm_1_2006_rows'*: Similar to *'quality_isoiec15415_rows'* but related to *'quality_isoiec29158_reflectance_margin_module_grades'* and *'quality_aimdpm_1_2006_reflectance_margin_module_grades'*.

'quality_isoiec29158_cols' **and** *'quality_aimdpm_1_2006_cols'*: Similar to *'quality_isoiec15415_cols'* but related to *'quality_isoiec29158_reflectance_margin_module_grades'* and *'quality_aimdpm_1_2006_reflectance_margin_module_grades'*.

- Data Matrix ECC 200, Aztec Code, QR Code and Micro QR Code:

'quality_isoiec29158' **and** *'quality_aimdpm_1_2006'***:** Tuple with the assessment of print quality in compliance with ISO/IEC TR 29158. This standard was previously established by the Automatic Identification Manufacturers as AIM DPM-1-2006. ISO/IEC TR 29158 is an extension to ISO/IEC 15415:2011 standard, which defines certain requirements for the gray-scale properties of the data code image and in doing so improves the reproducibility of the grading results among different vendors.
Using these values requires a thorough understanding of the underlying algorithms, and we recommend to read ISO/IEC TR 29158 (AIM DPM-1-2006) and ISO/IEC 15415:2011 along with this documentation.
It is very important to note that an ISO/IEC TR 29158 (AIM DPM-1-2006) standard conforming print quality assessment requires interactive image acquisition! No images of a data code symbol should be regarded if there is no information for the camera-lighting setup used to acquire these images. In most cases this implies that the user should set up and use her/his own camera-lighting configuration. Refer to ISO/IEC TR 29158 (AIM DPM-1-2006) for some prescribed camera-lighting configurations and general configuration principles that are relevant for the standard. Note also that, even though the implementation is strictly based on the standard, the computation of the print quality grades depends on the decoding algorithm used. Thus, the print quality inspection results are relevant only for the 2D data code reader implemented with HALCON.
Calling `get_data_code_2d_results` with argument *'quality_isoiec29158'* or *'quality_aimdpm_1_2006'* assumes that the image being processed fulfills the two criteria explained in detail below. The returned tuple represents the 13 print quality elements: [overall quality, cell contrast, cell modulation, fixed pattern damage, decode, axial non-uniformity, grid non-uniformity, unused error correction, mean light, reflectance margin, print growth, contrast uniformity, aperture]. The first eight as well as the last four grades have the same meaning as the ISO/IEC 15415:2011 grades with two exceptions: *contrast* and *modulation* are renamed to *cell contrast* and *cell modulation*, respectively, to reflect differences in the methods specified with both standards for estimating those values. The value of *mean light* is not a grade specified with ISO/IEC TR 29158 (AIM DPM-1-2006) standard. It is an estimation for the quality of the processed image computed as the mean gray-scale value of the centers of the light data code modules. It is a value between 0.0 and 1.0, corresponding to 0% to 100% of the maximum gray-scale value (255 for byte images). For *QR Code* and *Micro QR Code* also the *format information* and *version information* are returned as additional grading parameters.
The essence of the standard is a procedure for the adjustment of the camera system response (SR): *exposure time*, *gain* and/or another specific setting of the user's camera-lighting setup. The goal is to obtain an image of the inspected data code symbol with optimal gray-scale properties. In the image processing context, which is in the main focus of the standard, this is also the key to reproducibility of the grading results. An image has the required gray-scale properties when its *'mean light'* value is between 70% and 86% (*0.70* and *0.86*, respectively). Trying to achieve this condition for images of physically low-contrast data code symbols results usually in very high SR levels. Therefore, the standard defines additionally a calibration routine, which identifies for comparison purposes the normal (calibrated) system response settings (SRcal) by evaluating images of an object with well-known reflectance properties (such as NIST traceable calibrated conformance test card). Finally, the **two image acquisition criteria** for an image read: **first**, the image must have a *mean light* value between 70% and 86% (*0.70* and *0.86*, respectively); **second**, it has to be obtained with system response (SR) settings fulfilling: SR/SRcal ≤ 16. If no image of the data code symbol can be obtained to fulfill both criteria, the symbol must be graded with 0.
As neither the adjustment of the system response nor the verification of the SR criterion can be carried out from within a HALCON operator, the assessment of the AIM DPM-1-2006 quality grades should be performed in an appropriate program setup. The user is referred to two example programs provided with her/his installation of HALCON (calibration_aimdpm_1_2006.hdev and print_quality_aimdpm_1_2006.hdev in the directory HALCONEXAMPLES/hdevelop/Identification/Data-Code), which demonstrate the calibration and the grade evaluation routines, respectively. Both image criteria, as well as further details regarding the general camera setup and image acquisition are also explained and demonstrated there. Everyone who intends to use the AIM DPM-1-2006 grading standard can use the programs as a starting point and adapt them for her/his own camera-lighting setup.
The *aperture* is the size of the synthesized aperture in units of the module size of the symbol. This aperture is used to obtain the reference gray scale image during the grading procedure and selected by the procedure described in ISO/IEC TR 29158 (AIM DPM-1-2006), Chapter 7.3.
To be able to compute the quality values for QR Code, Micro QR Code, PDF417, and Aztec Code, it is necessary that the parameter *'persistence'* (see `set_data_code_2d_param`) is set to a value greater than or equal to *0*. For Data Matrix ECC 200, quality grading can also be performed with a *'persistence'*

value of *-1*.

'quality_isoiec29158_values' **and** *'quality_aimdpm_1_2006_values'***:** Similar to *'quality_isoiec15415_values'*, the raw values for all *'directly measurable'* grades are reported but in compliance with the ISO/IEC TR 29158 (AIM DPM-1-2006) print quality standard. The list of the reported and the excluded raw measurement values is the same as with *'quality_isoiec15415_values'*. The auxiliary ISO/IEC TR 29158 (AIM DPM-1-2006) grade *mean light* is reported in the same format (between 0.0 and 1.0) both by *'quality_isoiec29158'* and *'quality_isoiec29158_values'* (*'quality_aimdpm_1_2006'* and *'quality_aimdpm_1_2006_values'*).
To be able to compute the quality values for QR Code, Micro QR Code, PDF417, and Aztec Code, it is necessary that the parameter *'persistence'* (see `set_data_code_2d_param`) is set to a value greater than or equal to *0*. For Data Matrix ECC 200, quality grading can also be performed with a *'persistence'* value of *-1*.

'quality_isoiec29158_float_grades' Similar to *'quality_isoiec29158'*, but with continuous grading following ISO/IEC 29158:2020. The underlying grades are returned with one decimal place.
To be able to compute the quality values for QR Code, Micro QR Code, and Aztec Code, it is necessary that the parameter *'persistence'* (see `set_data_code_2d_param`) is set to a value greater than or equal to *0*. For Data Matrix ECC 200, quality grading can also be performed with a *'persistence'* value of *-1*.

- Data Matrix ECC 200:

'quality_isoiec29158_intermediate' **and** *'quality_aimdpm_1_2006_intermediate'***:** Tuple with intermediate results that are determined during the assessment of print quality of ECC 200 codes in compliance with the international standards ISO/IEC TR 29158 (or AIM DPM-1-2006), ISO/IEC 15415:2011 and ISO/IEC 16022:2006. Using these values requires a thorough understanding of the underlying algorithms, and we strongly recommend to read ISO/IEC TR 29158 (AIM DPM-1-2006), ISO/IEC 15415:2011 and ISO/IEC 16022:2006 along with this documentation.
A description of the intermediate results is given below. The returned intermediate grades (*'quality_isoiec29158_intermediate'* or *'quality_aimdpm_1_2006_intermediate'*) and values (*'quality_isoiec29158_intermediate_values'* or *'quality_aimdpm_1_2006_intermediate_values'*) are the minimal grades and values for the code under inspection. The names of the available intermediate results can be queried with *'quality_isoiec29158_intermediate_labels'* or *'quality_aimdpm_1_2006_intermediate_labels'*.

  - *'T2'*: The threshold created using the histogram of the reference gray-scale image pixel values at each intersection point of the grid using the method defined in Annex A, ISO/IEC TR 29158 (AIM DPM-1-2006).
  - *'MD'*: The value for MeanDark corresponding to T2, which is determined using the method defined in Annex A, ISO/IEC TR 29158 (AIM DPM-1-2006).
  - *'MLtarget'*: The value for MeanLight corresponding to T2, which is determined using the method defined in Annex A, ISO/IEC TR 29158 (AIM DPM-1-2006).
  - *'L1'* and *'L2'*: The grades of the vertical and horizontal portions of the outside L of the fixed pattern as defined in Chapter M.1.2, ISO/IEC 16022:2006. The corresponding number of damaged modules is returned as value.
  - *'QZL1'* and *'QZL2'*: The grades of the vertical and horizontal portions of the quiet zone adjacent to L1 and L2, respectively (see Chapter M.1.2, ISO/IEC 16022:2006). The corresponding number of damaged modules is returned as value.
  - *'Transition Ratio'*: The grade for the transition ratio test described in Chapter M.1.3 b), ISO/IEC 16022:2006. The transition ratio TR is returned as value. It measures the ratio between the number of transitions on the clock track and the associated solid area.
  - *'Clock track regularity'*: The grade for the clock track regularity test as described in Chapter M.1.3 e), ISO/IEC 16022:2006. If for any group of five adjacent modules more than two modules are considered as errors, this grade is 0, otherwise its 4. There is no corresponding value.
  - *'Clock track damage'*: The grade for the clock track damage (see Chapter M.1.3 f), ISO/IEC 16022:2006). The corresponding number of damaged modules is returned as value.
  - *'Solid fix pattern'*: The grade for the solid fixed pattern (see Chapter M.1.3 g), ISO/IEC 16022:2006). The corresponding number of damaged modules is returned as value.
  - *'Clock track and adjacent solid pattern'*: The overall grade for the clock track and adjacent solid pattern (see Chapter M.1.3 k), ISO/IEC 16022:2006). There is no corresponding value.

  - *'Distributed damage grade'*: The average grade for the for the fixed pattern. It is based on the grades for the L1, L2, QZL1, QZL2, and the overall grade for the clock track and adjacent solid pattern (see Chapter M.1.4, ISO/IEC 16022:2006, and Chapter 9.4, ISO/IEC TR 29158 (AIM DPM-1-2006)). There is no corresponding value.

  The grade for *fixed pattern damage* for the symbol is given by the minimum of the grades for *'L1'*, *'L2'*, *'QZL1'*, *'QZL2'*, *'Clock track and adjacent solid pattern'* and *'Distributed damage grade'* (see Chapter M1.4, ISO/IEC 16022:2006).

'quality_isoiec29158_intermediate_float_grades': Similar to *'quality_isoiec29158_intermediate'* but in compliance with the ISO/IEC 29158 print quality standard. The underlying grades are returned with one decimal place.

- Data Matrix ECC 200, QR Code, Aztec Code, and DotCode:

  'structured_append': If the symbol is part of a group of symbols ("Structured Append"), this parameter contains (1) the index of the symbol in the group, (2) the number of symbols that belong to the group, and (3) a number that serves as a group identifier. For Aztec Codes the group identifier is a string. If the symbol is not part of a group of symbols ("Structured Append") a empty tuple is returned in case of Aztec Codes.

  'gs1_lint_passed': If the symbol contains a GS1 formatted string and the data code model was created to read GS1 codes, additional linting can be performed on the contained string. *'gs1_lint_passed'* will return *'true'*, if this is successful. Otherwise, *'false'* is returned. Linting checks that
  - one AI (Application Identifier) does not occur multiple times with different contents,
  - AIs required by other AIs are contained,
  - AIs excluded by other AIs are not included, and that
  - specific formatting rules for AIs are respected.

  'gs1_lint_result': *'gs1_lint_result'* performs the linting tests documented in *'gs1_lint_passed'*. If the linting fails, a list of human-readable error messages is returned, if it is successful, *'ok'* is returned.

- DotCode:

  DotCodes are able to encode separate data messages in one single DotCode symbol. According to the AIM International Symbology Specification for DotCode, such separated messages should be treated as if they come from different symbols. In the following list, all results are listed which can be queried for DotCodes and need a dedicated result for each separated message segment. They are all postfixed with *'_separated'*. They return a tuple with the results separated for each separated message segment. In case of a symbol with separated messages the correspondent results without *'_separated'* postfix returns the result for the first separated message segment. Note: If one of the separated messages can not be decoded, no results are returned.

  'symbology_ident_separated': see *'symbology_ident'* for details.

  'decoded_string_separated': see *'decoded_string'* for details.

  'structured_append_separated': see *'structured_append'* for details.

  'reader_programming_separated': see *'reader_programming'* for details.

  'decoded_data_separated': see *'decoded_data'* for details.

  'segment_num': Number of separated message segments.

Status message

The status parameter that can be queried for all candidates reveals why and where in the evaluation phase a candidate was discarded. The following list shows the most important status messages in the order of their generation during the evaluation phase:

- Data Matrix ECC 200:

  'aborted: too close to image border': The symbol candidate is too close to the border. Only symbols that are completely within the image can be read.

  'aborted adjusting: ...': It is not possible to determine the exact position of the finder pattern in the processing image.

  'aborted finder pattern: ...': It is not possible to determine the width of one of the two legs of the L-shaped finder pattern.

'aborted leg widths: widths of the finder pattern legs differ too much': The widths of the two legs of the L-shaped finder pattern differ too much.

'aborted alternating side: ...': For one dimension of the candidate, two opposite borders were found during the symbol search phase. However, it is not possible to determine which is the alternating and which the solid side of the finder pattern.

'aborted border search: ...': For one dimension of the candidate, only the border that belongs to the solid side of the finder pattern was found during the symbol search phase. Searching the opposite (the alternating) side failed.

'aborted symbol: invalid size': The number of rows and columns of the symbol that was deduced from the alternating pattern does not yield in a valid ECC 200 code.

'aborted symbol: size does not fit strict model definition': Although the deduced symbol size is a valid ECC 200 size, it is not inside the range predefined by the model.

'aborted symbol: rectangular symbol does not fit strict mirror definition of model': The symbol was identified as a rectangular ECC 200 code. In conjunction with the mirroring parameter of the model, however, the symbol's rows and columns are swapped such that no valid ECC 200 code is achieved. This test is of course not possible for square symbols. There, a wrong mirroring specification will affect the reading of the symbol data and, in general, lead to the following error:

'error correction failed': The error correction failed because there are too many modules that couldn't be interpreted correctly. Normally, this indicates that the print and/or image quality is too bad, but it may also be provoked by a wrong mirroring specification in the model.

'decoding failed: special decoding reader requested': The decoded data contains a message for programming the data code reader. This feature is not supported.

'decoding failed: inconsistent data': The data coded in the symbol is not consistent and therefore cannot be read.

'decoding failed: invalid GS1 format': The data encoded in the symbol is not conform to the GS1 General Specifications. This indicates an attempt to read a non-GS1 symbol with a data code model created as GS1 reader.

'quiet zone is missing': The quiet zone of this candidate is missing or shows severe defects.

'invalid finder pattern or quiet zone check failed': The candidate could be decoded, but was discarded as possible false positive, because the finder pattern or quiet zone could not be found. This additional check is performed for symbols of size 10x10, 12x12 and 8x18, as these codes have little error correction and false positives may easily be found in unrelated texture.

- Data Matrix ECC 200 (if a high finder pattern tolerance is chosen, the following messages may occur additionally):

  'aborted module grid search: no regular module grid found' No regular module grid structure could be found within the candidate region. Therefore, the candidate is assumed to be no ECC 200 symbol.

  'aborted border search: overlap with previous result' The current candidate overlaps with a previously found result. Therefore, the search for a valid border was aborted.

  'aborted border search: no consistent border found' No consistent border that leads to a symbol size that is within the range of valid symbol sizes leads to a successful decoding.

  'aborted reading: result is likely to be random' The candidate could be decoded. However, because the symbol size was small and the code had too many decoding errors, the result is very likely to be random, and hence was rejected. This heuristic is applied for symbols of size 10×10 with at least 1 decoding error and for symbols of size 12×12 or 8×18 with more than 1 decoding error.

- QR Code:

  'aborted: too close to image border': The symbol candidate is too close to the border. Only symbols that are completely within the image can be read.

  'aborted adjusting: finder patterns': It is not possible to determine the exact position of the finder pattern in the processing image.

  'aborted symbol: different number of rows and columns': It is not possible to determine for both dimensions a consistent symbol size by the size and the position of the detected finder pattern. When reading Model 2 symbols, this error may occur only with small symbols ($<$ version 7 or 45×45 modules). For bigger symbols the size is coded within the symbol in the version information region. The estimated size is used only as a hint for finding the version information region.

'aborted symbol: invalid size': The size determined by the size and the position of the detected finder pattern is too small or (only Model 1) too big.

'decoding of version information failed': While processing a Model 2 symbol, the symbol version as determined by the finder pattern is at least 7 ($\geq 45 \times 45$ modules). However, reading the version from the appropriate region in the symbol failed.

'aborted symbol: size does not fit strict model definition': Although the deduced symbol size is valid, it is not inside the range predefined by the model.

'decoding of format information failed': Reading the format information (mask pattern and error correction level) from the appropriate region in the symbol failed.

'error correction failed': The error correction failed because there are too many modules that couldn't be interpreted correctly. Normally, this indicates that the print and/or image quality is too bad, but it may also be provoked by a wrong mirroring specification in the model.

'decoding failed: inconsistent data': The data coded in the symbol is not consistent and therefore cannot be read.

'decoding failed: invalid GS1 format': The data encoded in the symbol is not conform to the GS1 General Specifications. This indicates an attempt to read a non-GS1 symbol with a data code model created as GS1 reader.

- Aztec Code:

  'aborted: too close to image border': The symbol candidate is too close to the border or even partially outside the image.

  'aborted: mode message could not be decoded': The mode message of this candidate could not be decoded because there are too many modules that could not be interpreted correctly.

  'aborted: symbol expansion failed': The expansion of the candidate failed. Most likely because of too heavy perspective distortions.

  'aborted symbol: size does not fit strict model definition': Although the deduced symbol size is valid, it is not inside the range predefined by the model. Adapt the model (`set_data_code_2d_param`) to include this candidate in the final results.

  'aborted symbol: rectangular symbol does not fit strict mirror definition of model': Although this candidate is valid, it does not fit the mirroring constraint predefined by the model. Adapt the model (`set_data_code_2d_param`) to include this candidate in the final results.

  'aborted symbol: invalid format': Although this candidate is valid, it does not fit the format predefined by the model. Adapt the model (`set_data_code_2d_param`) to include this candidate in the final results.

  'error correction failed': The error correction failed because there are too many modules that could not be interpreted correctly. Normally, this indicates that the print and/or image quality is too bad, but it may also be provoked by a wrong mirroring specification in the model.

  'decoding failed: invalid GS1 format': The data encoded in the symbol is not conform to the GS1 General Specifications. This indicates an attempt to read a non-GS1 symbol with a data code model created as GS1 reader.

- PDF417:

  'aborted: too close to image border': The symbol candidate is too close to the border. Only symbols that are completely within the image can be read.

  'aborted symbol: size does not fit strict model definition': Although the deduced symbol size is valid, it is not inside the range predefined by the model.

  'error correction failed': The error correction failed because there are too many modules that couldn't be interpreted correctly. Normally, this indicates that the print and/or image quality is too bad, but it may also be provoked by a wrong mirroring specification in the model.

  'decoding failed: special decoding reader requested': The decoded data contains a message for programming the data code reader. This feature is not supported.

  'decoding failed: inconsistent data': The data coded in the symbol is not consistent and therefore cannot be read.

- DotCode:

  'aborted module grid search: no regular module grid found' No regular module grid structure could be found within the candidate region. Therefore, the candidate is assumed to be no DotCode symbol.

'aborted symbol: size does not fit strict model definition': Although the deduced symbol size is a valid DotCode size, it is not inside the range predefined by the model.

'error correction failed': The error correction failed because there are too many modules that couldn't be interpreted correctly. Normally, this indicates that the print and/or image quality is too bad, but it may also be provoked by a wrong mirroring specification in the model.

'decoding failed: inconsistent data': The data coded in the symbol is not consistent and therefore cannot be read.

'unmasking: invalid mask type': The first two bits of each DotCode encoded in the data stream of the code determine the so-called mask. The mask is used during encoding process in order to increase graphical robustness. Possible values are 0, 1, 2 and 3. If any other value is encoded in the first two bits, the candidate is marked with this status.

'quiet zone is missing': The quiet zone of this candidate is missing or shows severe defects.

'decoding failed: invalid GS1 format': The data encoded in the symbol is not conform to the GS1 General Specifications. This indicates an attempt to read a non-GS1 symbol with a data code model created as GS1 reader.

While processing a candidate, it is possible that internally several iterations for reading the symbol are performed. If all attempts fail, normally the last abortion state is stored in the candidate structure. E.g., if the QR Code model enables symbols with Model 1 and Model 2 specification, `find_data_code_2d` tries first to interpret the symbol as Model 2 type. If this fails, Model 1 interpretation is performed. If this also fails, the status variable is set to the latest failure state of the Model 1 interpretation. In order to get the error state of the Model 2 branch, the *'model_type'* parameter of the data code model must be restricted accordingly (with `set_data_code_2d_param`).

Parameters

▷ **`DataCodeHandle`** (input_control) . . . . . datacode_2d ⇝ *handle*
Handle of the 2D data code model.

▷ **`CandidateHandle`** (input_control) . . . . . integer ⇝ *string* / integer
Handle of the 2D data code candidate. Either an integer (usually the ResultHandle of `find_data_code_2d`) or a string representing a group of candidates.
Default: 'all_candidates'
Suggested values: `CandidateHandle` ∈ {0, 1, 2, 'general', 'all_candidates', 'all_results', 'all_undecoded', 'all_aborted'}

▷ **`ResultNames`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the results of the 2D data code to return.
Default: 'status'
List of values: `ResultNames` ∈ {'min_search_level', 'max_search_level', 'pass_num', 'result_num', 'candidate_num', 'undecoded_num', 'aborted_num', 'handle', 'pass', 'status', 'search_level', 'process_level', 'polarity', 'module_gap', 'mirrored', 'model_type', 'symbol_rows', 'symbol_cols', 'symbol_size', 'version', 'module_height', 'module_width', 'small_modules_robustness', 'module_aspect', 'slant', 'contrast', 'module_grid', 'decoded_string', 'decoding_error', 'symbology_ident', 'mask_pattern_ref', 'alternating_pattern_tolerance', 'finder_pattern_tolerance', 'contrast_tolerance', 'error_correction_level', 'bin_module_data', 'raw_coded_data', 'corr_coded_data', 'decoded_data', 'gs1_lint_passed', 'gs1_lint_result', 'quality_isoiec15415', 'quality_isoiec15415_labels', 'quality_isoiec15415_values', 'quality_isoiec15415_float_grades', 'quality_isoiec15415_reflectance_margin_module_grades', 'quality_isoiec15415_rows', 'quality_isoiec15415_cols', 'quality_isoiec15415_additional_reflectance_check', 'quality_aimdpm_1_2006', 'quality_aimdpm_1_2006_labels', 'quality_aimdpm_1_2006_values', 'quality_aimdpm_1_2006_reflectance_margin_module_grades', 'quality_aimdpm_1_2006_rows', 'quality_aimdpm_1_2006_cols', 'quality_isoiec29158', 'quality_isoiec29158_labels', 'quality_isoiec29158_values', 'quality_isoiec29158_reflectance_margin_module_grades', 'quality_isoiec29158_reflectance_margin_module_float_grades', 'quality_isoiec29158_rows', 'quality_isoiec29158_cols', 'quality_isoiec29158_float_grades', 'quality_isoiec15415_intermediate', 'quality_isoiec15415_intermediate_labels', 'quality_isoiec15415_intermediate_values', 'quality_aimdpm_1_2006_intermediate', 'quality_aimdpm_1_2006_intermediate_labels', 'quality_aimdpm_1_2006_intermediate_values', 'quality_isoiec29158_intermediate', 'quality_isoiec29158_intermediate_labels', 'quality_isoiec29158_intermediate_values', 'quality_isoiec29158_intermediate_float_grades', 'quality_semi_t10_values', 'quality_semi_t10_labels',

'structured_append', 'macro_exist', 'macro_segment_index', 'macro_file_id', 'macro_segment_count', 'macro_time_stamp', 'macro_checksum', 'macro_last_symbol', 'reader_programming', 'aborted', 'orientation', 'layer_num', 'codeword_num', 'format', 'symbology_ident_separated', 'decoded_string_separated', 'structured_append_separated', 'reader_programming_separated', 'decoded_data_separated', 'segment_num'}

▷ **ResultValues** (output_control) ..... attribute.value(-array) ⇝ *string* / integer / real
List with the results.

Example

```
* Example demonstrating how to access the results of the data code search.

* Create a model for reading Data matrix ECC 200 codes
create_data_code_2d_model ('Data Matrix ECC 200', [], [], DataCodeHandle)
* Read an image
read_image (Image, 'datacode/ecc200/ecc200_cpu_010')
* Read the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], \
                   ResultHandles, DecodedDataStrings)

* Get the number of passes
get_data_code_2d_results (DataCodeHandle, 'general', 'pass_num', Passes)

* Get a tuple with the status of all candidates
get_data_code_2d_results (DataCodeHandle, 'all_candidates', 'status', \
                          AllStatus)
* Get the handles of all candidates that were detected as a symbol but
* could not be read
get_data_code_2d_results (DataCodeHandle, 'all_undecoded', 'handle', \
                          HandlesUndecoded)

* For every undecoded symbol, get the contour, the symbol size, and
* the binary module data
dev_set_color ('red')
for i := 0 to |HandlesUndecoded| - 1 by 1
    * Get the contour of the symbol
    get_data_code_2d_objects (SymbolXLD, DataCodeHandle, \
                              HandlesUndecoded[i], 'candidate_xld')
    * Get the symbol size
    get_data_code_2d_results (DataCodeHandle, HandlesUndecoded[i], \
                              ['symbol_rows','symbol_cols'], SymbolSize)
    * Get the binary module data (has to be queried exclusively)
    get_data_code_2d_results (DataCodeHandle, HandlesUndecoded[i], \
                              'bin_module_data', BinModuleData)
    * Stop for inspecting the data
    stop ()
endfor

* Clear the model
clear_data_code_2d_model (DataCodeHandle)
```

Result

The operator `get_data_code_2d_results` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct and the requested results are available for the last symbol search. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on internal data level.

Possible Predecessors

find_data_code_2d, query_data_code_2d_params

Possible Successors

get_data_code_2d_objects

See also

query_data_code_2d_params, get_data_code_2d_objects, get_data_code_2d_param, set_data_code_2d_param

References

AIM Global Document: AIM DPM-1-2006: "Direct Part Mark (DPM) Quality Guideline"; Document Type: AIM Bar Code Guideline; Document Version: 1.0, 2006-12-12.

International Standard ISO/IEC 15415: "Information technology - Automatic identification and data capture techniques - Bar code symbol print quality test specification - Two-dimensional symbols"; Reference number ISO/IEC 15415:2011 (E); ISO/IEC 2011.

International Standard ISO/IEC 16022: "Information technology - Automatic identification and data capture techniques - Data Matrix bar code symbology specification"; Reference number ISO/IEC 16022:2006 (E); ISO/IEC 2006.

International Standard ISO/IEC 15438: "Information technology - Automatic identification and data capture techniques - PDF417 bar code symbology specification"; Reference number ISO/IEC 15438:2006 (E); ISO/IEC 2006.

International Standard ISO/IEC 18004: "Information technology - Automatic identification and data capture techniques - QR Code 2005 bar code symbology specification"; Reference number ISO/IEC 18004:2006 (E); ISO/IEC 2006.

International Standard ISO/IEC 24778: "Information technology - Automatic identification and data capture techniques - Aztec Code bar code symbology specification"; Reference number ISO/IEC 24778:2008 (E); ISO/IEC 2008.

Technical Report ISO/IEC TR 29158: "Information technology - Automatic identification and data capture techniques - Direct Part Mark (DPM) Quality Guideline"; Reference number ISO/IEC TR 29158:2011 (E); ISO/IEC 2011.

GS1 General Specifications; Version 12; Issue 1, Jan-2012; GS1.

Module

Data Code

```
query_data_code_2d_params ( : : DataCodeHandle,
    QueryName : GenParamName )
```

Get for a given 2D data code model the names of the generic parameters or objects that can be used in the other 2D data code operators.

The operator query_data_code_2d_params returns the names of the generic parameters that are supported by the 2D data code operators set_data_code_2d_param, get_data_code_2d_param, find_data_code_2d, get_data_code_2d_results, and get_data_code_2d_objects.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

The parameter QueryName is used to select the desired **parameter group**:

'get_model_params': get_data_code_2d_param – Parameters for querying the 2D data code model.

'set_model_params': set_data_code_2d_param – Parameters for adjusting the 2D data code model.

'find_params': find_data_code_2d – Parameters used while searching and reading the 2D data code symbols.

'get_result_params': get_data_code_2d_results – Parameters for querying the alphanumerical results of the symbol search.

'get_result_objects': get_data_code_2d_objects – Parameters for accessing the iconic objects of the symbol search.

'trained': set_data_code_2d_param – Parameters whose values were determined by training. The next training will not reset these parameters, but extend the parameter space if necessary.

The returned parameter list depends only on the type of the data code and not on the current state of the model or its results.

Parameters

▷ **DataCodeHandle** (input_control) ..... datacode_2d ⇝ *handle*
Handle of the 2D data code model.

▷ **QueryName** (input_control) ..... attribute.name ⇝ *string*
Name of the parameter group.
Default: 'get_result_params'
List of values: QueryName ∈ {'get_model_params', 'set_model_params', 'find_params', 'get_result_params', 'get_result_objects', 'trained'}

▷ **GenParamName** (output_control) ..... attribute.value-array ⇝ *string*
List containing the names of the supported generic parameters.

Example

```
* This example demonstrates how the names of all available model parameters
* can be queried.  This is used to request first the settings of the
* untrained and then the settings of the trained model.

* Create a model for reading Data matrix ECC 200 codes
create_data_code_2d_model ('Data Matrix ECC 200', [], [], DataCodeHandle)
* Query all the names of the generic parameters that can be passed to the
* operator get_data_code_2d_param to request the model
query_data_code_2d_params (DataCodeHandle, 'get_model_params', GenParamName)
* Request the current settings of the (untrained) model
get_data_code_2d_param(DataCodeHandle, GenParamName, ModelParams)

* Read a training image
read_image (Image, 'datacode/ecc200/ecc200_cpu_007')
* train the model with the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, 'train', 'all', \
                   ResultHandles, DecodedDataStrings)
* Request the current settings of the (now trained) model
get_data_code_2d_param(DataCodeHandle, GenParamName, TrainedModelParams)
* Create a tuple that demonstrates the changings
ModelAdaption := GenParamName + ':    ' + ModelParams + '  ->  ' + \
                 TrainedModelParams

* Clear the model
clear_data_code_2d_model (DataCodeHandle)
```

Result

The operator query_data_code_2d_params returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_data_code_2d_model

Possible Successors

get_data_code_2d_param, get_data_code_2d_results, get_data_code_2d_objects

Module

Data Code

read_data_code_2d_model (: : FileName : DataCodeHandle)

Read a 2D data code model from a file and create a new model.

The operator read_data_code_2d_model reads the 2D data code model file FileName and creates a new model that is an identical copy of the saved model. The parameter DataCodeHandle returns the handle of the new model. The model file FileName must be created by the operator write_data_code_2d_model. The default HALCON file extension for the 2D data code model is 'dcm'.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the 2D data code model file.
Default: 'data_code_model.dcm'
File extension: .dcm

▷ **DataCodeHandle** (output_control) ..... datacode_2d ⇝ *handle*
Handle of the created 2D data code model.

Example

```
* This example demonstrates how a model that was saved in an earlier
* session can be used again by reading the model file

* Create a model by reading by reading a data code model file
read_data_code_2d_model ('ecc200_trained_model.dcm', DataCodeHandle)
* Read a symbol image
read_image (Image, 'datacode/ecc200/ecc200_cpu_010')
* Read the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], \
                   ResultHandles, DecodedDataStrings)
* Clear the model
clear_data_code_2d_model (DataCodeHandle)
```

Result

The operator read_data_code_2d_model returns the value 2 (H_MSG_TRUE) if the named 2D data code file was found and correctly read. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

find_data_code_2d

Alternatives

create_data_code_2d_model

See also

write_data_code_2d_model, clear_data_code_2d_model

Module

Data Code

serialize_data_code_2d_model (: : DataCodeHandle : SerializedItemHandle)

Serialize a 2D data code model.

serialize_data_code_2d_model serializes a 2D data code model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_data_code_2d_model is converted to a serialized item. The 2D data code model is defined by the handle DataCodeHandle. The serialized 2D data code model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_data_code_2d_model.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

Parameters

▷ **DataCodeHandle** (input_control) ..... datacode_2d ⇝ *handle*
Handle of the 2D data code model.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_data_code_2d_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_data_code_2d_param, find_data_code_2d

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_data_code_2d_model

Alternatives

get_data_code_2d_param

See also

create_data_code_2d_model, set_data_code_2d_param, find_data_code_2d

Module

Data Code

set_data_code_2d_param (: : DataCodeHandle, GenParamName, GenParamValue :)

Set selected parameters of the 2D data code model.

The operator set_data_code_2d_param is used to set or change the different parameters of a 2D data code model in order to adapt the model to a particular symbol appearance. All parameters can also be set while creating a 2D data code model with create_data_code_2d_model. The current configuration of the data code model can be queried with get_data_code_2d_param. A list with the names of all parameters that can be set for the given 2D data code type is returned by query_data_code_2d_params.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

Note that the symbol structure of GS1 DataMatrix, GS1 QR Code, GS1 Aztec Code, and GS1 DotCode is identical to the structure of Data Matrix ECC 200, QR Code, Aztec Code, and DotCode, respectively. Therefore, all symbology specific parameters applying to Data Matrix ECC 200, QR Code, Aztec Code, or DotCode apply to their corresponding GS1variant as well. In the following, the explicit enumeration of the parameters for any particular GS1

code is omitted for sake of readability. Instead, the relevant parameters names are to be inferred from the parameters for the corresponding non-GS1 code type or can be explicitly queried by `query_data_code_2d_params` with parameter *'set_model_params'*.

The following overview lists the different generic parameters with the respective value ranges and default values in standard mode (*'standard_recognition'*). If the default values in enhanced mode (*'enhanced_recognition'*) differ from those in the standard mode, they are listed additionally. The same holds, if the default values in the maximum mode *'maximum_recognition'* differ from those in the enhanced mode:

Basic default settings:

- All data code types:

  'default_parameters': Reset all model parameters to one of the three basic default settings *standard*, *enhanced*, or *maximum* (see the following summary and `create_data_code_2d_model`). In addition to the parameter values, the training state of the model is reset.
  List of values: *'standard_recognition'*, *'enhanced_recognition'*, *'maximum_recognition'*
  Default: *'standard_recognition'*
  Attention: If this parameter is set together with a list of other parameters, this parameter must be at the first position.

  'trained': Set for the given parameters the training state to trained. The next training will not overwrite this parameters. If necessary only the parameter range is extended to cover the already trained symbols. For all possible parameters which can be trained see `find_data_code_2d`. With *'all'* the training state of all trainable parameters is set to trained.

Size and shape of the symbol:

- Data Matrix ECC 200 (including the finder pattern):

  Attention: After calling `set_data_code_2d_param` to change size or shape of the symbol, the parameters are evaluated in the given order. After each parameter, the consistency of the current settings is checked, and if necessary the current settings are adjusted. Therefore the order in which the changes are done may influence the final setting.

  'symbol_cols_min': Minimum number of module columns in the symbol.
  Value range: [*10*, *12*, *14*, ... *144*]
  Attention: Changing *'symbol_cols_min'* sets *'symbol_shape'* to *'any'*. If the size of the current symbol is only consistent with one shape type *'symbol_shape'* is restricted to *'rectangle'* or *'square'*.
  Default: *10*

  'symbol_cols_max': Maximum number of module columns in the symbol.
  Value range: [*10*, *12*, *14*, ... *144*]
  Attention: Changing *'symbol_cols_max'* sets *'symbol_shape'* to *'any'*. If the size of the current symbol is only consistent with one shape type *'symbol_shape'* is restricted to *'rectangle'* or *'square'*.
  Default: *144*

  'symbol_rows_min': Minimum number of module rows in the symbol.
  Value range: [*8*, *10*, *12*, ... *144*]
  Attention: Changing *'symbol_rows_min'* sets *'symbol_shape'* to *'any'*. If the size of the current symbol is only consistent with one shape type *'symbol_shape'* is restricted to *'rectangle'* or *'square'*.
  Default: *8*

  'symbol_rows_max': Maximum number of module rows in the symbol.
  Value range: [*8*, *10*, *12*, ... *144*]
  Attention: Changing *'symbol_rows_max'* sets *'symbol_shape'* to *'any'*. If the size of the current symbol is only consistent with one shape type *'symbol_shape'* is restricted to *'rectangle'* or *'square'*.
  Default: *144*

  'symbol_cols': Set *'symbol_cols_min'* and *'symbol_cols_max'* to the given value.
  Value range: [*10*, *12*, *14*, ... *144*]
  Attention: Changing *'symbol_cols'* sets *'symbol_shape'* to *'any'*. If the size of the current symbol is only consistent with one shape type *'symbol_shape'* is restricted to *'rectangle'* or *'square'*.

  'symbol_rows': Set *'symbol_rows_min'* and *'symbol_rows_max'* to the given value.
  Value range: [*8*, *10*, *12*, ... *144*]
  Attention: Changing *'symbol_rows'* sets *'symbol_shape'* to *'any'*. If the size of the current symbol is only consistent with one shape type *'symbol_shape'* is restricted to *'rectangle'* or *'square'*.

'symbol_shape': Possible restrictions on the module shape (*'rectangle'* and/or *'square'*).

Attention: setting the symbol shape all previously made restrictions concerning the symbol size may change. For *'square'* the minimum of *'symbol_cols_min'* and *'symbol_rows_min'* and the maximum of *'symbol_cols_max'* and *'symbol_rows_max'* will be used. Additional restrictions can be seen in the following table:

symbol_shape	'any'	'rectangle'	'square'
'symbol_cols_min'	>= 10	>= 18	>= 10
'symbol_cols_max'	<= 144	<= 144	<= 144
'symbol_rows_min'	>= 8	>= 8	>= 10
'symbol_rows_max'	<= 144	<= 26	<= 144

Furthermore, if *'symbol_cols_min'* is larger than *'symbol_rows_max'*, the setting for *'symbol_shape'* is ignored and its value set to *'rectangle'*.

The same search algorithm is used for both shapes if *'finder_pattern_tolerance'* is set to *'low'*. Thus, *'symbol_shape'* has no effect for the symbol search in this case. However, if *'finder_pattern_tolerance'* is set to *'high'* or *'any'*, the value of *'symbol_shape'* may speed up the symbol search significantly if it is set to *'rectangle'* or *'square'*.

List of values: *'rectangle'*, *'square'*, *'any'*

Default: *'any'*

'symbol_size_min': Set *'symbol_cols_min'* and *'symbol_rows_min'* to the given value and *'symbol_shape'* to *'square'*.

'symbol_size_max': Set *'symbol_cols_max'* and *'symbol_rows_max'* to the given value and *'symbol_shape'* to *'square'*.

'symbol_size': Set *'symbol_cols_min'*, *'symbol_cols_max'*, *'symbol_rows_min'* and *'symbol_rows_max'* to the given value and *'symbol_shape'* to *'square'*.

- QR Code (including the finder pattern):

'model_type': Type of the QR Code model. The old QR Code Model *1* (which has been removed from ISO/IEC 18004 in 2006) and the newer Model *2* are supported. With the specification *'any'* or *'0'* both models are carried out.

List of values: *1*, *2*, *'any'*, *'0'*

Default: *2* (enhanced: *2*, maximum: *'any'*)

'version_min': Minimum symbol version. The symbol version is directly linked to the symbol size. Symbols of version 1 are 21×21 modules in size, version 2 = 25×25 modules, etc. up to version 40 = 177×177 modules. The maximum size of Model 1 symbols is 73×73 = version 14.

Value range: [*1* ... *40*] (Model 1: [*1* ... *14*])

Default: *1*

'version_max': Maximum symbol version.

Value range: [*1* ... *40*] (Model 1: [*1* ... *14*])

Default: *40*

'version': Set *'version_min'* and *'version_max'* to the same value.

'symbol_size_min': Minimum size of the symbol in modules. This parameter can be used as an alternative to *'version_min'*.

Value range: [*21* ... *177*] (Model 1: [*21* ... *73*])

Default: *21*

'symbol_size_max': Maximum size of the symbol in modules. This parameter can be used as an alternative to *'version_max'*:

Value range: [*21* ... *177*] (Model 1: [*21* ... *73*])

Default: *177*

'symbol_size': Set *'symbol_size_min'* and *'symbol_size_max'* to the same value.

- Micro QR Code:

'version_min': Minimum symbol version. The symbol version is directly linked to the symbol size. Symbols are between 11×11 (version M1) and 17×17 (version M4) modules in size.

Value range: [*1* ... *4*]

Default: *1*

'version_max': Maximum symbol version.

Value range: [*1* ... *4*]

Default: *4*

'version': Set *'version_min'* and *'version_max'* to the same value.

'symbol_size_min': Minimum size of the symbol in modules. This parameter can be used as an alternative to *'version_min'*.
Value range: [*11* ... *17*]
Default: *11*

'symbol_size_max': Maximum size of the symbol in modules. This parameter can be used as an alternative to *'version_max'*:
Value range: [*11* ... *17*]
Default: *17*

'symbol_size': Set *'symbol_size_min'* and *'symbol_size_max'* to the same value.

- PDF417:

'symbol_cols_min': Minimum number of data columns in the symbol in codewords, i.e., excluding the codewords of the start/stop pattern and of the two row indicators.
Value range: [*1* ... *30*]
Default: *1*

'symbol_cols_max': Maximum number of data columns in the symbol in codewords, i.e., excluding the two codewords of the start/stop pattern and of the two indicators.
Value range: [*1* ... *30*]
Default: *20* (enhanced: *30*)

'symbol_rows_min': Minimum number of module rows in the symbol.
Value range: [*3* ... *90*]
Default: *5* (enhanced: *3*)

'symbol_rows_max': Maximum number of module rows in the symbol.
Value range: [*3* ... *90*]
Default: *45* (enhanced: *90*)

'symbol_cols': Set *'symbol_cols_min'* and *'symbol_cols_max'* to the same value.

'symbol_rows': Set *'symbol_rows_min'* and *'symbol_rows_max'* to the same value.

- Aztec Code (including the finder pattern):

'format': Format of the Aztec Code: space separated list with the values *'compact'*, *'full_range'*, or *'rune'*.
Default: *'compact full_range'*

'symbol_size_min': Minimum size of the symbol in modules.
Value range: [*11* ... *151*]
Default: *11*

'symbol_size_max': Maximum size of the symbol in modules.
Value range: [*11* ... *151*]
Default: *151*

'symbol_size': Set *'symbol_size_min'* and *'symbol_size_max'* to the same value.

- DotCode: A valid DotCode according to AIM DotCode Revision 4.0-1 consists of one even and one odd dimension. Within HALCON, the column dimension of a DotCode symbol is defined by the reading direction and always refers to the odd dimension.

'symbol_cols_min': Minimum number of module columns in the symbol. Must be odd.
Value range: [*5* ... *999*]
Default: *5*

'symbol_cols_max': Maximum number of module columns in the symbol. Must be odd.
Value range: [*5* ... *999*]
Default: *999*

'symbol_rows_min': Minimum number of module rows in the symbol. Must be even.
Value range: [*4* ... *998*]
Default: *4*

'symbol_rows_max': Maximum number of module rows in the symbol. Must be even.
Value range: [*4* ... *998*]
Default: *998*

'symbol_cols': Set *'symbol_cols_min'* and *'symbol_cols_max'* to the given value. Must bee odd.
Value range: [*5* ... *999*]

'symbol_rows': Set *'symbol_rows_min'* and *'symbol_rows_max'* to the given value. Must be even.
Value range: [*4* ... *998*]

Appearance of the modules in the image:

- All data code types:

  'polarity': Describes the polarity of the symbol in the image, i.e., the parameter determines if the symbol appears light on a dark background or dark on a light background.
  List of values: *'dark_on_light'*, *'light_on_dark'*, *'any'*.
  Default: *'dark_on_light'* (enhanced: *'any'*)

  'mirrored': Describes whether the symbol is or may be mirrored (which is equivalent to swapping rows and columns of the symbol).
  List of values: *'no'*, *'yes'*, *'any'*
  Default: *'any'*

- All data code types except Data Matrix ECC 200 and DotCode:

  'contrast_min': Minimum contrast between the foreground and the background of the symbol (specified as gray value difference). This measure corresponds with the minimum gradient between the symbol's foreground and the background and is therefore also depending on the image sharpness.
  Value range: [*1* ... *255*]
  Default: *30* (enhanced: *10*)

- Data Matrix ECC 200, Micro QR Code, and QR Code:

  'contrast_tolerance': Describes the tolerance of the search with respect to local contrast variations (e.g., in the presence of glare or reflections). Depending on the value of the parameter two different algorithms are applied. If *'contrast_tolerance'* is set to *'high'* the robustness in the presence of strong local contrast variations is improved. In the case where *'contrast_tolerance'* is set to *'low'* the algorithm used is less robust to strong local contrast variations, however, it is faster and still able to handle contrast variations under normal circumstances and therefore *'low'* should be used in most cases. If *'contrast_tolerance'* is set to *'any'* both algorithms are applied.
  List of values: *'low'*, *'high'*, *'any'*
  Default: *'low'* (enhanced: *'low'*, maximum: *'any'*)

- All data code types except PDF417:

  'module_size_min': Minimum size of the modules in the image in pixels. Please note that for optimal reading performance a module size of at least 3-4 pixels is recommended.
  Value range: [*1* ... *100*]
  Default: *6* (enhanced: *2*, maximum: *1*)
  For DotCode:
  Default: *4* (enhanced, maximum: *2*)

  'module_size_max': Maximum size of the modules in the image in pixels.
  Value range: [*2* ... *100*]
  Default: *20* (enhanced: *100*)
  For DotCode:
  Default: *100*

  'module_size': Set *'module_size_min'* and *'module_size_max'* to the same value.

  The gap between modules can be set via the *'module_gap*'* parameters as explained in the following paragraph:
  It is possible to specify whether neighboring foreground modules are connected or whether there is or may be a gap between them. If the foreground modules are connected and fill the module space completely, the gap parameter can be set to *'no'*. The parameter is set to *'small'* if there is a very small gap between two modules, i.e.,$<$ 10% of the module size. It can be set to *'big'* if the gap is bigger (in relation to the module size: $<$ 50%). The last two settings may also be useful if the foreground modules – although being connected – appear thinner as their entitled space (e.g., as a result of blooming caused by a bright illuminant). If the foreground modules appear only as very small dots, in general, an appropriate preprocessing of the image for detecting or enlarging the modules will be necessary (e.g., by `gray_erosion_shape` or `gray_dilation_shape`).

  'module_gap_min': Minimum gap.
  List of values: *'no'*, *'small'*, *'big'* (*'big'* is not available for DotCode)
  Default: *'no'*

'module_gap_max': Maximum gap.
List of values: *'no'*, *'small'*, *'big'*
For Data Matrix ECC 200, Micro QR Code, and QR Code:
Default: *'no'* (enhanced: *'small'*, maximum: *'big'*)
For DotCode:
Default: *'no'* (enhanced: *'small'*, maximum: *'small'*)
For Aztec Code:
Default: *'small'* (enhanced: *'big'*)

'module_gap': Set *'module_gap_min'* and *'module_gap_max'* to the same value.

- All data code types except DotCode:

'small_modules_robustness': Robustness of the decoding of data codes with very small module sizes. Setting the parameter *'small_modules_robustness'* to *'high'* increases the likelihood of being able to decode data codes with very small module sizes. Additionally, in that case the minimum module size should also be adapted accordingly, thus *'module_size_min'* and *'module_width_min'* (PDF417) should be set to the expected minimum module size and width, respectively. Setting *'small_modules_robustness'* to *'high'* can significantly increase the internal memory usage of `find_data_code_2d`. Thus, in the default case *'small_modules_robustness'* should be set to *'low'*. If *'small_modules_robustness'* is set to *'high'* the maximal accepted image size is bisected (see `get_system` *'halcon_xl'*). If this image size is exceeded when calling `find_data_code_2d`, an exception is raised.
List of values: *'low'*, *'high'*
Default: *'low'* (enhanced: *'low'*, maximum: *'high'*)

- PDF417:

'module_width_min': Minimum module width in the image in pixels.
Value range: [*1* ... *100*]
Default: *3* (enhanced: *2*, maximum: *1*)

'module_width_max': Maximum module width in the image in pixels.
Value range: [*2* ... *100*]
Default: *15* (enhanced: *100*)

'module_width': Set *'module_width_min'* and *'module_width_max'* to the same value.

'module_aspect_min': Minimum module aspect ratio (module height to module width).
Value range: [*0.5* ... *20.0*]
Default: *1.0*

'module_aspect_max': Maximum module aspect ratio (module height to module width).
Value range: [*0.5* ... *20.0*]
Default: *4.0* (enhanced: *10.0*)

'module_aspect': Set *'module_aspect_min'* and *'module_aspect_max'* to the same value.

- Data Matrix ECC 200:

'slant_max': Maximum deviation of the angle of the L-shaped finder pattern from the (ideal) right angle (the angle is specified in radians and corresponds to the distortion that occurs when the symbol is printed or during the image acquisition).
Value range: [*0.0* ... *0.5235*]
Default: *0.1745* = 10° (enhanced: *0.5235* = 30°)

'finder_pattern_tolerance': Tolerance of the search with respect to a defect or partially occluded finder pattern. The finder pattern includes the L-shaped side as well as the opposite alternating side. Dependent on this parameter, different algorithms are used during the symbol search in `find_data_code_2d`. In one case (*'low'*), it is assumed that the finder pattern is present to a high degree and shows almost no disturbances. In the other case (*'high'*), the finder pattern may be defect or partially occluded without influencing the recognition and the reading of the symbol. Note, however, that in this mode the parameters for the symbol search should be restricted as narrow as possible by using `set_data_code_2d_param` because otherwise the runtime of `find_data_code_2d` may significantly increase. Also note that the two algorithms slightly differ from each other in terms of robustness. This may lead to different results depending on the value of *'finder_pattern_tolerance'* even if the finder pattern of the symbol is not disturbed. For example, if *'high'* is chosen, only symbols with an equidistant module grid can be found (see below), and hence the robustness to perspective distortions is decreased. Finally, if *'finder_pattern_tolerance'* is set to *'any'* both algorithms are applied.
List of values: *'low'*, *'high'*, *'any'*
Default: *'low'* (enhanced: *'low'*, maximum: *'any'*)

'alternating_pattern_tolerance': Tolerance of the search with respect to the variation of the module widths along the two sides with the alternating pattern. The alternating pattern analysis is decisive if *'finder_pattern_tolerance'* is set to *'low'* (or *'any'* which includes *'low'*). Note, however, that this parameter has no effect if *'finder_pattern_tolerance'* has been set to *'high'*. A total of three different values are allowed for this parameter, with a higher setting always including the lower values: *'low'* allows only a small variation of the module widths along the sides with the alternating pattern. *'medium'* first tries to meet the strict requirements of *'low'*, and if this fails, a larger variation of the module widths along the sides with the alternating pattern is allowed. Finally, *'high'* includes everything that *'medium'* allowed and additionally expands the parameter space for the symbol's outer boundaries through further attempts. It also softens the relation between the number of edges detected and the symbol type used for subsequent decoding. The greater robustness in difficult cases comes with the disadvantage that *'high'* in particular can lead to a significant runtime increase. To counteract this, the parameter space should be reduced in total, e.g., by setting the expected symbol sizes as tight as possible.
List of values: *'low'*, *'medium'*, *'high'*
Default: *'low'* (enhanced: *'medium'*, maximum: *'high'*)

'module_grid': Describes whether the size of the modules may vary (in a specific range) or not. Dependent on this parameter different algorithms are used for calculating the module's center positions. If it is set to *'fixed'*, an equidistant grid is used. Allowing a variable module size (*'variable'*), the grid is aligned only to the alternating side of the finder pattern. With *'any'* both approaches are tested one after the other. Please note that the value of *'module_grid'* is ignored if *'finder_pattern_tolerance'* is set to *'high'*. In this case an equidistant grid is assumed.
List of values: *'fixed'*, *'variable'*, *'any'*
Default: *'fixed'* (enhanced: *'any'*)

- QR Code:

  'position_pattern_min': Number of position detection patterns that have to be visible for generating a new symbol candidate.
  Value range: [*2*, *3*]
  Default: *3* (enhanced: *2*)

- Aztec Code:

  'finder_pattern_tolerance': Tolerance of the search with respect to a defect or partially occluded finder pattern. Depending on this parameter, different algorithms are used during the symbol search in `find_data_code_2d`. In one case (*'low'*), it is assumed that all rings of the finder pattern can be extracted. In the other case (*'high'*) it is assumed that at least one of the rings of the finder pattern can be extracted. As a consequence the runtime of the reader increases, if this parameter is set to *'high'*.
  List of values: *'low'*, *'high'*
  Default: *'low'* (enhanced: *'high'*)

  'additional_levels': To increase the robustness of the Aztec Code reader, a number of additional search levels (in addition to the search levels derived from the minimum and maximum module dimensions) can be specified via this parameter. Be aware that this increases the runtime of the reader, especially if no code is found.
  Value range: [*0* ... *2*]
  Default: *0*

General model behavior:

- All data code types:

  'persistence': Controls whether certain intermediate results of the symbol search with `find_data_code_2d` are stored temporarily or persistently in the model. The memory requirements of `find_data_code_2d` are significantly smaller if the data is stored temporarily (default). On the other hand, by using the persistent storage it is possible to access some of the data for debugging reasons after searching for symbols, e.g., to investigate why a symbol could not be read. The memory requirements of `find_data_code_2d` can further be reduced by setting *'persistence'* to *-1* such that only the data necessary for retrieving the decoded data is stored with the model. Be aware that print quality inspection can only be performed for Data Matrix ECC 200 symbols with this setting of *'persistence'*.
  List of values: *-1* (only decoded data), *0* (temporary), *1* (persistent)
  Default: *0*

'discard_undecoded_candidates' Controls whether candidates that could not be successfully decoded are stored in the model. Set this parameter to *'yes'* to reduce the amount of memory consumed by the model. Note that in this case information about undecoded candidates cannot be queried by using `get_data_code_2d_objects` or `get_data_code_2d_results`.
List of values: *'yes'*, *'no'*
Default: *'no'*

'strict_model': Controls the behavior of `find_data_code_2d` while detecting symbols that could be read but that do not fit the model restrictions on the size of the symbols. They can be rejected (strict model, set to *'yes'*) or returned as a result independent of their size and the size specified in the model (lax model, set to *'no'*). Please note that for DotCode symbols the module size restrictions (*'module_size_min'* and *'module_size_max'*) are not checked even if *'strict_model'* is set to *'yes'*.
List of values: *'yes'* (strict), *'no'* (not strict)
Default: *'yes'*

'string_encoding': Sets the expected string encoding of the string that is encoded in the symbol. It's possible to switch between UTF-8, Latin-1, Shift JIS and the string encoding of the current locale. If necessary, the string will be transcoded accordingly. In raw mode, the string will be passed unchanged.
List of values: *'utf8'*, *'locale'*, *'latin1'*, *'shiftjis'*, *'raw'*
Default: *'latin1'*

'timeout': By the use of this parameter, it is possible to abort `find_data_code_2d` after a defined period in milliseconds. This is especially useful in cases where a maximum cycle time has to be ensured. All results gained before the timeout can be accessed by `get_data_code_2d_results`. Passing values less or equal zero implies a deactivation of the timeout (default).
The temporal accuracy is about 10 ms. It depends on several factors including the speed of your computer, the image size and the *'timer_mode'* set via `set_system`.
`find_data_code_2d` does not raise an exception if a timeout occurs. To check whether `find_data_code_2d` has been interrupted, check the parameter *'aborted'* in `get_data_code_2d_results`.
Note that the timeout is ignored if `find_data_code_2d` runs in training mode.
Suggested values: *'false'*, *-1*, *20* ... *100*
Default: *'false'*

'abort': Using this option, it is possible to abort `find_data_code_2d` from another thread. When `set_data_code_2d_param` is called with *'abort'*, an instance of `find_data_code_2d` with the model `DataCodeHandle` running in another thread is requested to abort. If there is no `find_data_code_2d` running with this model, nothing happens.
The operator `find_data_code_2d` might not return immediately. It has to reach a cancellation point to ensure a proper cleanup. Depending on different factors like the computer performance this may take up to 10 ms.
All decoded results until this moment, are still returned. Note that the parameter is ignored if `find_data_code_2d` runs in training mode.
Note: This is the only action with a data code handle, which can be used from different threads without requiring additional synchronization.
Default: *'true'* (The value is not processed.)

- All data code types except Aztec Code:

  'strict_quiet_zone': Controls the behavior of `find_data_code_2d` while detecting symbols that could be read but show defects in their quiet zone. If *'strict_quiet_zone'* is set to *'yes'* the quiet zone of all decoded symbol is validated similar to the method used for print quality inspection. Symbols with poor grades for their quiet zone are not returned as a result. Their *'status'* is set to *'quiet zone is missing'*. If *'strict_quiet_zone'* is set to *'no'* (this is the default case), all readable symbols are returned as a result.
  List of values: *'yes'*, *'no'*
  Default: *'no'*

- All data code types except DotCode:

  'quality_isoiec15415_aperture_size': The reference image, on which the quality grades Symbol Contrast, Modulation, Reflectance Margin, and Fixed Pattern Damage are assessed, is obtained by applying a synthetic aperture (circular mean filter) on the original image. This parameter determines the filter size as the part of the module size. According to the standard, this value should be chosen between 0.5 and 0.8 depending on the application.
  For further guidance on selecting the aperture, see ISO/IEC 15415:2011 Annex D.2.

Value range: [*0* . . . *1*]
Default: *0.8*

'quality_isoiec15415_reflectance_reference': The contrast used to assess the quality grade Symbol Contrast is normalized with a reference grayvalue for reflectance. This value can be obtained with barium sulphate or magnesium oxide samples, i.a., see ISO/IEC 15415:2011 section 7.3.
Value range: [*1* . . . *255*]
Default: *255*

'quality_isoiec15415_smallest_module_size': The module size used to determine the size of the synthetic aperture. This can either be a fixed value in pixel or *'adaptive'*. In the later case, the module size obtained by the preceding `find_data_code_2d` is used. The final diameter of the filter is given by multiplying *'quality_isoiec15415_aperture_size'* with this size. See ISO/IEC 15415:2011 section 7.3.3.
Value range: >= *0*
Default: *'adaptive'*

- Data Matrix ECC 200:

'quality_isoiec15415_decode_algorithm': The computation of the quality grades is based on the module grid determined for print quality grading. Use this parameter to choose the algorithm used to obtain this grid.

'robust': Recommended for process control matters in which no calibrated camera and lighting setup based on ISO/IEC 15415:2011 is provided. The returned quality grades as well as the calculation of the grading parameters is in compliance with the international standard ISO/IEC 15415:2011. To compensate for variances caused by the flexibility in hardware setups additional information gathered during decoding of the code as well as assumptions regarding the hardware setups are used to increase the robustness of the quality inspection.

'reference': Recommended for verification of print quality. In compliance with the international standard ISO/IEC 15415:2011 the verification requires a specific camera calibration as well as lighting setup to return standard compliant grading results. In contrast to the standard, we explicitly apply a lower bound of 20% for the relative aperture factor during the computation of the constants d_min, m_min, and g_max. Additionally, if the module size (as determined by the preceding `find_data_code_2d` call) drops below 4 pixels, the algorithm cannot be used and all grades are reported as -1. The stated lower bounds help to define a reasonable application window for the reference decoding algorithm. Finally, all grades will be set to -1 if the algorithm fails during the module grid determination.

Default: *'robust'*

'decoding_scheme': Controls the decoding step of Data Matrix ECC 200. When setting this to *'raw'*, this allows to read symbols where the appearance and the error correction step conform to ISO/IEC 16022: 2006, but where the encoding is custom. The decoded data are the error corrected data, are retrieved with the *'decoded_data'* parameter of `get_data_code_2d_results` and must be further processed by the user.
List of values: *'default'*, *'raw'*
Default: *'default'*

- Data Matrix ECC 200, DotCode, Micro QR Code, and QR Code:

'candidate_selection': Controls the selection of candidate regions that are used for symbol detection. Setting this parameter to *'extensive'* increases the number of generated candidate regions and thus the likelihood of detecting a code. When this parameter is set to *'all'*, all possible candidates are used. If *'candidate_selection'* is set to *'default'*, less candidate regions are used.
List of values: *'default'*, *'extensive'*,*'all'* (*'all'* is not available for Data Matrix ECC 200)
For Data Matrix ECC 200:
Default: *'default'* (enhanced: *'extensive'*, maximum: *'extensive'*)
For DotCode, Micro QR Code, and QR Code:
Default: *'default'* (enhanced: *'extensive'*, maximum: *'all'*)

- DotCode:

'max_allowed_error_correction': Controls the maximum allowed error correction. Due to the high error correction capacity, it is possible to successfully decode false positive DotCode candidates. Especially candidates which only cover a small part of a real DotCode may be decoded successfully because of the error correction capabilities. This is the case because DotCode symbols can consist of almost every size. So there are fewer criteria to decide whether the candidate is valid or not. In order to tackle this

problem, the parameter *'max_allowed_error_correction'* can be used to specify the percentage of maximum allowed error correction. Per default the value is set to *0.9*, which means 90%. This choice helps to reduce the number of false reads without any significant impact on the number of correct decodes. Setting the value to e.g., *0.5* means, only candidates which could be decoded with maximum 50% used error correction, will be returned as successfully decoded results.
Value range: [*0.0* ... *1.0*]
Default: *'0.9'*

When setting the model parameters, **attention** should be payed especially to the following issues:

- Symbols whose size does not comply with the size restrictions made in the model (with the generic parameters *'symbol_rows*'*, *'symbol_cols*'*, *'symbol_size*'*, or *'version*'*) will not be read if *'strict_model'* is set to *'yes'*, which is the default. This behavior is useful if symbols of a specific size have to be detected while other symbols should be ignored. On the other hand, neglecting this parameter can lead to problems, e.g., if one symbol of an image sequence is used to adjust the model (including the symbol size), but later in the application the symbol size varies, which is quite common in practice.
- The runtime of `find_data_code_2d` depends mostly on the following model parameters, namely in cases where the requested number of symbols *cannot* be found in the image: *'polarity'*, *'module_size_min'* (Data Matrix ECC 200, QR Code, and Micro QR Code) and *'module_size_min'* together with *'module_aspect_min'* (PDF417), and if the minimum module size is very small also the parameters *'module_gap_*'* (Data Matrix ECC 200, Aztec Code, QR Code, and Micro QR Code), for QR Code also *'position_pattern_min'*. For Data Matrix ECC 200 symbols, if *'finder_pattern_tolerance'* is set to *'high'* or *'any'*, the symbol size should be restricted with *'symbol_size_min'*, *'symbol_size_max'*, *'symbol_size'*, and *'symbol_shape'* as narrow as possible.

Parameters

▷ **DataCodeHandle** (input_control) ..... datacode_2d ⇝ *handle*
Handle of the 2D data code model.

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Names of the generic parameters that shall be adjusted for the 2D data code.
Default: 'polarity'
List of values: `GenParamName` ∈ {'default_parameters', 'strict_model', 'persistence', 'polarity', 'mirrored', 'contrast_min', 'candidate_selection', 'discard_undecoded_candidates', 'model_type', 'version', 'version_min', 'version_max', 'string_encoding', 'symbol_size', 'symbol_size_min', 'symbol_size_max', 'symbol_cols', 'symbol_cols_min', 'symbol_cols_max', 'symbol_rows', 'symbol_rows_min', 'symbol_rows_max', 'symbol_shape', 'module_size', 'module_size_min', 'module_size_max', 'small_modules_robustness', 'module_width_min', 'module_width_max', 'module_width', 'module_aspect_min', 'module_aspect_max', 'module_aspect', 'module_gap', 'module_gap_min', 'module_gap_max', 'slant_max', 'module_grid', 'position_pattern_min', 'strict_quiet_zone', 'abort', 'trained', 'timeout', 'alternating_pattern_tolerance', 'finder_pattern_tolerance', 'format', 'additional_levels', 'contrast_tolerance', 'quality_isoiec15415_aperture_size', 'quality_isoiec15415_decode_algorithm', 'quality_isoiec15415_reflectance_reference', 'quality_isoiec15415_smallest_module_size', 'decoding_scheme', 'max_allowed_error_correction'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *string* / integer / real
Values of the generic parameters that are adjusted for the 2D data code.
Default: 'light_on_dark'
Suggested values: `GenParamValue` ∈ {'standard_recognition', 'enhanced_recognition', 'maximum_recognition', 'all', 'yes', 'no', 'any', 'dark_on_light', 'light_on_dark', 'square', 'rectangle', 'small', 'big', 'fixed', 'variable', 'low', 'high', 'default', 'extensive', 'utf8', 'locale', 'raw', 'robust', 'reference', 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 30, 50, 70, 90, 12, 14, 16, 18, 20, 22, 24, 26, 32, 36, 40, 44, 48, 52, 64, 72, 80, 88, 96, 104, 120, 132, 144}

Example

```
* This examples shows how a model can be adapted to a specific symbol if
* the symbol parameters are known

* Create a model for reading Data matrix ECC 200 codes
create_data_code_2d_model ('Data Matrix ECC 200', [], [], DataCodeHandle)
```

```
* Restrict the model by setting the module size
set_data_code_2d_param (DataCodeHandle, \
                        ['module_size_min','module_size_max'], [4,7])
* Change the polarity setting of the model from 'dark_on_light' to
* 'light_on_dark'
set_data_code_2d_param (DataCodeHandle, 'polarity', \
                        'light_on_dark')

* Read an image
read_image (Image, 'datacode/ecc200/ecc200_cpu_010')
* Read the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, [], [], \
                   ResultHandles, DecodedDataStrings)
* Clear the model
clear_data_code_2d_model (DataCodeHandle)
```

Result

The operator `set_data_code_2d_param` returns the value `2` (`H_MSG_TRUE`) if the given parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DataCodeHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_data_code_2d_model`, `read_data_code_2d_model`

Possible Successors

`get_data_code_2d_param`, `find_data_code_2d`, `write_data_code_2d_model`

Alternatives

`read_data_code_2d_model`

See also

`query_data_code_2d_params`, `get_data_code_2d_param`, `get_data_code_2d_results`, `get_data_code_2d_objects`

Module

Data Code

write_data_code_2d_model (: : DataCodeHandle, FileName :)

Writes a 2D data code model into a file.

The operator `write_data_code_2d_model` writes a 2D data code model, which was created by `create_data_code_2d_model`, into a file with the name `FileName`. This facilitates creating an identical copy of the saved model in a later session with the operator `read_data_code_2d_model`. The default HALCON file extension for the 2D data code model is 'dcm'. The handle of the model to write is passed in `DataCodeHandle`.

For an explanation of the concept of the 2D data code reader see the introduction of chapter Identification / Data Code.

Parameters

▷ **DataCodeHandle** (input_control) . . . . . datacode_2d ⇝ *handle*
Handle of the 2D data code model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the 2D data code model file.
Default: 'data_code_model.dcm'
File extension: .dcm

Example

```
* This example demonstrates how a trained model can be saved for
* a future session

* Create a model for reading Data matrix ECC 200 codes
create_data_code_2d_model ('Data Matrix ECC 200', [], [], DataCodeHandle)
* Read a training image
read_image (Image, 'datacode/ecc200/ecc200_cpu_007')
* Train the model with the symbol in the image
find_data_code_2d (Image, SymbolXLDs, DataCodeHandle, 'train', 'all', \
                   ResultHandles, DecodedDataStrings)
* Write the model into a file
write_data_code_2d_model (DataCodeHandle, 'ecc200_trained_model.dcm')
* Clear the model
clear_data_code_2d_model (DataCodeHandle)
```

Result

The operator write_data_code_2d_model returns the value 2 (H_MSG_TRUE) if the passed handle is valid and if the model can be written into the named file. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_data_code_2d_param, find_data_code_2d

Alternatives

get_data_code_2d_param

See also

create_data_code_2d_model, set_data_code_2d_param, find_data_code_2d

Module

Data Code

Chapter 15

Image

This chapter contains operators regarding the handling of images.

In order to understand the different types of images you can process in HALCON, the three components of an image (pixels, channels and domain) are explained in the following paragraphs.

To learn more about the basic concept of images in HALCON you can also see the example program `halcon_basic_concepts.hdev`.

Pixels

In HALCON pixels can be used in order to represent information of various kinds. Therefore different pixel types are distinguished. The following table lists the different pixel types and the corresponding standard image types for images. Note that this list is not exclusive (e.g., a gray value image can be of multiple other image types as well). You can convert the image type using `convert_image_type`.

Pixel Type	Standard Image Type
Gray Values	`byte, uint2`
Difference	`int1, int2`
2D Histogram	`int4`
Edge Directions	`direction`
Derivatives	`real`
Fourier Transform	`complex`
Hue Values	`cyclic`
Vector Field	`vector_field`

Note that the image type `vector_field` can be specified further by using `vector_field_absolute` or `vector_field_relative`. There is also the image type `int8` (64 bits with sign), which is only available on 64 bit systems. Further information on the different pixel types is given below.

Gray Values Gray images are of type `byte` (8 bits without sign) or `uint2` (16 bits without sign) and consist of pixels usually representing local intensities of light on a sensor.

Gray value image.

Difference In order to show the differences between two images e.g., the image types `int1` (8 bits with sign) or `int2` (16 bits with sign) are well suited.

(1) (2) (3)

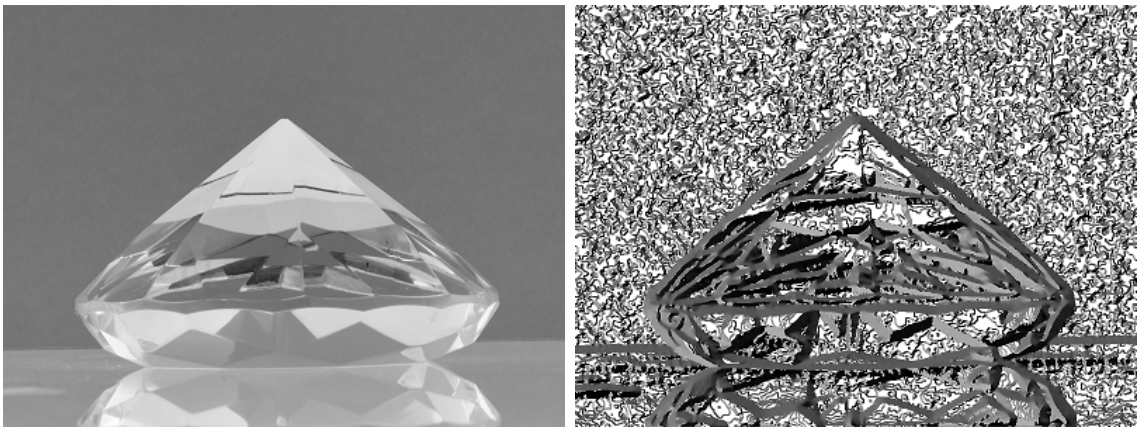
Comparing an image (1) with another (e.g., consecutively taken) one (2) by subtracting the latter from the former results in a difference image (3).

2D Histogram To examine image features based on the occurrence of gray values in two images you can use a 2D histogram, which is of type `int4` (32 bits with sign). Thereby, the axes of the 2D histogram each represent the gray values of an input image. The gray values of corresponding pixels in the input images are registered in the 2D histogram accordingly. The higher the frequency of a specific combination of gray values, the higher the gray value in the output image (see also `histo_2dim`).

(1) (2) (3)

Two channels ((1), (2)) of an exemplary image and their respective 2D histogram (3).

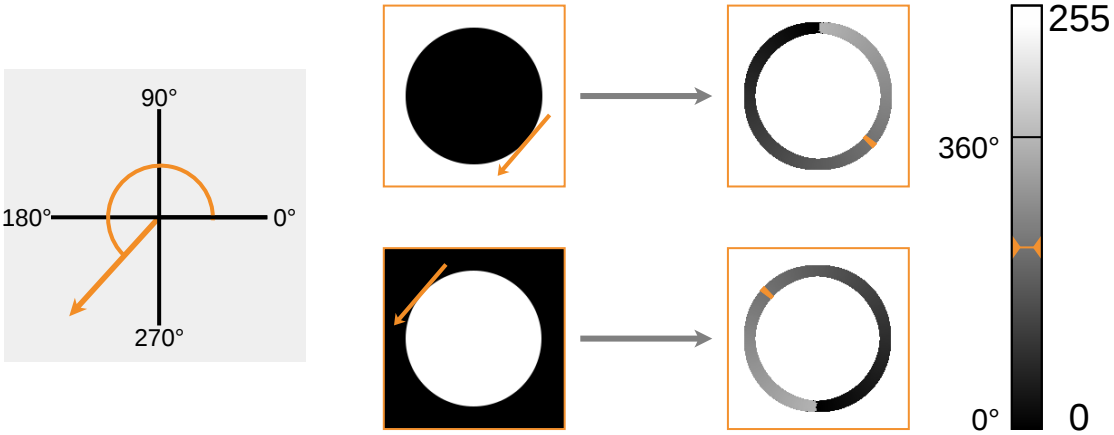
Edge Directions To represent the orientation of the edge gradient, the image type `direction` (8 bits without sign) is available.



(1) (2)

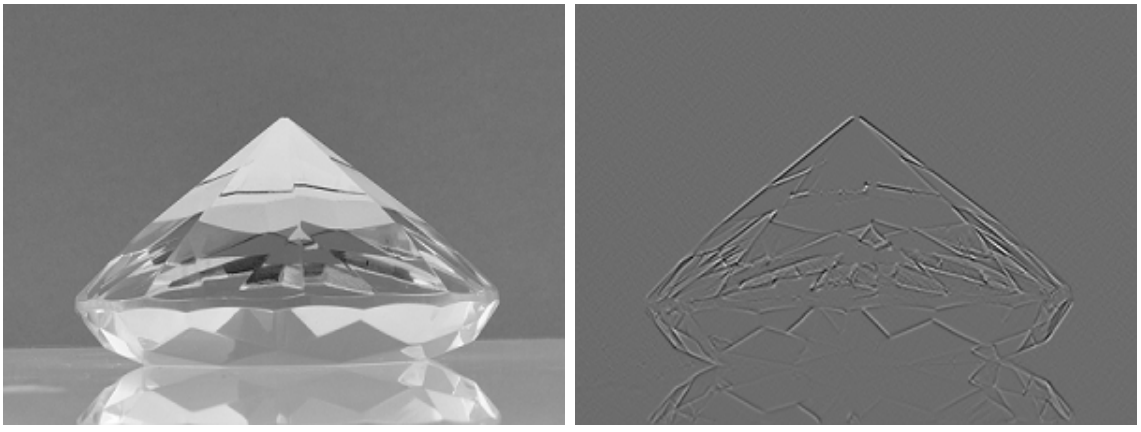
Examining an image (1) by visualizing the orientation of its image edges (2).

For images of type `direction` an edge direction of x degrees in mathematically positive sense and with respect to the horizontal axis is stored as x / 2 in the edge direction image (resulting in gray values from 0 to 179). Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction).



Visualizing the edge directions for dark and bright objects using `sobel_dir`. The edge direction (marked with an tangential arrow in the input image) of a pixel is encoded by a corresponding gray value in the result image on the right. Locations without an edge are represented by a gray value of 255.

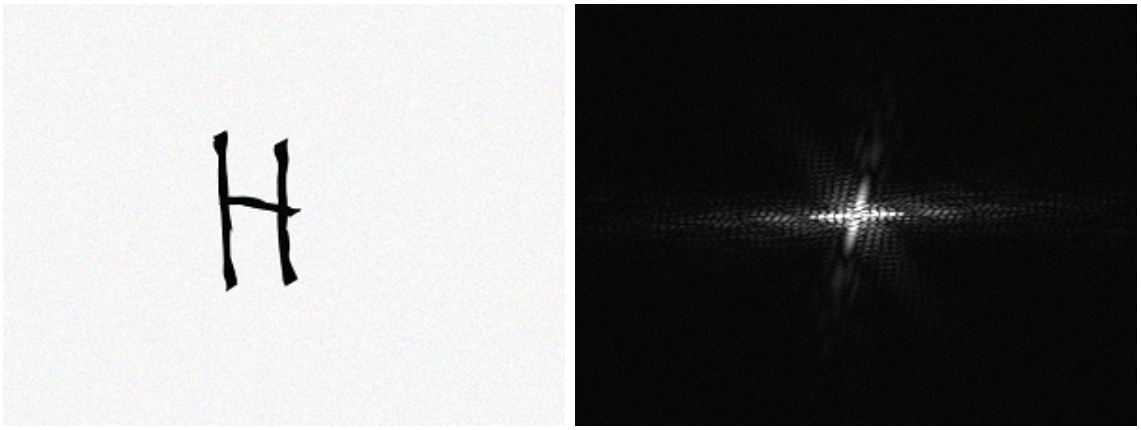
Derivatives The image type `real` (32 bits floating point value) is used to represent derivatives of an image, e.g., in order to extract edges.



(1) (2)

Some features of an image (1) (e.g., edges, gradients) can be examined by considering its derivatives (2).

Fourier Transform To inspect the frequency domain of an image the Fourier transform is used. The combination of magnitude and phase of the frequencies is represented by complex numbers, therefore the image type is `complex` (two `real` values per pixel).



(1) (2)

One strategy to identify letters (1) is to observe its frequency domain by calculating the Fourier transform (2).

Hue Values Hue values are encoded `cyclic` (8 bits without sign) so that $255 + 1 = 0$.

(1) (2)

Pixels representing hue values. When increasing a pixel value beyond 255, it is shifted to the other end of the spectrum. Therefore adding a constant number to each cyclic pixel value of (1) results in an image like (2).

Vector Field A special image type to represent absolute/relative optical flow is `vector_field` (composed by two `real` images for x and y direction). The type of vector field can be specified further by assigning `vector_field_absolute` (interpreted as absolute coordinates) or `vector_field_relative` (interpreted as vectors).

(1) (2) (3)

Relative motion of pixels (e.g., in consecutive images (1) and (2)) can be represented in a vector field (3).

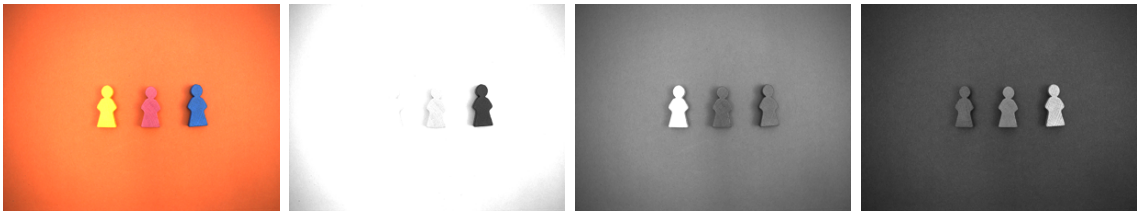
Channels

Besides different pixel types, an image can use different image channels in order to store specific information. Basically, you can use channels to assign multiple values to a pixel. The following sections mention a few cases, where different numbers of channels are useful.

Gray Value Images (Single-Channel Images) Often, there is only one value stored for every pixel of an image. A gray value image for example, stores a value representing local light intensity on a sensor for each pixel. Nevertheless, various types of information can be assigned to those pixels (e.g., the derivatives of gray values, distances). Visualizing them analogous to a gray value image can help interpreting the data.

RGB Images (Three-Channel Images) For a colored image, the pixel values are typically stored in three channels each representing the intensity of either red, green or blue light of the respective pixel (RGB image). Through additive mixing of the three values assigned to one pixel, a specific color is defined.

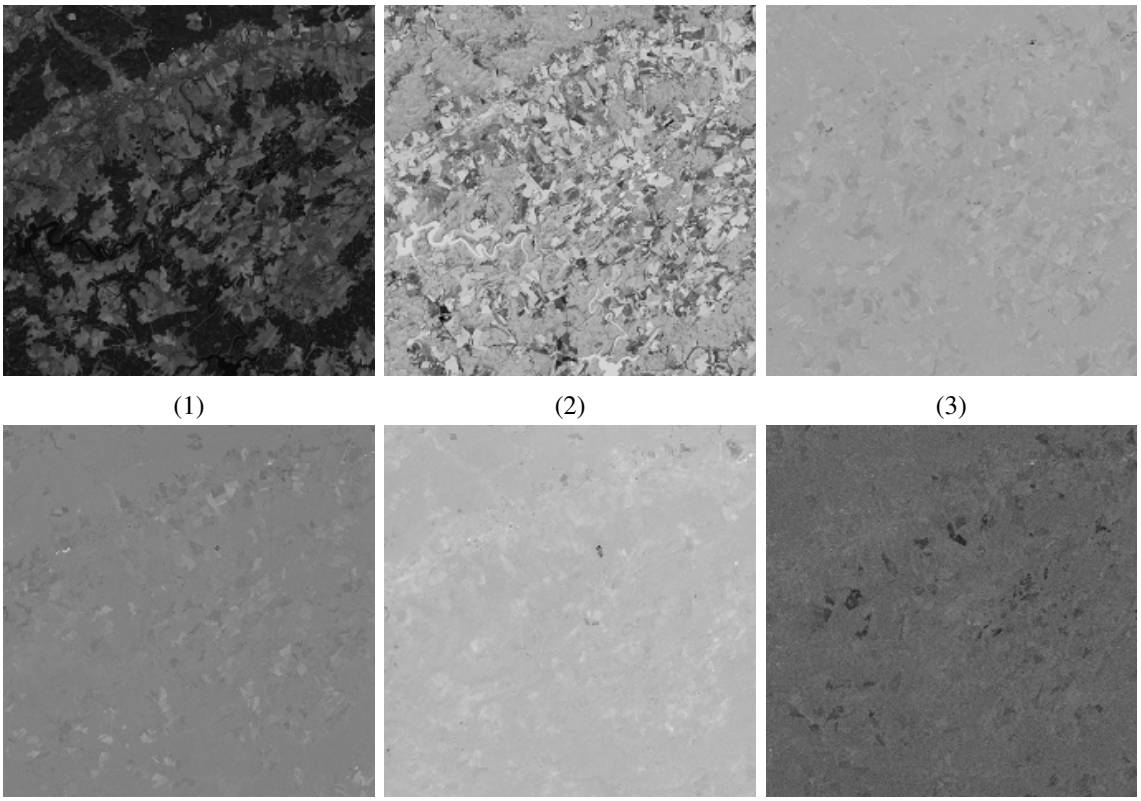
Besides the RGB model there are several further color spaces. For more information about those models and how to transform between the different color spaces see `trans_to_rgb` and `trans_from_rgb` respectively.



(1) (2) (3) (4)

RGB image (1) and its three channels separately: The local intensities differ for red (2), green (3), and blue channel (4).

Multispectral Images (Multi-Channel Images) With special cameras, a variety of spectral bands can be registered in an image, including from outside the visible light spectrum. Satellite cameras, for instance, often acquire multispectral data and store them in multiple distinct channels respectively.



(1) (2) (3)

(4) (5) (6)

Six channels of an satellite image: the different channels of a multispectral image can be used to extract features that are most prominent in a special spectral band.

Multi-Channel Images Besides representing light intensities, assembling multiple channels in one image can help with various other tasks.

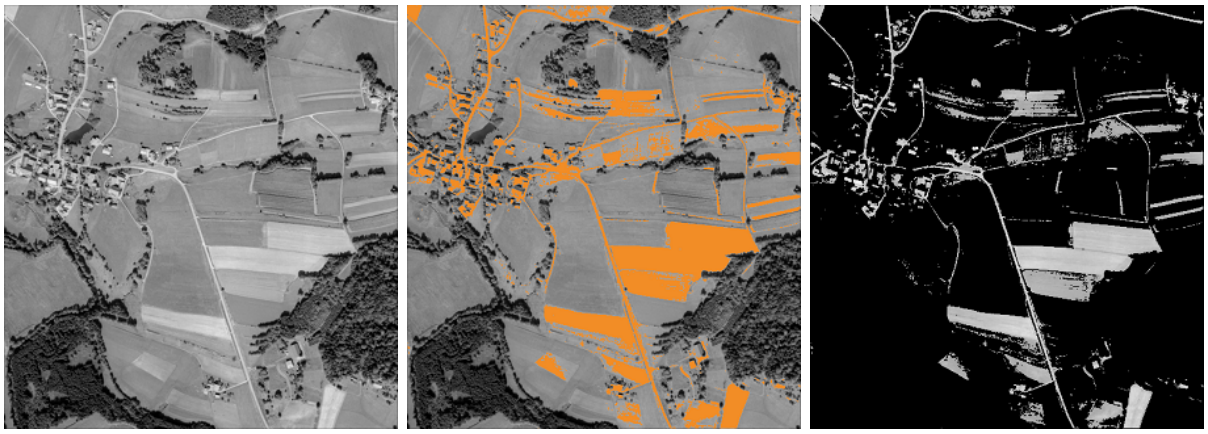
For example, combining a gray value image with an additional channel containing the respective depth values of the image pixels allows you to visualize the scene in a 3D plot.

Instead of using a depth camera you can also use a multi-channel image to extract depth information. Therefore, each channel must contain one of a series of images taken with different focus levels.

Besides containing spacial information, the channels of an image can also be interpreted as a feature space in order to perform a principal components analysis.

Domain

The image domain determines the area of an image that is used in the succeeding operations. In order to focus the processing on regions of interest and/or speed up the operations you can reduce the domain to the relevant parts of an image.



(1) (2) (3)

The domain of the original image (1) is reduced, e.g., considering a minimum gray value threshold (2). Areas that are still relevant for further operations (orange) remain in the result image (3), whereas unwanted areas of the image are excluded (black).

15.1 Access

get_grayval (Image : : Row, Column : Grayval)

Access the gray values of an image object.

The parameter Grayval is a tuple of floating point numbers or integer numbers which returns the gray values of several pixels of Image. For a multi-channel image, a group of elements, in particular one value for each channel (or two values for each channel for complex or vector field images), is returned for each pixel. The row coordinates of the pixels are specified in the tuple Row, the column coordinates are specified in the tuple Column.

Note that get_grayval does not take the domain of the image into account, i.e., if the domain has been reduced, e.g., with reduce_domain, gray values are returned even for points that lie outside the domain.

Attention

The type of the values of Grayval depends on the type of the gray values of the channels of the image Image. The operator get_grayval produces quite some overhead. Typically, it is used to get single gray values of an image (e.g., get_mposition followed by get_grayval). It is not suitable for programming image processing operations such as filters. In this case it is more useful to use the operator get_image_pointer1 and to directly use the C or C++ interface for integrating own procedures.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Image whose gray value is to be accessed.

▷ **Row** (input_control) ..... point.y(-array) ⇝ *integer*
Row coordinates of pixels to be viewed.
Default: 0
Suggested values: Row ∈ {0, 64, 128, 256, 512, 1024}
Value range: 0 ≤ Row ≤ 32767 (lin)
Minimum increment: 1
Recommended increment: 1
Restriction: 0 <= Row && Row < height(Image)

▷ **Column** (input_control) ..... point.x(-array) ⇝ *integer*
Column coordinates of pixels to be viewed.
Number of elements: Column == Row
Default: 0
Suggested values: Column ∈ {0, 64, 128, 256, 512, 1024}
Value range: 0 ≤ Column ≤ 32767 (lin)
Minimum increment: 1
Recommended increment: 1
Restriction: 0 <= Column && Column < width(Image)

▷ **Grayval** (output_control) ..... grayval-array ⇝ *real* / integer
Gray values of indicated pixels.

Result

If the state of the parameters is correct, the operator get_grayval returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image

Alternatives

get_image_pointer1, get_grayval_interpolated, get_grayval_contour_xld

See also

set_grayval

Module

Foundation

get_grayval_contour_xld (Image, Contour : : Interpolation : Grayval)

Return gray values of an image at the positions of an XLD contour.

The operator get_grayval_contour_xld returns interpolated gray values at several subpixel positions of the Image. The coordinates of the positions are specified via one XLD contour Contour. The gray values are returned in Grayval.

The interpolation method can be selected via the parameter Interpolation:

'nearest_neighbor': The results are the gray values of the nearest pixels to the selected coordinates. For images of type byte, direction, cyclic, uint2, int1, int2, int4, and int8, the parameter Grayval is a tuple of integer numbers. For images of type real and complex, the parameter Grayval is a tuple of floating point numbers.

'bilinear': The parameter Grayval is computed using a bilinear interpolation of the four neighboring gray values of the selected coordinates. The result is a tuple of floating point numbers. The runtime increases significantly compared to *'nearest_neighbor'*. Direction and cyclic images are treated like byte images.

'bicubic': The parameter Grayval is computed using a bicubic interpolation of sixteen neighboring gray values of the selected coordinates. The result is a tuple of floating point numbers. The runtime increases significantly compared to *'bilinear'*. Direction and cyclic images are treated like byte images. In this mode, the resulting gray values may contain values that lie outside of the range of numbers that can be represented by the input image type.

'bicubic_clipped': The parameter Grayval is computed using a bicubic interpolation of sixteen neighboring gray values of the selected coordinates. The result is a tuple of floating point numbers. The runtime increases significantly compared to *'bilinear'*. Direction and cyclic images are treated like byte images. In this mode, resulting gray values that lie outside of the range of numbers that can be represented by the input image type are clipped to that range.

Note that get_grayval_contour_xld does not take the domain of the image into account, i.e., if the domain has been reduced, e.g., with reduce_domain, gray values are returned even for points that lie outside the domain.

Please note also that each point of the XLD contour must be in the range $-0.5 \leq$ row coordinate $<$ height(Image)-0.5 and $-0.5 \leq$ column coordinate $<$ width(Image)-0.5.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex
Image whose gray values are to be accessed.

▷ **Contour** (input_object) ..... xld_cont ⇝ *object*
Input XLD contour with the coordinates of the positions.

▷ **Interpolation** (input_control) ..... string ⇝ *string*
Interpolation method.
Default: 'nearest_neighbor'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'bicubic_clipped'}

▷ **Grayval** (output_control) ..... grayval(-array) ⇝ *real* / integer
Gray values of the selected image coordinates.

Result

If the parameters are valid, the operator get_grayval_contour_xld returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, draw_xld, gen_contour_polygon_xld, edges_sub_pix

Alternatives

get_grayval_interpolated

Module

Foundation

get_grayval_interpolated (Image : : Row, Column, Interpolation : Grayval)

Return gray values of an image at the positions given by tuples of rows and columns.

The operator get_grayval_interpolated returns interpolated gray values at several subpixel positions of an Image. The coordinates of the row positions are given in the tuple Row, the coordinates of column positions are given in the tuple Column. The gray values are returned in Grayval as a tuple of floating point numbers.

The interpolation method can be selected via the parameter `Interpolation`. The following interpolation methods are supported:

'bilinear': The parameter `Grayval` is computed using a bilinear interpolation of the four neighboring gray values of the selected coordinates. Note: direction and cyclic images are treated like byte images.

'bicubic': The parameter `Grayval` is computed using a bicubic interpolation of sixteen neighboring gray values of the selected coordinates. Direction and cyclic images are treated like byte images. In this mode, the resulting gray values may contain values that lie outside of the range of numbers that can be represented by the input image type.

'bicubic_clipped': The parameter `Grayval` is computed using a bicubic interpolation of sixteen neighboring gray values of the selected coordinates. Direction and cyclic images are treated like byte images. In this mode, resulting gray values that lie outside of the range of numbers that can be represented by the input image type are clipped to that range.

Note that `get_grayval_interpolated` does not take the domain of the image into account, i.e., if the domain has been reduced, e.g., with `reduce_domain`, gray values are returned even for points that lie outside the domain.

Parameters

▷ **`Image`** (input_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex
Image whose gray values are to be accessed.

▷ **`Row`** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinates of positions.
Default: 0
Suggested values: `Row` ∈ {0, 64, 128, 256, 512, 1024}
Value range: -0.5 ≤ Row
Restriction: -0.5 <= Row && Row < height(Image) - 0.5

▷ **`Column`** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinates of positions.
Number of elements: Column == Row
Default: 0
Suggested values: `Column` ∈ {0, 64, 128, 256, 512, 1024}
Value range: -0.5 ≤ Column
Restriction: -0.5 <= Column && Column < width(Image) - 0.5

▷ **`Interpolation`** (input_control) ..... string ⇝ *string*
Interpolation method.
Default: 'bilinear'
List of values: `Interpolation` ∈ {'bilinear', 'bicubic', 'bicubic_clipped'}

▷ **`Grayval`** (output_control) ..... grayval(-array) ⇝ *real*
Gray values of the selected image coordinates.

Result

If the parameters are valid, the operator `get_grayval_interpolated` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_image`

Alternatives

`get_grayval_contour_xld`, `get_grayval`

Module

Foundation

get_image_pointer1 (Image : : : Pointer, Type, Width, Height)

Access the pointer of a channel.

The operator get_image_pointer1 returns a pointer to the first channel of the image Image. Additionally, the image type (Type = 'byte', 'int2', 'uint2', etc.) and the image size (width and height) are returned. Consequently, a direct access to the image data in the HALCON database via the pointer is possible from the programming language in which HALCON is used. An image is stored in HALCON linearized in row major order, i.e., line by line. Note that the image types 'complex' and 'vector_type' are handled in a specific way. In particular, 'complex' images are interleaved, i.e., the real and the imaginary parts are alternating. In contrast, 'vector_field' images consist of two matrices, one for the rows and one for the columns, which are stored in the HALCON database one after the other.

Attention

The pointer returned by get_image_pointer1 may only be used as long as the corresponding image object exists in the HALCON database. This is the case as long as the corresponding variable in the programming language in which HALCON is used is valid. If this is not observed, unexpected behavior or program crashes may result.

If data is written to an existing image via the pointer, all image objects that reference the image are modified. If, for example, the domain of an image is restricted via reduce_domain, the original image object with the full domain and the image object with the reduced domain share the same image matrix (i.e., get_image_pointer1 returns the same pointer for both images). Consequently, if one of the two images in this example is modified, both image objects are affected. Therefore, if the pointer is used to write image data in the programming language in which HALCON is used, the image data should be written into an image object that has been created solely for this purpose, e.g., using gen_image1. For multi-channel input images the type and the pointer of the first channel is returned.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Input image.

▷ **Pointer** (output_control) ..... pointer(-array) ⇝ *integer*
Pointer to the image data in the HALCON database.

▷ **Type** (output_control) ..... string(-array) ⇝ *string*
Type of image.
List of values: Type ∈ {'int1', 'int2', 'uint2', 'int4', 'int8', 'byte', 'real', 'direction', 'cyclic', 'complex', 'vector_field_absolute', 'vector_field_relative'}

▷ **Width** (output_control) ..... extent.x(-array) ⇝ *integer*
Width of image.

▷ **Height** (output_control) ..... extent.y(-array) ⇝ *integer*
Height of image.

Example

```
Hobject  Image;
char     typ[128];
Hlong     width,height;
unsigned char *ptr;

read_image(&Image,"fabrik");
get_image_pointer1(Image,(Hlong*)&ptr,typ,&width,&height);
```

Result

The operator get_image_pointer1 returns the value 2 (H_MSG_TRUE) if exactly one image was passed. The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image

Alternatives

set_grayval, get_grayval, get_image_pointer3, get_image_size, get_image_type

See also

paint_region, paint_gray

Module

Foundation

get_image_pointer1_rect (Image : : : PixelPointer, Width, Height, VerticalPitch, HorizontalBitPitch, BitsPerPixel)

Access to the image data pointer and the image data inside the smallest rectangle of the domain of the input image.

The operator get_image_pointer1_rect returns the pointer PixelPointer which points to the beginning of the image data inside the smallest rectangle of the domain of Image. VerticalPitch corresponds to the width of the input image Image multiplied with the number of bytes per pixel (HorizontalBitPitch / 8). Width and Height correspond to the size of the smallest rectangle of the input region. HorizontalBitPitch is the horizontal distance (in bits) between two neighboring pixels. BitsPerPixel is the number of used bits per pixel. get_image_pointer1_rect is symmetrical to gen_image1_rect.

Attention

The operator get_image_pointer1_rect should only be used for entry into newly created images, since otherwise the gray values of other images might be overwritten (see relational structure).

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / int4
  Input image (HImage).
- ▷ **PixelPointer** (output_control) ..... pointer ⇝ *integer*
  Pointer to the image data.
- ▷ **Width** (output_control) ..... extent.x ⇝ *integer*
  Width of the output image.
- ▷ **Height** (output_control) ..... extent.y ⇝ *integer*
  Height of the output image.
- ▷ **VerticalPitch** (output_control) ..... integer ⇝ *integer*
  Width(input image)*(HorizontalBitPitch/8).
- ▷ **HorizontalBitPitch** (output_control) ..... integer ⇝ *integer*
  Distance between two neighboring pixels in bits .
- ▷ **BitsPerPixel** (output_control) ..... integer ⇝ *integer*
  Number of used bits per pixel.

Example

```
Hobject       image,reg,imagereduced;
char          typ[128];
Hlong          width,height,vert_pitch,hori_bit_pitch,bits_per_pix, winID;
unsigned char *ptr;

open_window(0,0,512,512,"black",winID);
read_image(&image,"monkey");
draw_region(&reg,winID);
reduce_domain(image,reg,&imagereduced);
get_image_pointer1_rect(imagereduced,(Hlong*)&ptr,&width,&height,
                        &vert_pitch,&hori_bit_pitch,&bits_per_pix);
```

Result

The operator get_image_pointer1_rect returns the value 2 (H_MSG_TRUE) if exactly one image was passed. The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, gen_image1_rect

Alternatives

set_grayval, get_grayval, get_image_pointer3, get_image_pointer1

See also

paint_region, paint_gray, gen_image1_rect

Module

Foundation

get_image_pointer3 (ImageRGB : : : PointerRed, PointerGreen, PointerBlue, Type, Width, Height)

Access the pointers of a colored image.

The operator get_image_pointer3 returns a C pointer to the three channels of a colored image (ImageRGB). Additionally the image type (Type = 'byte', 'int2','float' etc.) and the image size (Width and Height) are returned. Consequently a direct access to the image data in the HALCON database from the HALCON host language via the pointer is possible. An image is stored in HALCON as a vector of image lines. The image types 'complex' and 'vector_type' are handled in a specific way. In particular, 'complex' images are interleaved, i.e., the real and the imaginary parts are alternating, whereas 'vector_field' images consist of two matrices, one for the rows and one for the columns, which are stored in the HALCON database one after the other. The three channels must have the same pixel type and the same size.

Attention

Only one image can be passed. The operator get_image_pointer3 should only be used for entry into newly created images, since otherwise the gray values of other images might be overwritten (see relational structure).

Parameters

▷ **ImageRGB** (input_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Input image.

▷ **PointerRed** (output_control) ..... pointer(-array) ⇝ *integer*
Pointer to the pixels of the first channel.

▷ **PointerGreen** (output_control) ..... pointer(-array) ⇝ *integer*
Pointer to the pixels of the second channel.

▷ **PointerBlue** (output_control) ..... pointer(-array) ⇝ *integer*
Pointer to the pixels of the third channel.

▷ **Type** (output_control) ..... string(-array) ⇝ *string*
Type of image.
List of values: Type ∈ {'int1', 'int2', 'uint2', 'int4', 'int8', 'byte', 'real', 'direction', 'cyclic', 'complex', 'vector_field_absolute', 'vector_field_relative'}

▷ **Width** (output_control) ..... extent.x(-array) ⇝ *integer*
Width of image.

▷ **Height** (output_control) ..... extent.y(-array) ⇝ *integer*
Height of image.

Result

The operator `get_image_pointer3` returns the value `2` (`H_MSG_TRUE`) if exactly one image is passed. The behavior in case of empty input (no input images available) is set via the operator `set_system ('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_image`

Alternatives

`set_grayval`, `get_grayval`, `get_image_pointer1`

See also

`paint_region`, `paint_gray`

Module

Foundation

`get_image_size` (Image : : : Width, Height)

Return the size of an image.

The operator `get_image_size` returns the image size (width and height) of the input image.

Parameters

▷ **`Image`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Input image.

▷ **`Width`** (output_control) ..... extent.x(-array) ⇝ *integer*
Width of image.

▷ **`Height`** (output_control) ..... extent.y(-array) ⇝ *integer*
Height of image.

Result

The operator `get_image_size` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_image`

Alternatives

`get_image_pointer1`

See also

`get_image_type`

Module

Foundation

get_image_time (Image : : : MSecond, Second, Minute, Hour, Day, YDay, Month, Year)

Request time at which the image was created.

The operator get_image_time returns the time at which the image was created internally in HALCON. This time doesn't necessarily correlate with the time the image was acquired.

For some image acquisition interfaces, the time of the acquisition can be queried with get_framegrabber_param.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Input image.

▷ **MSecond** (output_control) ..... integer ⇝ *integer*
Milliseconds (0..999).

▷ **Second** (output_control) ..... integer ⇝ *integer*
Seconds (0..59).

▷ **Minute** (output_control) ..... integer ⇝ *integer*
Minutes (0..59).

▷ **Hour** (output_control) ..... integer ⇝ *integer*
Hours (0..23).

▷ **Day** (output_control) ..... integer ⇝ *integer*
Day of the month (1..31).

▷ **YDay** (output_control) ..... integer ⇝ *integer*
Day of the year (1..366).

▷ **Month** (output_control) ..... integer ⇝ *integer*
Month (1..12).

▷ **Year** (output_control) ..... integer ⇝ *integer*
Year (xxxx).

Result

The operator get_image_time returns the value 2 (H_MSG_TRUE) if exactly one image was passed. The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, grab_image

See also

count_seconds

Module

Foundation

get_image_type (Image : : : Type)

Return the type of an image.

The operator get_image_type returns the image type (Type = 'byte', 'int2', 'uint2', etc.).

Attention

For multi-channel input images the type of the first channel is returned.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Input image.

▷ **Type** (output_control) ..... string(-array) ⇝ *string*
Type of image.
List of values: `Type` ∈ {'int1', 'int2', 'uint2', 'int4', 'int8', 'byte', 'real', 'direction', 'cyclic', 'complex', 'vector_field_absolute', 'vector_field_relative'}

Result

The operator `get_image_type` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_image`

Alternatives

`get_image_pointer1`

See also

`get_image_size`

Module

Foundation

15.2 Acquisition

`close_framegrabber` (: : AcqHandle :)

Close specified image acquisition device.

The operator `close_framegrabber` closes the image acquisition device specified by `AcqHandle`. In particular, allocated memory for data buffers is released and the image acquisition device is made available for other processes.

Attention

For a multithreaded application, `info_framegrabber`, `open_framegrabber`, and `close_framegrabber` are executed exclusively. Thus, they block the concurrent execution of each other, but run in parallel with all non-exclusive operators outside of this group.

Make sure that `close_framegrabber` is not called for a framegrabber handle that is being used by another thread concurrently.

Parameters

▷ **AcqHandle** (input_control) ..... framegrabber ⇝ *handle*
Handle of the image acquisition device to be closed.

Result

If the specified image acquisition device could be closed, `close_framegrabber` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- AcqHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

grab_image, grab_image_async

See also

open_framegrabber

Module

Foundation

get_framegrabber_callback (: : AcqHandle, CallbackType : CallbackFunction, UserContext)

Query callback function of an image acquisition device.

The operator get_framegrabber_callback queries a callback function for the image acquisition device specified by AcqHandle. If the callback function was registered via set_framegrabber_callback before, CallbackFunction contains a pointer to it, otherwise NULL. Furthermore, UserContext contains a pointer to the user-specific context data that was set via set_framegrabber_callback (or NULL).

With the parameter CallbackType, you can select different callback types. Suggested values are:

'exception': The image acquisition has raised an exception.

'exposure_end': The exposure of the next image has been finished.

'exposure_start': The exposure of the next image has been started.

'transfer_end': A new image is ready to be fetched by grab_image_async.

Depending on the functionality of the underlying API, additional values for CallbackType are possible. All actually supported callback types of a specific image acquisition device can be queried by calling get_framegrabber_param with the parameter 'available_callback_types'. For more details see the documentation of the specific image acquisition interface.

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively.

get_framegrabber_callback runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

▷ **AcqHandle** (input_control) ..... framegrabber ⇝ *handle*
Handle of the acquisition device to be used.

▷ **CallbackType** (input_control) ..... string ⇝ *string*
Callback type.
Default: 'transfer_end'
Suggested values: CallbackType ∈ {'exception', 'exposure_end', 'exposure_start', 'transfer_end'}

▷ **CallbackFunction** (output_control) ..... pointer ⇝ *integer*
Pointer to the callback function.

▷ **UserContext** (output_control) ..... pointer ⇝ *integer*
Pointer to user-specific context data.

Result

If the image acquisition device is open and the specified callback type is supported, the operator `get_framegrabber_callback` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_framegrabber`, `set_framegrabber_callback`

Possible Successors

`grab_image`, `grab_data`, `grab_image_start`, `grab_image_async`, `grab_data_async`, `set_framegrabber_param`, `close_framegrabber`

See also

`open_framegrabber`, `info_framegrabber`, `set_framegrabber_callback`

Module

Foundation

`get_framegrabber_lut` `( : : AcqHandle : ImageRed, ImageGreen, ImageBlue )`

Query look-up table of the image acquisition device.

The operator `get_framegrabber_lut` queries the look-up table (LUT) of the image acquisition device specified by `AcqHandle`. Note that this operation is not supported for all kinds of image acquisition devices.

Attention

For a multithreaded application, `info_framegrabber`, `open_framegrabber`, and `close_framegrabber` are executed exclusively.

`get_framegrabber_lut` runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

▷ **`AcqHandle`** (input_control) ..... framegrabber ⇝ *handle*
Handle of the acquisition device to be used.

▷ **`ImageRed`** (output_control) ..... integer-array ⇝ *integer*
Red level of the LUT entries.

▷ **`ImageGreen`** (output_control) ..... integer-array ⇝ *integer*
Green level of the LUT entries.

▷ **`ImageBlue`** (output_control) ..... integer-array ⇝ *integer*
Blue level of the LUT entries.

Result

The operator `get_framegrabber_lut` returns the value `2` (`H_MSG_TRUE`) if the image acquisition device is open.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_framegrabber`

Possible Successors

`set_framegrabber_lut`

See also

set_framegrabber_lut, open_framegrabber

Module

Foundation

get_framegrabber_param (: : AcqHandle, Param : Value)

Query specific parameters of an image acquisition device.

The operator get_framegrabber_param returns specific parameter values for the image acquisition device specified by AcqHandle. The standard parameters listed below are available for all image acquisition devices. Additional parameters may be supported by a specific image acquisition device. A list of those parameters can be obtained with the query *'parameter'* via info_framegrabber.

Standard values for Param, see open_framegrabber:

'name' Name of the image acquisition interface.

'horizontal_resolution' Horizontal resolution of the image acquisition device.

'vertical_resolution' Vertical resolution of the image acquisition device.

'image_width' Width of the specified image part.

'image_height' Height of the specified image part.

'start_row' Row coordinate of upper left corner of specified image part.

'start_column' Column coordinate of upper left corner of specified image part.

'field' Selected video field or full frame.

'bits_per_channel' Number of transferred bits per pixel and image channel.

'color_space' Color space of resulting image.

'generic' Generic value with device-specific meaning.

'external_trigger' External triggering (*'true'*/ *'false'*).

'camera_type' Type of used camera (interface-specific).

'device' Device name of the image acquisition device.

'port' Port the image acquisition device is connected to.

'line_in' Camera input line of multiplexer (optional).

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively.

get_framegrabber_param runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

▷ **AcqHandle** (input_control) . . . . . framegrabber ⇝ *handle*
Handle of the acquisition device to be used.

▷ **Param** (input_control) . . . . . string(-array) ⇝ *string*
Parameter of interest.
Default: 'revision'
Suggested values: Param ∈ {'bits_per_channel', 'camera_type', 'color_space', 'continuous_grabbing', 'device', 'external_trigger', 'field', 'generic', 'grab_timeout', 'horizontal_resolution', 'image_available', 'image_height', 'image_width', 'line_in', 'name', 'port', 'revision', 'start_column', 'start_row', 'vertical_resolution', 'volatile'}

▷ **Value** (output_control) . . . . . string(-array) ⇝ *string* / real / integer / handle
Parameter value.

Result

If the image acquisition device is open and the specified parameter is supported, the operator get_framegrabber_param returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_framegrabber, set_framegrabber_param

Possible Successors

grab_image, grab_data, grab_image_start, grab_image_async, grab_data_async, set_framegrabber_param, close_framegrabber

See also

open_framegrabber, info_framegrabber, set_framegrabber_param

Module

Foundation

grab_data (: Image, Region, Contours : AcqHandle : Data)

Synchronous grab of images and preprocessed image data from the specified image acquisition device.

The operator grab_data grabs images and preprocessed image data via the image acquisition device specified by AcqHandle. The desired operational mode of the image acquisition device as well as a suitable image part and additional interface-specific settings can be specified using the operators open_framegrabber and set_framegrabber_param. Depending on the current configuration of the image acquisition device, the preprocessed image data can be returned in terms of images (Image), regions (Region), XLD contours (Contours), and control data (Data).

To abort the grab, the operator set_framegrabber_param with the parameter 'do_abort_grab' can be used if the specific image acquisition interface supports it. Note that as an exception from the description of the concurrent usage in multiple threads (see below) 'do_abort_grab' can also be used from another thread.

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively.

grab_data runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

▷ **Image** (output_object) ..... image(-array) ⇝ *object :* byte / real / uint2
Grabbed image data.

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Preprocessed image regions.

▷ **Contours** (output_object) ..... xld_cont(-array) ⇝ *object*
Preprocessed XLD contours.

▷ **AcqHandle** (input_control) ..... framegrabber ⇝ *handle*
Handle of the acquisition device to be used.

▷ **Data** (output_control) ..... string(-array) ⇝ *string* / real / integer / handle
Preprocessed control data.

Example

```
* Select a suitable image acquisition interface name AcqName
info_framegrabber(AcqName,'port',Information,Values)
* Open image acquisition device using the default settings, see
* documentation of the actually used interface for more details
open_framegrabber(AcqName,1,1,0,0,0,0,'default',-1,'default',-1.0, \
                  'default','default','default',-1,-1,AcqHandle)
* Grab and segment image
grab_data (Image, Region, Contours, AcqHandle, Data)
* Process Region...
close_framegrabber(AcqHandle)
```

Result

If the image acquisition device is open and supports the image acquisition via grab_data, the operator grab_data returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_framegrabber, grab_image_start, set_framegrabber_param

Possible Successors

grab_data_async, grab_image_start, grab_image, grab_image_async, set_framegrabber_param, close_framegrabber

See also

open_framegrabber, info_framegrabber, set_framegrabber_param

Module

Foundation

grab_data_async (: Image, Region, Contours : AcqHandle, MaxDelay : Data)

Asynchronous grab of images and preprocessed image data from the specified image acquisition device.

The operator grab_data_async requests asynchronously grabbed images and preprocessed image data from the image acquisition device specified by AcqHandle. By default, grab_data_async also starts the next asynchronous grab before the operator returns. More information about the behavior of a specific image acquisition device can be found in the corresponding interface documentation in the directory doc/html/reference/acquisition. The desired operational mode of the image acquisition device as well as a suitable image part and additional interface-specific settings can be specified using the operators open_framegrabber and set_framegrabber_param. Depending on the current configuration of the image acquisition device, the preprocessed image data can be returned in terms of images (Image), regions (Region), XLD contours (Contours), and control data (Data).

The grab of the next image is finished by calling grab_data_async or grab_image_async. If more than MaxDelay ms have passed since the asynchronous grab was started, the asynchronously grabbed image is considered as too old and a new image is grabbed, if necessary. If a negative value is assigned to MaxDelay, this control mechanism is deactivated.

To abort the grab, the operator set_framegrabber_param with the parameter 'do_abort_grab' can be used if the specific image acquisition interface supports it. Note that as an exception from the description of the concurrent usage in multiple threads (see below) 'do_abort_grab' can also be used from another thread.

Please note that if you call the operators grab_image or grab_data after grab_data_async, the asynchronous grab started by grab_data_async is aborted and a new synchronous grab is started.

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively.

grab_data_async runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

▷ **Image** (output_object) ..... image(-array) ⇝ *object :* byte / real / uint2
Grabbed image data.

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Pre-processed image regions.

▷ **Contours** (output_object) ..... xld_cont(-array) ⇝ *object*
Pre-processed XLD contours.

▷ **AcqHandle** (input_control) . . . . . framegrabber ⇝ *handle*
Handle of the acquisition device to be used.

▷ **MaxDelay** (input_control) . . . . . number ⇝ *real*
Maximum tolerated delay between the start of the asynchronous grab and the delivery of the image [ms].
Default: -1.0
Suggested values: MaxDelay ∈ {-1.0, 20.0, 33.3, 40.0, 66.6, 80.0, 99.9}

▷ **Data** (output_control) . . . . . string(-array) ⇝ *string* / real / integer / handle
Pre-processed control data.

Example

```
* Select a suitable image acquisition interface name AcqName
open_framegrabber(AcqName,1,1,0,0,0,0,'default',-1,'default',-1.0, \
                  'default','default','default',-1,-1,AcqHandle)
* Grab image, segment it, and start next grab
grab_data_async (Image1, Region1, Contours1, AcqHandle, -1.0, Data1)
* Process data 1...
* Finish asynchronous grab, segment this image, and start next grab
grab_data_async (Image2, Region2, Contours2, AcqHandle, -1.0, Data2)
* Process data 2...
close_framegrabber(AcqHandle)
```

Result

If the image acquisition device is open and supports the image acquisition via grab_data_async, the operator grab_data_async returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_framegrabber, grab_image_start, set_framegrabber_param

Possible Successors

grab_image_async, set_framegrabber_param, close_framegrabber

See also

open_framegrabber, info_framegrabber, set_framegrabber_param

Module

Foundation

grab_image (: Image : AcqHandle :)

Synchronous grab of an image from the specified image acquisition device.

The operator grab_image grabs an image via the image acquisition device specified by AcqHandle. The desired operational mode of the image acquisition device as well as a suitable image part and additional interface-specific settings can be specified using the operators open_framegrabber and set_framegrabber_param.

To abort the grab, the operator set_framegrabber_param with the parameter 'do_abort_grab' can be used if the specific image acquisition interface supports it. Note that as an exception from the description of the concurrent usage in multiple threads (see below) 'do_abort_grab' can also be used from another thread.

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively.

grab_image runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

▷ **Image** (output_object) . . . . . image ⇝ *object :* byte / uint2
Grabbed image.

▷ **AcqHandle** (input_control) . . . . . framegrabber ⇝ *handle*
Handle of the acquisition device to be used.

Example

```
* Select a suitable image acquisition interface name AcqName
info_framegrabber(AcqName,'port',Information,Values)
* Open image acquisition device using the default settings, see
* documentation of the actually used interface for more details
open_framegrabber(AcqName,1,1,0,0,0,0,'default',-1,'default',-1.0, \
                  'default','default','default',-1,-1,AcqHandle)
grab_image(Image,AcqHandle)
close_framegrabber(AcqHandle)
```

Result

If the image could be acquired successfully, the operator `grab_image` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_framegrabber`, `set_framegrabber_param`

Possible Successors

`grab_image_start`, `grab_image_async`, `close_framegrabber`

See also

`open_framegrabber`, `info_framegrabber`, `set_framegrabber_param`

Module

Foundation

grab_image_async (: Image : AcqHandle, MaxDelay :)

Asynchronous grab of an image from the specified image acquisition device.

The operator `grab_image_async` requests an asynchronously grabbed image from the image acquisition device specified by `AcqHandle`. By default, `grab_image_async` also starts the next asynchronous grab before the operator returns. More information about the behavior of a specific image acquisition device can be found in the corresponding interface documentation in the directory `doc/html/reference/acquisition`. The desired operational mode of the image acquisition device as well as a suitable image part and additional interface-specific settings can be specified using the operators `open_framegrabber` and `set_framegrabber_param`.

The grab is finished by calling `grab_image_async` or `grab_data_async`. If more than `MaxDelay` ms have passed since the asynchronous grab was started, the asynchronously grabbed image is considered as too old and a new image is grabbed, if necessary. If a negative value is assigned to `MaxDelay`, this control mechanism is deactivated.

To abort the grab, the operator `set_framegrabber_param` with the parameter `'do_abort_grab'` can be used if the specific image acquisition interface supports it. Note that as an exception from the description of the concurrent usage in multiple threads (see below) `'do_abort_grab'` can also be used from another thread.

Please note that if you call the operators `grab_image` or `grab_data` after `grab_image_async`, the asynchronous grab started by `grab_image_async` is aborted and a new synchronous grab is started.

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively.

grab_image_async runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

▷ **Image** (output_object) . . . . . image ⇝ *object :* byte / int2
Grabbed image.

▷ **AcqHandle** (input_control) . . . . . framegrabber ⇝ *handle*
Handle of the acquisition device to be used.

▷ **MaxDelay** (input_control) . . . . . number ⇝ *real*
Maximum tolerated delay between the start of the asynchronous grab and the delivery of the image [ms].
Default: -1.0
Suggested values: MaxDelay ∈ {-1.0, 20.0, 33.3, 40.0, 66.6, 80.0, 99.9}

Example

```
* Select a suitable image acquisition interface named AcqName
open_framegrabber('AcqName', 1, 1, 0, 0, 0, 0, 'default' ,-1 , \
                  'default' ,-1.0, 'default', 'default', 'default', -1, \
                  -1, AcqHandle)
* Grab image + start next grab
grab_image_async(Image1, AcqHandle, -1.0)
* Process Image1 ...
* Finish asynchronous grab + start next grab
grab_image_async(Image2, AcqHandle, -1.0)
* Process Image2 ...
close_framegrabber(AcqHandle)
```

Result

If the image acquisition device is open and supports asynchronous grabbing the operator grab_image_async returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

grab_image_start, open_framegrabber, set_framegrabber_param

Possible Successors

grab_data_async, set_framegrabber_param, close_framegrabber

See also

grab_image_start, open_framegrabber, info_framegrabber, set_framegrabber_param

Module

Foundation

grab_image_start (: : AcqHandle, MaxDelay :)

Start an asynchronous grab from the specified image acquisition device.

The operator grab_image_start starts the asynchronous grab of an image via the image acquisition device specified by AcqHandle. The desired operational mode of the image acquisition device as well as a suitable image part and additional interface-specific settings can be specified using the operators open_framegrabber and set_framegrabber_param.

The grab is finished via grab_image_async or grab_data_async. The MaxDelay parameter is obsolete and does not effect the new asynchronous grab. Note that you can check for a too old image by using the MaxDelay parameter of the operator grab_image_async or grab_data_async, respectively.

Please note that the operator grab_image_start makes sense only when used together with grab_image_async or grab_data_async. If you call the operators grab_image or grab_data instead, the asynchronous grab started by grab_image_start is aborted and a new synchronous grab is started.

To abort the grab, the operator set_framegrabber_param with the parameter 'do_abort_grab' can be used if the specific image acquisition interface supports it. Note that as an exception from the description of the concurrent usage in multiple threads (see below) 'do_abort_grab' can also be used from another thread.

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively.

grab_image_start runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

▷ **AcqHandle** (input_control) ..... framegrabber ⇝ *handle*
Handle of the acquisition device to be used.

▷ **MaxDelay** (input_control) ..... number ⇝ *real*
This parameter is obsolete and has no effect.
Default: -1.0
Suggested values: MaxDelay ∈ {-1.0, 20.0, 33.3, 40.0, 66.6, 80.0, 99.9}

Example

```
* Select a suitable image acquisition interface named AcqName.
open_framegrabber('AcqName', 1, 1, 0, 0, 0, 0, 'default', -1, 'default', \
                  -1.0, 'default', 'default', 'default', -1, -1, AcqHandle)
* Start asynchronous grabbing.
grab_image_start(AcqHandle, -1)
* Run acquisition loop.
while (true)
    * Get image, start next grab.
    grab_image_async(Image, AcqHandle, -1.0)
    * Next: Do something with the grabbed image.
endwhile
close_framegrabber(AcqHandle)
```

Result

If the image acquisition device is open and supports asynchronous grabbing the operator grab_image_start returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_framegrabber, set_framegrabber_param

Possible Successors

grab_image_async, grab_data_async, set_framegrabber_param, close_framegrabber

See also

open_framegrabber, info_framegrabber, set_framegrabber_param

Module

Foundation

info_framegrabber (: : Name, Query : Information, ValueList)

Query information about the specified image acquisition interface.

The operator info_framegrabber returns information about the image acquisition device Name. The desired information is specified via Query. A textual description according to the selected topic is returned in Information. If applicable, ValueList contains a list of supported values. Up to now, the following queries are possible:

'bits_per_channel': List of all supported values for the parameter BitsPerChannel, see open_framegrabber.

'camera_type': Description and list of all supported values for the parameter CameraType, see open_framegrabber.

'color_space': List of all supported values for the parameter ColorSpace, see open_framegrabber.

'defaults': Interface-specific default values in ValueList, see open_framegrabber.

'device': List of all supported values for the parameter Device, see open_framegrabber.

'external_trigger': List of all supported values for the parameter ExternalTrigger, see open_framegrabber.

'field': List of all supported values for the parameter Field, see open_framegrabber.

'general': General information (in Information).

'generic': Generic value with device-specific meaning, see open_framegrabber.

'horizontal_resolution': List of all supported values for the parameter HorizontalResolution, see open_framegrabber.

'image_height': List of all supported values for the parameter ImageHeight, see open_framegrabber.

'image_width': List of all supported values for the parameter ImageWidth, see open_framegrabber.

'info_boards': Information about actually installed boards or cameras. This data is especially useful for the auto-detect mechanism of the Image Acquisition Assistant in HDevelop.

'line_in': List of all supported values for the parameter LineIn, see open_framegrabber.

'parameters': List of all interface-specific parameters which are accessible via set_framegrabber_param or get_framegrabber_param.

'parameters_readonly': List of all interface-specific parameters which are only accessible via get_framegrabber_param.

'parameters_writeonly': List of all interface-specific parameters which are only accessible via set_framegrabber_param.

'port': List of all supported values for the parameter Port, see open_framegrabber.

'revision': Version number of the image acquisition interface.

'start_column': List of all supported values for the parameter StartColumn, see open_framegrabber.

'start_row': List of all supported values for the parameter StartRow, see open_framegrabber.

'vertical_resolution': List of all supported values for the parameter VerticalResolution, see open_framegrabber.

Please check also the directory doc/html/reference/acquisition for documentation about specific image grabber interfaces.

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively. Thus, they block the concurrent execution of each other, but run in parallel with all non-exclusive operators outside of this group.

On Windows Systems, error dialog boxes from the operating system can occur when dependency modules of the interface are not found, e.g., the according SDK was not installed. The occurrence of the error boxes can be controlled by setting Windows' Error Mode. Please refer to the description of SetErrorMode within the Windows MSDN documentation.

Parameters

▷ **Name** (input_control) . . . . . string ⇝ *string*
HALCON image acquisition interface name, i.e., name of the corresponding DLL (Windows) or shared library (Linux).
Default: 'File'
Suggested values: Name ∈ {'ABS', 'ADLINK', 'AlkUSB3', 'Andor', 'BitFlow', 'Crevis', 'DahengCAM', 'DirectFile', 'DirectShow', 'Ensenso-NxLib', 'File', 'FocalSpecLCI', 'GenICamTL', 'GigEVision2', 'GingaDG', 'Ginga++', 'GStreamer', 'heliCamC3', 'KeyenceVJ', 'LinX', 'LPS36', 'LuCam', 'MatrixVisionAcquire', 'MediaFoundation', 'MILLite', 'MultiCam', 'O3D3xx', 'Opteon', 'PhoXi', 'PixeLINK', 'pylon', 'RealSense', 'SaperaLT', 'Sentech', 'SICK-3DCamera', 'SICK-ScanningRuler', 'SiliconSoftware', 'Slink', 'TWAIN', 'uEye', 'USB3Vision', 'Video4Linux2', 'VRmUsbCam'}

▷ **Query** (input_control) . . . . . string ⇝ *string*
Name of the chosen query.
Default: 'info_boards'
List of values: Query ∈ {'defaults', 'general', 'info_boards', 'parameters', 'parameters_readonly', 'parameters_writeonly', 'revision', 'bits_per_channel', 'camera_type', 'color_space', 'device', 'external_trigger', 'field', 'generic', 'horizontal_resolution', 'image_height', 'image_width', 'port', 'start_column', 'start_row', 'vertical_resolution'}

▷ **Information** (output_control) . . . . . string ⇝ *string*
Textual information (according to Query).

▷ **ValueList** (output_control) . . . . . string-array ⇝ *string* / integer / real
List of values (according to Query).

Example

```
* Select a suitable image acquisition interface name AcqName
info_framegrabber(AcqName,'port',Information,Values)
* Open image acquisition device using the default settings, see
* documentation of the actually used interface for more details
open_framegrabber(AcqName,1,1,0,0,0,0,'default',-1,'default',-1.0, \
                  'default','default','default',-1,-1,AcqHandle)
grab_image(Image,AcqHandle)
close_framegrabber(AcqHandle)
```

Result

If the parameter values are correct and the specified image acquisition interface is available, info_framegrabber returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_framegrabber

Possible Successors

open_framegrabber

See also

open_framegrabber

Module

Foundation

open_framegrabber (: : Name, HorizontalResolution, VerticalResolution, ImageWidth, ImageHeight, StartRow, StartColumn, Field, BitsPerChannel, ColorSpace, Generic, ExternalTrigger, CameraType, Device, Port, LineIn : AcqHandle)

Open and configure an image acquisition device.

The operator open_framegrabber opens and configures the chosen image acquisition device. During this process, the connection to the image acquisition device is tested, the image acquisition device is locked for other processes, and, if necessary, memory is reserved for the data buffers. The actual image grabbing is done via the operators grab_image, grab_data, grab_image_async, or grab_data_async. If the image acquisition device is not needed anymore, it should be closed via the operator close_framegrabber, releasing it for other processes. Some image acquisition devices allow to open several instances of the same image acquisition device class.

For all parameters image acquisition device-specific default values can be chosen explicitly (see the parameter description below). Additional information for a specific image acquisition device is available via info_framegrabber. A comprehensive documentation of all image acquisition device-specific parameters can be found in the corresponding interface documentation in the directory doc/html/reference/acquisition.

The meaning of the particular parameters is as follows:

HorizontalResolution, **VerticalResolution** Desired resolution of the image acquisition device.

ImageWidth, **ImageHeight** Size of the image part to be returned by grab_image etc.

StartRow, **StartColumn** Upper left corner of the desired image area.

Field Desired half image (*'first'*, *'second'*, or *'next'*) or selection of a full image.

BitsPerChannel Number of bits, which are transferred from the image acquisition device per pixel and image channel (typically *5*, *8*, *10*, *12*, or *16*).

ColorSpace Output color format of the grabbed images (typically *'gray'* or *'raw'* for single-channel or *'rgb'* or *'yuv'* for three-channel images).

Generic Generic parameter with device-specific meaning which can be queried by info_framegrabber.

ExternalTrigger Activation of external triggering (if available).

CameraType More detailed specification of the desired image acquisition device (typically the type of the analog video format or the name of the desired camera configuration file).

Device Device name of the image acquisition device.

Port Port the image acquisition device is connected to.

LineIn Camera input line of multiplexer (if available).

The operator open_framegrabber returns a handle (AcqHandle) to the opened image acquisition device.

Attention

Due to the multitude of supported image acquisition devices, open_framegrabber contains a large number of parameters. However, not all parameters are needed for a specific image acquisition device.

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively. Thus, they block the concurrent execution of each other, but run in parallel with all non-exclusive operators outside of this group.

On Windows Systems, error dialog boxes from the operating system can occur when dependency modules of the interface are not found, e.g., the according SDK was not installed. The occurrence of the error boxes can be controlled by setting Windows' Error Mode. Please refer to the description of SetErrorMode within the Windows MSDN documentation.

Parameters

▷ **Name** (input_control) ..... string ⇝ *string*
HALCON image acquisition interface name, i.e., name of the corresponding DLL (Windows) or shared library (Linux).
Default: 'File'
Suggested values: Name ∈ {'ABS', 'ADLINK', 'AlkUSB3', 'Andor', 'BitFlow', 'Crevis', 'DahengCAM', 'DirectFile', 'DirectShow', 'Ensenso-NxLib', 'File', 'FocalSpecLCI', 'GenICamTL', 'GigEVision2',

'GingaDG', 'Ginga++', 'GStreamer', 'heliCamC3', 'KeyenceVJ', 'LinX', 'LPS36', 'LuCam', 'MatrixVisionAcquire', 'MediaFoundation', 'MILLite', 'MultiCam', 'O3D3xx', 'Opteon', 'PhoXi', 'PixeLINK', 'pylon', 'RealSense', 'SaperaLT', 'Sentech', 'SICK-3DCamera', 'SICK-ScanningRuler', 'SiliconSoftware', 'Slink', 'TWAIN', 'uEye', 'USB3Vision', 'Video4Linux2', 'VRmUsbCam'}

▷ **HorizontalResolution** (input_control) . . . . . extent.x ⇝ *integer*
Desired horizontal resolution of image acquisition interface (absolute value or *1* for full resolution, *2* for half resolution, or *4* for quarter resolution).
Default: 1
Suggested values: HorizontalResolution ∈ {1, 2, 4, 1600, 1280, 768, 640, 384, 320, 192, 160, -1}

▷ **VerticalResolution** (input_control) . . . . . extent.y ⇝ *integer*
Desired vertical resolution of image acquisition interface (absolute value or *1* for full resolution, *2* for half resolution, or *4* for quarter resolution).
Default: 1
Suggested values: VerticalResolution ∈ {1, 2, 4, 1200, 1024, 576, 480, 288, 240, 144, 120, -1}

▷ **ImageWidth** (input_control) . . . . . rectangle.extent.x ⇝ *integer*
Width of desired image part (absolute value or *0* for HorizontalResolution - 2*StartColumn).
Default: 0
Suggested values: ImageWidth ∈ {0, -1}

▷ **ImageHeight** (input_control) . . . . . rectangle.extent.y ⇝ *integer*
Height of desired image part (absolute value or *0* for VerticalResolution - 2*StartRow).
Default: 0
Suggested values: ImageHeight ∈ {0, -1}

▷ **StartRow** (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Line number of upper left corner of desired image part (or border height if ImageHeight = *0*).
Default: 0
Suggested values: StartRow ∈ {0, -1}

▷ **StartColumn** (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column number of upper left corner of desired image part (or border width if ImageWidth = *0*).
Default: 0
Suggested values: StartColumn ∈ {0, -1}

▷ **Field** (input_control) . . . . . string ⇝ *string*
Desired half image or full image.
Default: 'default'
Suggested values: Field ∈ {'first', 'second', 'next', 'interlaced', 'progressive', 'default'}

▷ **BitsPerChannel** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of transferred bits per pixel and image channel (-1: device-specific default value).
Default: -1
Suggested values: BitsPerChannel ∈ {5, 8, 10, 12, 14, 16, -1}

▷ **ColorSpace** (input_control) . . . . . string(-array) ⇝ *string*
Output color format of the grabbed images, typically *'gray'* or *'raw'* for single-channel or *'rgb'* or *'yuv'* for three-channel images (*'default'*: device-specific default value).
Default: 'default'
Suggested values: ColorSpace ∈ {'gray', 'raw', 'rgb', 'yuv', 'default'}

▷ **Generic** (input_control) . . . . . string(-array) ⇝ *real* / string / integer
Generic parameter with device-specific meaning.
Default: -1

▷ **ExternalTrigger** (input_control) . . . . . string ⇝ *string*
External triggering.
Default: 'default'
List of values: ExternalTrigger ∈ {'true', 'false', 'default'}

▷ **CameraType** (input_control) . . . . . string(-array) ⇝ *string*
Type of used camera (*'default'*: device-specific default value).
Default: 'default'
Suggested values: CameraType ∈ {'ntsc', 'pal', 'auto', 'default'}

▷ **Device** (input_control) . . . . . string(-array) ⇝ *string*
Device the image acquisition device is connected to (*'default'*: device-specific default value).
Default: 'default'
Suggested values: Device ∈ {'-1', '0', '1', '2', '3', 'default'}

▷ **Port** (input_control) ..... integer(-array) ⇝ *integer*
Port the image acquisition device is connected to (*-1*: device-specific default value).
Default: -1
Suggested values: Port ∈ {0, 1, 2, 3, -1}

▷ **LineIn** (input_control) ..... integer(-array) ⇝ *integer*
Camera input line of multiplexer (*-1*: device-specific default value).
Default: -1
Suggested values: LineIn ∈ {1, 2, 3, 4, -1}

▷ **AcqHandle** (output_control) ..... framegrabber ⇝ *handle*
Handle of the opened image acquisition device.

Example

```
* Select a suitable image acquisition interface name AcqName
info_framegrabber(AcqName,'port',Information,Values)
* Open image acquisition device using the default settings, see
* documentation of the actually used interface for more details
open_framegrabber(AcqName,1,1,0,0,0,0,'default',-1,'default',-1.0, \
                  'default','default','default',-1,-1,AcqHandle)
grab_image(Image,AcqHandle)
close_framegrabber(AcqHandle)
```

Result

If the parameter values are correct and the desired image acquisition device could be opened, open_framegrabber returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

info_framegrabber

Possible Successors

grab_image, grab_data, grab_image_start, grab_image_async, grab_data_async, set_framegrabber_param, set_framegrabber_callback

See also

info_framegrabber, close_framegrabber, grab_image

Module

Foundation

set_framegrabber_callback (: : AcqHandle, CallbackType, CallbackFunction, UserContext :)

Register a callback function for an image acquisition device.

The operator set_framegrabber_callback registers a callback function for the image acquisition device specified by AcqHandle. The parameter CallbackFunction contains a pointer to the function to register, while CallbackType specifies to which function of the underlying API the callback should be connected.

Suggested values for CallbackType are:

'exception': The image acquisition has raised an exception.

'exposure_end': The exposure of the next image has been finished.

'exposure_start': The exposure of the next image has been started.

'transfer_end': A new image is ready to be fetched by grab_image_async.

Depending on the functionality of the underlying API, additional values for CallbackType are possible. All actually supported callback types of a specific image acquisition device can be queried by calling get_framegrabber_param with the parameter 'available_callback_types'. For more details see the documentation of the specific image acquisition interface.

Once the callback is registered, on every occurrence of the underlying event (e.g., the notification that the exposure has finished) the specified callback function CallbackFunction will be called. If CallbackFunction is set to NULL, the corresponding callback will be unregistered.

The signature of the callback function is the following:

```
Herror HAcqCallback(void *AcqHandle, void *Context, void *UserContext)
```

The first parameter of the callback function contains the handle to the image acquisition device passed in AcqHandle, the second one provides a pointer to interface-specific context data, and the third parameter is a user-specific pointer that is specified in UserContext.

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively.

set_framegrabber_callback runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

- ▷ **AcqHandle** (input_control) ..... framegrabber ⇝ *handle*
  Handle of the acquisition device to be used.
- ▷ **CallbackType** (input_control) ..... string ⇝ *string*
  Callback type.
  Default: 'transfer_end'
  Suggested values: CallbackType ∈ {'exception', 'exposure_end', 'exposure_start', 'transfer_end'}
- ▷ **CallbackFunction** (input_control) ..... pointer ⇝ *integer*
  Pointer to the callback function to be set.
- ▷ **UserContext** (input_control) ..... pointer ⇝ *integer*
  Pointer to user-specific context data.

Result

If the image acquisition device is open and the specified callback was registered successfully, the operator set_framegrabber_callback returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- AcqHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

open_framegrabber, set_framegrabber_param

Possible Successors

grab_image, grab_data, grab_image_start, grab_image_async, grab_data_async, get_framegrabber_param

See also

open_framegrabber, get_framegrabber_callback

Module

Foundation

set_framegrabber_lut (: : AcqHandle, ImageRed, ImageGreen, ImageBlue :)

Set look-up table of the image acquisition device.

The operator set_framegrabber_lut sets the look-up table (LUT) of the image acquisition device specified by AcqHandle. Note that this operation is not supported for all kinds of image acquisition devices.

Attention

For a multithreaded application, info_framegrabber, open_framegrabber, and close_framegrabber are executed exclusively.

set_framegrabber_lut runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

- ▷ **AcqHandle** (input_control) ..... framegrabber ⇝ *handle*
  Handle of the acquisition device to be used.
- ▷ **ImageRed** (input_control) ..... integer-array ⇝ *integer*
  Red level of the LUT entries.
- ▷ **ImageGreen** (input_control) ..... integer-array ⇝ *integer*
  Green level of the LUT entries.
- ▷ **ImageBlue** (input_control) ..... integer-array ⇝ *integer*
  Blue level of the LUT entries.

Result

The operator set_framegrabber_lut returns the value 2 (H_MSG_TRUE) if the specified LUT is correct and the image acquisition device is open.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- AcqHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

open_framegrabber, get_framegrabber_lut

Possible Successors

grab_image, grab_data, grab_image_start, grab_image_async, grab_data_async

See also

get_framegrabber_lut, open_framegrabber

Module

Foundation

set_framegrabber_param (: : AcqHandle, Param, Value :)

Set specific parameters of an image acquisition device.

The operator set_framegrabber_param sets specific parameters for the image acquisition device specified by AcqHandle. Additional information for a specific image acquisition device is available via info_framegrabber. A comprehensive documentation of all image acquisition device-specific parameters can be found in the corresponding interface documentation in the directory doc/html/reference/acquisition.

Attention

For a multithreaded application, `info_framegrabber`, `open_framegrabber`, and `close_framegrabber` are executed exclusively.

`set_framegrabber_param` runs in parallel with all non-exclusive operators inside and outside of this group.

Parameters

▷ **`AcqHandle`** (input_control) . . . . . framegrabber ⇝ *handle*
Handle of the acquisition device to be used.

▷ **`Param`** (input_control) . . . . . string(-array) ⇝ *string*
Parameter name.
Suggested values: `Param` ∈ {'color_space', 'continuous_grabbing', 'external_trigger', 'grab_timeout', 'image_height', 'image_width', 'port', 'start_column', 'start_row', 'volatile'}

▷ **`Value`** (input_control) . . . . . string(-array) ⇝ *string* / real / integer / handle
Parameter value to be set.

Result

If the image acquisition device is open and the specified parameter / parameter value is supported, the operator `set_framegrabber_param` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- AcqHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`open_framegrabber`

Possible Successors

`grab_image`, `grab_data`, `grab_image_start`, `grab_image_async`, `grab_data_async`, `get_framegrabber_param`

See also

`open_framegrabber`, `info_framegrabber`, `get_framegrabber_param`

Module

Foundation

15.3 Channel

`access_channel` (MultiChannelImage : Image : Channel :)

Access a channel of a multi-channel image.

The operator `access_channel` accesses a channel of the (multi-channel) input image. The result is a one-channel image. The definition domain of the input is adopted. The channels are numbered from 1 to n. The number of channels can be determined via the operator `count_channels`.

Parameters

▷ **`MultiChannelImage`** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

▷ **Image** (output_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

One channel of MultiChannelImage.

▷ **Channel** (input_control) ..... channel ⇝ *integer*

Index of channel to be accessed.
Default: 1
Suggested values: Channel ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
Value range: 1 ≤ Channel

Example

```
read_image(&Color,"patras");    /* read color image */
access_channel(Color,&Red,1);   /* extract red channel */
disp_image(Red,WindowHandle);
```

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

count_channels

Possible Successors

disp_image

Alternatives

decompose2, decompose3, decompose4, decompose5

See also

count_channels

Module

Foundation

append_channel (MultiChannelImage, Image : ImageExtended : :)

Append additional matrices (channels) to the image.

The operator append_channel appends the matrices of the image Image to the matrices of MultiChannelImage. The result is an image containing as many matrices (channels) as MultiChannelImage and Image combined. The definition domain of the output image is calculated as the intersection of the definition domains of both input images. MultiChannelImage may be a region only that is then interpreted as the definition domain of an image without channels. No new storage is allocated for the multi-channel image. Instead, the created multi-channel image contains references to the existing input images.

Parameters

▷ **MultiChannelImage** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Image to be appended.

▷ **ImageExtended** (output_object) ..... multichannel-image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Image appended by Image.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

disp_image

Alternatives

compose2, compose3, compose4, compose5

Module

Foundation

channels_to_image (Images : MultiChannelImage : :)

Convert one-channel images into a multi-channel image

The operator channels_to_image converts several one-channel images into a multi-channel image. The new definition domain is the intersection of the definition domains of the input images. No new storage is allocated for the multi-channel images, unless images of different dimensions are combined. In that case, new storage is allocated for all images that do not have both maximum width and height among all input images. For all images with maximum dimensions, the created multi-channel image contains references to the existing input images.

Parameters

▷ **Images** (input_object) ..... singlechannelimage-array ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

One-channel images to be combined into a one-channel image.

▷ **MultiChannelImage** (output_object) ..... multichannel-image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

count_channels, disp_image

See also

image_to_channels

Module

Foundation

compose2 (Image1, Image2 : MultiChannelImage : :)

Convert two images into a two-channel image.

The operator compose2 converts 2 one-channel images into a 2-channel image. The definition domain is calculated as the intersection of the definition domains of the input images. No new storage is allocated for the multi-channel image. Instead, the created multi-channel image contains references to the existing input images.

Parameters

▷ **Image1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 1.

▷ **Image2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 2.

▷ **MultiChannelImage** (output_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

disp_image

Alternatives

append_channel

See also

decompose2

Module

Foundation

compose3 (Image1, Image2, Image3 : MultiChannelImage : :)

Convert 3 images into a three-channel image.

The operator compose3 converts 3 one-channel images into a 3-channel image. The definition domain is calculated as the intersection of the definition domains of the input images. No new storage is allocated for the multi-channel image. Instead, the created multi-channel image contains references to the existing input images.

Parameters

▷ **Image1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 1.

▷ **Image2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 2.

▷ **Image3** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 3.

▷ **MultiChannelImage** (output_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

disp_image

Alternatives

append_channel

See also

decompose3

Module

Foundation

compose4 (Image1, Image2, Image3, Image4 : MultiChannelImage : :)

Convert 4 images into a four-channel image.

The operator compose4 converts 4 one-channel images into a 4-channel image. The definition domain is calculated as the intersection of the definition domains of the input images. No new storage is allocated for the multi-channel image. Instead, the created multi-channel image contains references to the existing input images.

Parameters

▷ **Image1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 1.

▷ **Image2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 2.

▷ **Image3** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 3.

▷ **Image4** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 4.

▷ **MultiChannelImage** (output_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

disp_image

Alternatives

append_channel

See also

decompose4

Module

Foundation

compose5 (Image1, Image2, Image3, Image4, Image5 : MultiChannelImage : :)

Convert 5 images into a five-channel image.

The operator compose5 converts 5 one-channel images into a 5-channel image. The definition domain is calculated as the intersection of the definition domains of the input images. No new storage is allocated for the multi-channel image. Instead, the created multi-channel image contains references to the existing input images.

Parameters

▷ **Image1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 1.

▷ **Image2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 2.

▷ **Image3** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 3.

▷ **Image4** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 4.

▷ **Image5** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 5.

▷ **MultiChannelImage** (output_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

disp_image

Alternatives

append_channel

See also

decompose5

Module

Foundation

compose6 (Image1, Image2, Image3, Image4, Image5, Image6 : MultiChannelImage : :)

Convert 6 images into a six-channel image.

The operator compose6 converts 6 one-channel images into a 6-channel image. The definition domain is calculated as the intersection of the definition domains of the input images. No new storage is allocated for the multi-channel image. Instead, the created multi-channel image contains references to the existing input images.

Parameters

▷ **Image1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 1.

▷ **Image2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 2.

▷ **Image3** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 3.

▷ **Image4** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 4.

▷ **Image5** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 5.

▷ **Image6** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 6.

▷ **MultiChannelImage** (output_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

disp_image

Alternatives

append_channel

See also

decompose6

Module

Foundation

compose7 (Image1, Image2, Image3, Image4, Image5, Image6, Image7 : MultiChannelImage : :)

Convert 7 images into a seven-channel image.

The operator compose7 converts 7 one-channel images into a 7-channel image. The definition domain is calculated as the intersection of the definition domains of the input images. No new storage is allocated for the multi-channel image. Instead, the created multi-channel image contains references to the existing input images.

Parameters

▷ **Image1** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 1.

▷ **Image2** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 2.

▷ **Image3** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 3.

▷ **Image4** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 4.

▷ **Image5** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 5.

▷ **Image6** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 6.

▷ **Image7** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image 7.

▷ **MultiChannelImage** (output_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

disp_image

Alternatives

append_channel

See also

decompose7

Module

Foundation

count_channels (MultiChannelImage : : : Channels)

Count channels of image.

The operator count_channels counts the number of channels of all input images.

Parameters

▷ **MultiChannelImage** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

One- or multi-channel image.

▷ **Channels** (output_control) . . . . . integer(-array) ⇝ *integer*
Number of channels.

Example

```
read_image(&Color,"patras");
count_channels(Color,&num_channels);
for (i=1; i<=num_channels; i++)
{
  access_channel(Color,&Channel,i);
  disp_image(Channel,WindowHandle);
  clear_obj(Channel);
}
```

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

access_channel, append_channel, disp_image

See also

append_channel, access_channel

Module

Foundation

decompose2 (MultiChannelImage : Image1, Image2 : :)

Convert a two-channel image into two images.

The operator decompose2 converts a 2-channel image into two one-channel images with the same definition domain. No new storage is allocated for the output images. Instead, the created images contain references to the existing input image channels.

Parameters

▷ **MultiChannelImage** (input_object) . . . . . multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Multi-channel image.

▷ **Image1** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Output image 1.

▷ **Image2** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Output image 2.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

count_channels

Possible Successors

disp_image

Alternatives

access_channel, image_to_channels

See also

compose2

Module

Foundation

decompose3 (MultiChannelImage : Image1, Image2, Image3 : :)

Convert a three-channel image into three images.

The operator decompose3 converts a 3-channel image into three one-channel images with the same definition domain. No new storage is allocated for the output images. Instead, the created images contain references to the existing input image channels.

Parameters

▷ **MultiChannelImage** (input_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

▷ **Image1** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 1.

▷ **Image2** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 2.

▷ **Image3** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 3.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

count_channels

Possible Successors

disp_image

Alternatives

access_channel, image_to_channels

See also

compose3

Module

Foundation

decompose4 (MultiChannelImage : Image1, Image2, Image3, Image4 : :)

Convert a four-channel image into four images.

The operator decompose4 converts a 4-channel image into four one-channel images with the same definition domain. No new storage is allocated for the output images. Instead, the created images contain references to the existing input image channels.

Parameters

▷ **MultiChannelImage** (input_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

▷ **Image1** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 1.

▷ **Image2** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 2.

▷ **Image3** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 3.

▷ **Image4** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 4.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

count_channels

Possible Successors

disp_image

Alternatives

access_channel, image_to_channels

See also

compose4

Module

Foundation

decompose5 (MultiChannelImage : Image1, Image2, Image3, Image4, Image5 : :)

Convert a five-channel image into five images.

The operator decompose5 converts a 5-channel image into five one-channel images with the same definition domain. No new storage is allocated for the output images. Instead, the created images contain references to the existing input image channels.

Parameters

▷ **MultiChannelImage** (input_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

▷ **Image1** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 1.

▷ **Image2** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 2.

▷ **Image3** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 3.

▷ **Image4** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 4.

▷ **Image5** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 5.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

count_channels

Possible Successors

disp_image

Alternatives

access_channel, image_to_channels

See also

compose5

Module

Foundation

decompose6 (MultiChannelImage : Image1, Image2, Image3, Image4, Image5, Image6 : :)

Convert a six-channel image into six images.

The operator `decompose6` converts a 6-channel image into six one-channel images with the same definition domain. No new storage is allocated for the output images. Instead, the created images contain references to the existing input image channels.

Parameters

▷ **MultiChannelImage** (input_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

▷ **Image1** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 1.

▷ **Image2** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 2.

▷ **Image3** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 3.

▷ **Image4** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 4.

▷ **Image5** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 5.

▷ **Image6** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 6.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`count_channels`

Possible Successors

`disp_image`

Alternatives

`access_channel`, `image_to_channels`

See also

`compose6`

Module

Foundation

decompose7 (MultiChannelImage : Image1, Image2, Image3, Image4, Image5, Image6, Image7 : :)

Convert a seven-channel image into seven images.

The operator `decompose7` converts a 7-channel image into seven one-channel images with the same definition domain. No new storage is allocated for the output images. Instead, the created images contain references to the existing input image channels.

Parameters

▷ **MultiChannelImage** (input_object) ..... multichannel-image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image.

▷ **Image1** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 1.

▷ **Image2** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 2.

▷ **Image3** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 3.

▷ **Image4** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 4.

▷ **Image5** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 5.

▷ **Image6** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 6.

▷ **Image7** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Output image 7.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`count_channels`

Possible Successors

`disp_image`

Alternatives

`access_channel`, `image_to_channels`

See also

`compose7`

Module

Foundation

image_to_channels (MultiChannelImage : Images : :)

Convert a multi-channel image into One-channel images

The operator image_to_channels generates a one-channel image for each channel of the multi-channel image in MultiChannelImage. The definition domains are adopted from the input image. As many images are created as MultiChannelImage has channels. No new storage is allocated for the output images. Instead, the created images contain references to the existing input image channels.

Parameters

▷ **MultiChannelImage** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Multi-channel image to be decomposed.

▷ **Images** (output_object) ..... singlechannelimage-array ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Generated one-channel images.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

count_channels

Possible Successors

disp_image

Alternatives

access_channel, decompose2, decompose3, decompose4, decompose5

See also

channels_to_image

Module

Foundation

15.4 Creation

copy_image (Image : DupImage : :)

Copy an image and allocate new memory for it.

copy_image copies the input image into a new image with the same domain as the input image. In contrast to HALCON operators such as copy_obj, physical copies of all channels are created. This can be used, for example, to modify the gray values of the new image (see get_image_pointer1).

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Image to be copied.

▷ **DupImage** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Copied image.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, gen_image_const

Possible Successors

set_grayval, get_image_pointer1

Alternatives

set_grayval, paint_gray, gen_image_const, gen_image_proto

See also

get_image_pointer1

Module

Foundation

gen_image1 (: Image : Type, Width, Height, PixelPointer :)

Create an image from a pointer to the pixels.

The operator gen_image1 creates an image of the size Width × Height. The pixels in PixelPointer are stored line-sequentially. The type of the given pixels (PixelPointer) must correspond to Type (see gen_image_const for a more detailed description of the pixel types). The storage for the new image is newly created by HALCON. Thus, the storage on the PixelPointer can be released after the call. Note that how to pass a pointer value depends on the used operator signature and programming environment. Make sure to pass the actual memory address where the image data is stored, not the address of a pointer variable. Care must be taken not to truncate 64-bit pointers on 64-bit architectures.

Attention

gen_image1 does not check whether the PixelPointer is valid or not. Thus, it must be ensured by the user that it is valid. Otherwise, the program may crash!

Parameters

▷ **Image** (output_object) . . . . . image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Created image with new image matrix.

▷ **Type** (input_control) . . . . . string ⇝ *string*
Pixel type.
Default: 'byte'
List of values: Type ∈ {'byte', 'direction', 'cyclic', 'int1', 'int2', 'uint2', 'int4', 'real'}

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of image.
Default: 512
Suggested values: Width ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of image.
Default: 512
Suggested values: Height ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

▷ **PixelPointer** (input_control) . . . . . pointer ⇝ *integer*
Pointer to first gray value.

Example

```
void NewImage(Hobject *new)
{
  unsigned char  image[768*525];
  int            r,c;
  for (r=0; r<525; r++)
    for (c=0; c<768; c++)
      image[r*768+c] = c % 255;
  gen_image1(new,"byte",768,525,(Hlong)image);
}
```

Result

If the parameter values are correct, the operator gen_image1 returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_image_const, get_image_pointer1

Alternatives

gen_image3, gen_image1_extern, gen_image_const, get_image_pointer1

See also

reduce_domain, paint_gray, paint_region, set_grayval

Module

Foundation

gen_image1_extern (: Image : Type, Width, Height, PixelPointer, ClearProc :)

Create an image from a pointer on the pixels with storage management.

The operator gen_image1_extern creates an image of the size Width × Height. The pixels in PixelPointer are stored line-sequentially. The type of the given pixels (PixelPointer) must correspond to Type (see gen_image_const for a more detailed description of the image types). Note that how to pass a pointer value depends on the used operator signature and programming environment. Make sure to pass the actual memory address where the image data is stored, not the address of a pointer variable. Care must be taken not to truncate 64-bit pointers on 64-bit architectures.

The memory for the new image is not newly allocated by HALCON, contrary to gen_image1, and thus is not copied either. This means that the memory space that PixelPointer points to must be released by deleting the object Image. This is done by the procedure ClearProc provided by the caller. This procedure must have the following signature

```
void ClearProc(void* ptr);
```

and will be called using __cdecl calling convention when deleting Image. If the memory shall not be released (in the case of frame grabbers or static memory) a procedure “without trunk” or the NULL-Pointer can be passed. Analogous to the parameter PixelPointer the pointer has to be passed to the procedure depending on the used operator signature and programming environment.

Attention

gen_image1_extern does not check if enough memory for an image of Width × Height is allocated in PixelPointer.

Also, gen_image1_extern does not check whether the PixelPointer is valid or not. Thus, it must be ensured by the user that it is valid. Otherwise, the program may crash!

Parameters

▷ **Image** (output_object) . . . . . image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Created HALCON image.

▷ **Type** (input_control) . . . . . string ⇝ *string*
Pixel type.
Default: 'byte'
List of values: `Type` ∈ {'int1', 'int2', 'uint2', 'int4', 'byte', 'real', 'direction', 'cyclic'}

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of image.
Default: 512
Suggested values: `Width` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of image.
Default: 512
Suggested values: `Height` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

▷ **PixelPointer** (input_control) . . . . . pointer ⇝ *integer*
Pointer to the first gray value.

▷ **ClearProc** (input_control) . . . . . pointer ⇝ *integer*
Pointer to the procedure re-releasing the memory of the image when deleting the object.
Default: 0

Example

```
void NewImage(Hobject *new)
{
  unsigned char  *image;
  int            r,c;
  image = malloc(640*480);
  for (r=0; r<480; r++)
    for (c=0; c<640; c++)
      image[r*640+c] = c % 255;
  gen_image1_extern(new,"byte",640,480,(Hlong)image,(Hlong)free);
}
```

Result

The operator `gen_image1_extern` returns the value `2` (`H_MSG_TRUE`) if the parameter values are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`gen_image1`, `gen_image_const`, `get_image_pointer1`, `gen_image3_extern`

See also

`reduce_domain`, `paint_gray`, `paint_region`, `set_grayval`

Module

Foundation

gen_image1_rect (: Image : PixelPointer, Width, Height, VerticalPitch, HorizontalBitPitch, BitsPerPixel, DoCopy, ClearProc :)

Create an image with a rectangular domain from a pointer on the pixels (with storage management).

The operator gen_image1_rect creates an image of size (VerticalPitch / (HorizontalBitPitch/8)) * Height. The pixels pointed to by PixelPointer are stored line by line. Note that how to pass a pointer value depends on the used operator signature and programming environment. Make sure to pass the actual memory address where the image data is stored, not the address of a pointer variable. Care must be taken not to truncate 64-bit pointers on 64-bit architectures.

VerticalPitch determines the distance (in bytes) between pixel m in row n and pixel m in row n+1 inside of memory. All rows of the 'input image' have the same vertical pitch. The width of the output image is VerticalPitch / (HorizontalBitPitch / 8), its height is Height. The domain of the output image Image is a rectangle of the size Width * Height. The parameter HorizontalBitPitch is the horizontal distance (in bits) between two neighboring pixels. BitsPerPixel is the number of used bits per pixel.

If DoCopy is set *'true'*, the image data pointed to by PixelPointer is copied and memory for the new image is newly allocated by HALCON. Else the image data is not duplicated and the memory space that PixelPointer points to must be released when deleting the object Image. This is done by the procedure ClearProc provided by the caller. This procedure must have the following signature

```
void ClearProc(void* ptr);
```

and will be called using __cdecl calling convention when deleting Image. If the memory shall not be released (in the case of frame grabbers or static memory) a procedure "without trunk" or the NULL-pointer can be passed. Analogously to the parameter PixelPointer the pointer has to be passed to the procedure depending on the used operator signature and programming environment. If DoCopy is *'true'* then ClearProc is irrelevant. The operator gen_image1_rect is symmetrical to get_image_pointer1_rect.

Attention

gen_image1_rect does not check whether the PixelPointer is valid or not. Thus, it must be ensured by the user that it is valid. Otherwise, the program may crash!

Parameters

▷ **Image** (output_object) ..... image ⇝ *object :* byte / uint2 / int4
Created HALCON image.

▷ **PixelPointer** (input_control) ..... pointer ⇝ *integer*
Pointer to the first pixel.

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the image.
Default: 512
Suggested values: Width ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the image.
Default: 512
Suggested values: Height ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

▷ **VerticalPitch** (input_control) ..... integer ⇝ *integer*
Distance (in bytes) between pixel m in row n and pixel m in row n+1 of the 'input image'.
Restriction: VerticalPitch >= Width * HorizontalBitPitch / 8

▷ **HorizontalBitPitch** (input_control) ..... integer ⇝ *integer*
Distance between two neighboring pixels in bits .
Default: 8
List of values: HorizontalBitPitch ∈ {8, 16, 32}

▷ **BitsPerPixel** (input_control) . . . . . integer ⇝ *integer*
Number of used bits per pixel.
Default: 8
List of values: BitsPerPixel ∈ {8, 9, 10, 11, 12, 13, 14, 15, 16, 32}
Restriction: BitsPerPixel <= HorizontalBitPitch

▷ **DoCopy** (input_control) . . . . . string ⇝ *string*
Copy image data.
Default: 'false'
Suggested values: DoCopy ∈ {'true', 'false'}

▷ **ClearProc** (input_control) . . . . . pointer ⇝ *integer*
Pointer to the procedure releasing the memory of the image when deleting the object.
Default: 0

Example

```
void NewImage(Hobject *new)
{
  unsigned char       *image;
  int                 r,c;

  image = malloc(640*480);
  for (r=0; r<480; r++)
    for (c=0; c<640; c++)
      image[r*640+c] = c % 255;
  gen_image1_rect(new,(Hlong)image,400,480,640,8,8,'false',(long)free);
}
```

Result

The operator gen_image1_rect returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_image_pointer1_rect

Alternatives

gen_image1, gen_image1_extern, gen_image_const

See also

get_image_pointer1_rect

Module

Foundation

gen_image3 (: ImageRGB : Type, Width, Height, PixelPointerRed, PixelPointerGreen, PixelPointerBlue :)

Create an image from three pointers to the pixels (red/green/blue).

The operator gen_image3 creates a three-channel image of the size Width × Height. The pixels in PixelPointerRed, PixelPointerGreen and PixelPointerBlue are stored line-sequentially. The type of the given pixels (PixelPointerRed etc.) must correspond to Type (see gen_image_const for a more detailed description of the pixel types). The storage for the new image is newly created by HALCON. Thus, it can be released after the call. Note that how to pass a pointer value depends on the used operator signature and

programming environment. Make sure to pass the actual memory address where the image data is stored, not the address of a pointer variable. Care must be taken not to truncate 64-bit pointers on 64-bit architectures.

Attention

`gen_image3` does not check whether the pixels in `PixelPointerRed`,`PixelPointerGreen`, and `PixelPointerBlue` are valid or not. Thus, it must be ensured by the user that they are valid. Otherwise, the program may crash!

Parameters

▷ **`ImageRGB`** (output_object) . . . . . image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Created image with new image matrix.

▷ **`Type`** (input_control) . . . . . string ⇝ *string*
Pixel type.
Default: 'byte'
List of values: `Type` ∈ {'byte', 'direction', 'cyclic', 'int1', 'int2', 'uint2', 'int4', 'real'}

▷ **`Width`** (input_control) . . . . . extent.x ⇝ *integer*
Width of image.
Default: 512
Suggested values: `Width` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **`Height`** (input_control) . . . . . extent.y ⇝ *integer*
Height of image.
Default: 512
Suggested values: `Height` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

▷ **`PixelPointerRed`** (input_control) . . . . . pointer ⇝ *integer*
Pointer to first red value (channel 1).

▷ **`PixelPointerGreen`** (input_control) . . . . . pointer ⇝ *integer*
Pointer to first green value (channel 2).

▷ **`PixelPointerBlue`** (input_control) . . . . . pointer ⇝ *integer*
Pointer to first blue value (channel 3).

Example

```
void NewRGBImage(Hobject *new)
{
  unsigned char  red[768*525];
  unsigned char  green[768*525];
  unsigned char  blue[768*525];
  int            r,c;
  for (r=0; r<525; r++)
    for (c=0; c<768; c++)
    {
      red[r*768+c]   = c % 255;
      green[r*768+c] = (767 - c) % 255;
      blue[r*768+c]  = r % 255;
    }
    gen_image3(new,"byte",768,525,(Hlong)red,(long)green,(long)blue);
}

main()
{
  Hobject  rgb;
  open_window(0,0,768,525,0,"","",&WindowHandle);
  NewRGBImage(&rgb);
```

```
  disp_color(rgb,WindowHandle);
}
```

Result

If the parameter values are correct, the operator gen_image3 returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_image_const, get_image_pointer1

Possible Successors

disp_color

Alternatives

gen_image1, compose3, gen_image_const

See also

reduce_domain, paint_gray, paint_region, set_grayval, get_image_pointer1, decompose3

Module

Foundation

gen_image3_extern (: Image : Type, Width, Height, PointerRed, PointerGreen, PointerBlue, ClearProc :)

Create a three-channel image from three pointers on the pixels with storage management.

The operator gen_image3_extern creates a three-channel image of the size Width × Height. The pixels in PointerRed, PointerGreen, and PointerBlue are stored line-sequentially. The type of the given pixels must correspond to Type (see gen_image_const for a more detailed description of the image types). Note that how to pass a pointer value depends on the used operator signature and programming environment. Make sure to pass the actual memory address where the image data is stored, not the address of a pointer variable. Care must be taken not to truncate 64-bit pointers on 64-bit architectures.

The memory for the new image is not newly allocated by HALCON, contrary to gen_image3, and thus is not copied either. This means that the memory space that PointerRed, PointerGreen, and PointerBlue point to must be released by deleting the object Image. This is done by the procedure ClearProc provided by the caller. This procedure must have the following signature

```
void ClearProc(void* ptr);
```

and will be called using __cdecl calling convention when deleting Image. If the memory shall not be released (in the case of frame grabbers or static memory) a procedure “without trunk” or the NULL-Pointer can be passed. Analogous to the parameters PointerRed, PointerGreen, and PointerBlue the pointer has to be passed to the procedure depending on the used operator signature and programming environment.

Attention

gen_image3_extern does not check if enough memory for an image of Width × Height is allocated in PointerRed, PointerGreen, and PointerBlue.

Also, gen_image3_extern does not check whether the pixels in PointerRed, PointerGreen, and PointerBlue are valid or not. Thus, it must be ensured by the user that they are valid. Otherwise, the program may crash!

Parameters

▷ **Image** (output_object) ..... image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Created HALCON image.

▷ **Type** (input_control) ..... string ⇝ *string*
Pixel type.
Default: 'byte'
List of values: `Type` ∈ {'int1', 'int2', 'uint2', 'int4', 'byte', 'real', 'direction', 'cyclic'}

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of image.
Default: 512
Suggested values: `Width` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of image.
Default: 512
Suggested values: `Height` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

▷ **PointerRed** (input_control) ..... pointer ⇝ *integer*
Pointer to the first gray value of the first channel.

▷ **PointerGreen** (input_control) ..... pointer ⇝ *integer*
Pointer to the first gray value of the second channel.

▷ **PointerBlue** (input_control) ..... pointer ⇝ *integer*
Pointer to the first gray value of the third channel.

▷ **ClearProc** (input_control) ..... pointer ⇝ *integer*
Pointer to the procedure re-releasing the memory of the image when deleting the object.
Default: 0

Example

```
void NewImage(Hobject *new)
{
  unsigned char  *image_red;
  unsigned char  *image_green;
  unsigned char  *image_blue;
  int            r,c;
  image_red = malloc(640*480);
  image_green = malloc(640*480);
  image_blue = malloc(640*480);
  for (r=0; r<480; r++)
    for (c=0; c<640; c++)
    {
      image_red[r*640+c] = c % 255;
      image_green[r*640+c] = (c+64) % 255;
      image_blue[r*640+c] = (c+128) % 255;
    }
  gen_image3_extern(new,"byte",640,480,(Hlong)image_red,\
(Hlong)image_green,(Hlong)image_blue,(Hlong)free);
}
```

Result

The operator `gen_image3_extern` returns the value `2` (`H_MSG_TRUE`) if the parameter values are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

gen_image3, gen_image_const, get_image_pointer3, gen_image1_extern

See also

reduce_domain, paint_gray, paint_region, set_grayval

Module

Foundation

gen_image_const (: Image : Type, Width, Height :)

Create an image with constant gray value.

The operator gen_image_const creates an image of the indicated size. The width and height of the image are determined by Width and Height. HALCON supports the following image types:

'byte' 1 byte per pixel, unsigned
Value range: (0..255)

'int1' 1 byte per pixel, signed
Value range: (-128..127)

'uint2' 2 bytes per pixel, unsigned
Value range: (0..65535)

'int2' 2 bytes per pixel, signed
Value range: (-32768..32767)

'int4' 4 bytes per pixel, signed
Value range: (-2147483648..2147483647)

'int8' 8 bytes per pixel, signed (only available on 64 bit systems)
Value range: (-9223372036854775808..9223372036854775807)

'real' 4 bytes per pixel, floating point
Precision: 6 significant decimal digits
Value range: (-3.4e38..3.4e38)

'complex' Two matrices of type *'real'*

'vector_field_relative' Two matrices of type *'real'*
Interpretation: Vectors

'vector_field_absolute' Two matrices of type *'real'*
Interpretation: Absolute coordinates

'direction' 1 byte per pixel, unsigned
Interpretation: Angle divided by two
Value range: (0..179)
Attention: The values 180..254 are automatically set to the value 255, which is interpreted as undefined angle.

'cyclic' 1 byte per pixel, unsigned, cyclic arithmetics
Value range: (0..255)

With the operator set_system('init_new_image',<'true'/'false'>), it can be controlled whether the created image is initialized with 0 or not.

Parameters

▷ **Image** (output_object) ..... image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Created image with new image matrix.

▷ **Type** (input_control) ..... string ⇝ *string*
Pixel type.
Default: 'byte'
List of values: `Type` ∈ {'byte', 'direction', 'cyclic', 'int1', 'int2', 'uint2', 'int4', 'int8', 'real', 'complex', 'vector_field_absolute', 'vector_field_relative'}

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of image.
Default: 512
Suggested values: `Width` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of image.
Default: 512
Suggested values: `Height` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

Example

```
gen_image_const(&New,"byte",width,height);
get_image_pointer1(New,(Hlong*)&pointer,type,&width,&height);
for (row=0; row<height-1; row++)
{
  for (col=0; col<width-1; col++)
  {
    pointer[row*width+col] = (row + col) % 256;
  }
}
```

Result

If the parameter values are correct, the operator `gen_image_const` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`paint_region`, `reduce_domain`, `get_image_pointer1`, `copy_obj`

Alternatives

`gen_image1`, `gen_image3`

See also

`reduce_domain`, `paint_gray`, `paint_region`, `set_grayval`, `get_image_pointer1`

Module

Foundation

gen_image_gray_ramp (: ImageGrayRamp : Alpha, Beta, Mean, Row, Column, Width, Height :)

Create a gray value ramp.

The operator gen_image_gray_ramp creates a gray value ramp according to the following equation:

$$\texttt{ImageGrayRamp}(r, c) = \texttt{Alpha}(r - \texttt{Row}) + \texttt{Beta}(c - \texttt{Column}) + \texttt{Mean}$$

The size of the image is determined by Width and Height The gray values are of the type byte. Gray values outside the valid area are clipped.

Parameters

▷ **ImageGrayRamp** (output_object) ..... image ⇝ *object :* byte
Created image with new image matrix.

▷ **Alpha** (input_control) ..... real ⇝ *real*
Gradient in line direction.
Default: 1.0
Suggested values: Alpha ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Beta** (input_control) ..... real ⇝ *real*
Gradient in column direction.
Default: 1.0
Suggested values: Beta ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Mean** (input_control) ..... real ⇝ *real*
Mean gray value.
Default: 128
Suggested values: Mean ∈ {0.0, 20.0, 40.0, 60.0, 80.0, 100.0, 120.0, 140.0, 160.0, 180.0, 200.0, 220.0, 255.0}
Minimum increment: 1
Recommended increment: 10

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Line index of reference point.
Default: 256
Suggested values: Row ∈ {128, 256, 512, 1024}
Minimum increment: 1
Recommended increment: 10

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column index of reference point.
Default: 256
Suggested values: Column ∈ {128, 256, 512, 1024}
Minimum increment: 1
Recommended increment: 10

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of image.
Default: 512
Suggested values: Width ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of image.
Default: 512
Suggested values: Height ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

Result

If the parameter values are correct `gen_image_gray_ramp` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`moments_gray_plane`

Possible Successors

`paint_region`, `reduce_domain`, `get_image_pointer1`, `copy_obj`

Alternatives

`gen_image1`

See also

`reduce_domain`, `paint_gray`, `gen_image_const`

Module

Foundation

gen_image_interleaved (: ImageRGB : PixelPointer, ColorFormat, OriginalWidth, OriginalHeight, Alignment, Type, ImageWidth, ImageHeight, StartRow, StartColumn, BitsPerChannel, BitShift :)

Create a three-channel image from a pointer to the interleaved pixels.

The operator `gen_image_interleaved` creates a three-channel image from an input image, whose pixels are stored line-sequentially in `PixelPointer`. The size of the input image has to be passed in `OriginalWidth` and `OriginalHeight`, the format of the interleaved pixels in `ColorFormat`.

The output image will be sized `ImageWidth` × `ImageHeight`. Together with the coordinates of upper left corner `StartRow` and `StartColumn` any section of the input image can be extracted. When a *0* is passed to `ImageWidth`, `ImageHeight`, `StartRow`, and `StartColumn`, the output image will have the same dimensions as the input image.

Note that the image type `Type` (see `gen_image_const`) of the output image `ImageRGB` has to be chosen such that the whole range of possible color values of the input image can be represented. I.e. `gen_image_interleaved` does not allow to create a byte image from an input image with `ColorFormat` *'rgb48'*.

When the formats *'rgb48'*, *'bgr48'*, *'rgbx64'*, and *'bgr64'* do not use the full range of 16 bits per channel and pixel, the number of actually used bits should be passed in `BitsPerChannel`. Furthermore, the pixel values of the input image can be shifted by `BitShift` bits to the right.

The fourth channel of four-channel input images in the formats *'rgbx'*, *'bgrx'*, *'rgbx64'*, and *'bgrx64'* is simply discarded.

The storage for the new image is newly created by HALCON. Thus, it can be released after the call. Note that how to pass a pointer value depends on the used operator signature and programming environment. Make sure to pass the actual memory address where the image data is stored, not the address of a pointer variable. Care must be taken not to truncate 64-bit pointers on 64-bit architectures.

Possible values for `ColorFormat`:

'rgb555': 16 bit rgb triple (5 bit per pixel and channel), the padding bit (X) being the first bit. The bit pattern is XRRRRRGG|GGGBBBBB.

'bgr555': 16 bit bgr triple (5 bit per pixel and channel), the padding bit (X) being the first bit. The bit pattern is XBBBBBGG|GGGRRRRR.

'rgb5551': 16 bit rgb triple (5 bit per pixel and channel), the padding bit (X) being the last bit. The bit pattern is RRRRRGGG|GGBBBBBX.

'bgr5551': 16 bit bgr triple (5 bit per pixel and channel), the padding bit (X) being the last bit. The bit pattern is BBBBBGGG|GGRRRRRX.

'rgb565': 16 bit rgb triple (5 bit per pixel and channel, 6 bit for the green channel). The bit pattern is RRRRRGGG|GGGBBBBB.

'bgr565': 16 bit bgr triple (5 bit per pixel and channel, 6 bit for the green channel). The bit pattern is BBBB-BGGG|GGGRRRRR.

'rgb': 24 bit rgb triple (8 bit per pixel and channel)

'bgr': 24 bit bgr triple (8 bit per pixel and channel)

'rgbx': 32 bit rgb quadruple (8 bit per pixel and channel)

'bgrx': 32 bit bgr quadruple (8 bit per pixel and channel)

'rgb48': 48 bit rgb triple (16 bit per pixel and channel)

'bgr48': 48 bit bgr triple (16 bit per pixel and channel)

'rgbx64': 64 bit rgb quadruple (16 bit per pixel and channel)

'bgrx64': 64 bit bgr quadruple (16 bit per pixel and channel)

The values *'rgb555'*, *'bgr555'*, *'rgb565'*, *'bgr565'*, *'rgb5551'* and *'bgr5551'* can be used with the suffix 'le' (Little Endian: Lower byte is expected first) or - which is the default if the suffix is omitted - 'be' (Big Endian: Higher byte is expected first). For example, the bit pattern for `ColorFormat` = *'rgb555'* is XRRRRRGG|GGGBBBBB, while the bit pattern for `ColorFormat` = *'rgb555le'* is GGGBBBBB|XRRRRRGG.

Attention

`gen_image_interleaved` does not check whether the `PixelPointer` is valid or not. Thus, it must be ensured by the user that it is valid. Otherwise, the program may crash!

Parameters

▷ **`ImageRGB`** (output_object) ..... image ⇝ *object :* byte / uint2
Created image with new image matrix.

▷ **`PixelPointer`** (input_control) ..... pointer ⇝ *integer*
Pointer to interleaved pixels.

▷ **`ColorFormat`** (input_control) ..... string ⇝ *string*
Format of the input pixels.
Default: 'rgb'
List of values: `ColorFormat` ∈ {'rgb', 'bgr', 'rgbx', 'bgrx', 'rgb48', 'bgr48', 'rgbx64', 'bgrx64', 'rgb555', 'bgr555', 'rgb565', 'bgr565', 'rgb555le', 'bgr555le', 'rgb565le', 'bgr565le', 'rgb555be', 'bgr555be', 'rgb565be', 'bgr565be', 'rgb5551', 'bgr5551', 'rgb5551le', 'bgr5551le', 'rgb5551be', 'bgr5551be'}

▷ **`OriginalWidth`** (input_control) ..... extent.x ⇝ *integer*
Width of input image.
Default: 512
Suggested values: `OriginalWidth` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ OriginalWidth (lin)
Minimum increment: 1
Recommended increment: 10

▷ **`OriginalHeight`** (input_control) ..... extent.y ⇝ *integer*
Height of input image.
Default: 512
Suggested values: `OriginalHeight` ∈ {128, 256, 512, 1024}
Value range: 1 ≤ OriginalHeight (lin)
Minimum increment: 1
Recommended increment: 10

▷ **`Alignment`** (input_control) ..... integer ⇝ *integer*
Reserved.

▷ **`Type`** (input_control) ..... string ⇝ *string*
Pixel type of output image.
Default: 'byte'
List of values: `Type` ∈ {'byte', 'uint2'}

▷ **ImageWidth** (input_control) . . . . . rectangle.extent.x ⇝ *integer*
Width of output image.
Default: 0
Suggested values: ImageWidth ∈ {128, 256, 512, 1024}
Value range: 0 ≤ ImageWidth (lin)
Minimum increment: 1
Recommended increment: 10

▷ **ImageHeight** (input_control) . . . . . rectangle.extent.y ⇝ *integer*
Height of output image.
Default: 0
Suggested values: ImageHeight ∈ {128, 256, 512, 1024}
Value range: 0 ≤ ImageHeight (lin)
Minimum increment: 1
Recommended increment: 10

▷ **StartRow** (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Line number of upper left corner of desired image part.
Default: 0
Suggested values: StartRow ∈ {-1, 0}

▷ **StartColumn** (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column number of upper left corner of desired image part.
Default: 0
Suggested values: StartColumn ∈ {-1, 0}

▷ **BitsPerChannel** (input_control) . . . . . integer ⇝ *integer*
Number of used bits per pixel and channel of the output image (-1: All bits are used).
Default: -1
Suggested values: BitsPerChannel ∈ {5, 8, 10, 12, 16, -1}

▷ **BitShift** (input_control) . . . . . integer ⇝ *integer*
Number of bits that the color values of the input pixels are shifted to the right (only uint2 images).
Default: 0
Suggested values: BitShift ∈ {0, 2, 4, 6}

Result

If the parameter values are correct, the operator gen_image_interleaved returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

disp_color

See also

reduce_domain, paint_gray, paint_region, set_grayval, gen_image_const

Module

Foundation

gen_image_proto (Image : ImageCleared : Grayval :)

Create an image with a specified constant gray value.

gen_image_proto creates an output image ImageCleared with the constant gray value Grayval. If the input image is of type direction, gray values in result image that are not in the value range that is valid for direction images are set to the value 255 to mark them as invalid. ImageCleared has the same dimensions and pixel type as the input image Image.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image.

▷ **ImageCleared** (output_object) ..... image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Image with constant gray value.

▷ **Grayval** (input_control) ..... number ⇝ *real* / integer

Gray value to be used for the output image.
Default: 0
Suggested values: `Grayval` ∈ {0, 1, 2, 5, 10, 16, 32, 64, 128, 253, 254, 255}

Result

`gen_image_proto` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`set_grayval`, `paint_gray`, `gen_image_const`, `copy_image`

See also

`get_image_pointer1`

Module

Foundation

gen_image_surface_first_order (: ImageSurface : Type, Alpha, Beta, Gamma, Row, Column, Width, Height :)

Create a tilted gray surface with first order polynomial.

The operator `gen_image_surface_first_order` creates a tilted gray value surface according to the following equation:

$$\begin{aligned} \texttt{ImageSurface}(r,c) &= \texttt{Alpha}(r-\texttt{Row}) \\ &+ \texttt{Beta}(c-\texttt{Column}) \\ &+ \texttt{Gamma} \end{aligned}$$

The size of the image is determined by `Width` and `Height`. The parameters `Row` and `Column` define the reference point of the created gray surface. If `fit_surface_first_order` was used to determine the parameters of the gray surface, this reference point should correspond to the center of gravity that is used in the surface equation (see `fit_surface_first_order`). Its coordinates can be computed as follows:

```
intersection(ROI, Image, RegionIntersection)
fit_surface_first_order(RegionIntersection, Image, Algorithm,
Iterations, ClippingFactor, Alpha, Beta, Gamma)
area_center(RegionIntersection, Area, Row, Column)
gen_image_surface_first_order(ImageSurface, Type, Alpha, Beta,
Gamma, Row, Column, Width, Height)
```

The gray values are of the type `Type` (see `gen_image_const` for a detailed description of the pixel types). Gray values outside the valid area are clipped.

Parameters

▷ **ImageSurface** (output_object) ..... image ⇝ *object :* byte / uint2 / real
Created image with new image matrix.

▷ **Type** (input_control) ..... string ⇝ *string*
Pixel type.
Default: 'byte'
List of values: Type ∈ {'byte', 'uint2', 'real'}

▷ **Alpha** (input_control) ..... number ⇝ *real*
First order coefficient in vertical direction.
Default: 1.0
Suggested values: Alpha ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Beta** (input_control) ..... number ⇝ *real*
First order coefficient in horizontal direction.
Default: 1.0
Suggested values: Beta ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Gamma** (input_control) ..... number ⇝ *real*
Zero order coefficient.
Default: 1.0
Suggested values: Gamma ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Row** (input_control) ..... number ⇝ *real*
Row coordinate of the reference point of the surface.
Default: 256.0
Suggested values: Row ∈ {0.0, 128.0, 256.0, 512.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Column** (input_control) ..... number ⇝ *real*
Column coordinate of the reference point of the surface.
Default: 256.0
Suggested values: Column ∈ {0.0, 128.0, 256.0, 512.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of image.
Default: 512
Suggested values: Width ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of image.
Default: 512
Suggested values: Height ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

Result

If the parameter values are correct gen_image_surface_first_order returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fit_surface_first_order

Possible Successors

sub_image

See also

gen_image_gray_ramp, gen_image_surface_second_order, gen_image_const

Module

Foundation

gen_image_surface_second_order (: ImageSurface : Type, Alpha, Beta, Gamma, Delta, Epsilon, Zeta, Row, Column, Width, Height :)

Create a curved gray surface with second order polynomial.

The operator gen_image_surface_second_order creates a curved gray value surface according to the following equation:

$$\begin{aligned} \texttt{ImageSurface}(r,c) &= \texttt{Alpha}(r-\texttt{Row})^2 \\ &+ \texttt{Beta}(c-\texttt{Column})^2 \\ &+ \texttt{Gamma}(r-\texttt{Row})*(c-\texttt{Column}) \\ &+ \texttt{Delta}(r-\texttt{Row}) \\ &+ \texttt{Epsilon}(c-\texttt{Column}) \\ &+ \texttt{Zeta} \end{aligned}$$

The size of the image is determined by Width and Height. The parameters Row and Column define the reference point of the created gray surface. If fit_surface_second_order was used to determine the parameters of the gray surface, this reference point should correspond to the center of gravity that is used in the surface equation (see fit_surface_second_order). Its coordinates can be computed as follows:

```
intersection(ROI, Image, RegionIntersection)
fit_surface_second_order(RegionIntersection, Image, Algorithm,
Iterations, ClippingFactor, Alpha, Beta, Gamma, Delta, Epsilon,
Zeta)
area_center(RegionIntersection, Area, Row, Column)
gen_image_surface_second_order(ImageSurface, Type, Alpha, Beta,
Gamma, Delta, Epsilon, Zeta, Row, Column, Width, Height)
```

The gray values are of the type Type (see gen_image_const for a detailed description of the pixel types). Gray values outside the valid area are clipped.

Parameters

▷ **ImageSurface** (output_object) ..... image ⇝ *object :* byte / uint2 / real
Created image with new image matrix.

▷ **Type** (input_control) ..... string ⇝ *string*
Pixel type.
Default: 'byte'
List of values: Type ∈ {'byte', 'uint2', 'real'}

▷ **Alpha** (input_control) . . . . . number ⇝ *real*
Second order coefficient in vertical direction.
Default: 1.0
Suggested values: Alpha ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Beta** (input_control) . . . . . number ⇝ *real*
Second order coefficient in horizontal direction.
Default: 1.0
Suggested values: Beta ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Gamma** (input_control) . . . . . number ⇝ *real*
Mixed second order coefficient.
Default: 1.0
Suggested values: Gamma ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Delta** (input_control) . . . . . number ⇝ *real*
First order coefficient in vertical direction.
Default: 1.0
Suggested values: Delta ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Epsilon** (input_control) . . . . . number ⇝ *real*
First order coefficient in horizontal direction.
Default: 1.0
Suggested values: Epsilon ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Zeta** (input_control) . . . . . number ⇝ *real*
Zero order coefficient.
Default: 1.0
Suggested values: Zeta ∈ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Row** (input_control) . . . . . number ⇝ *real*
Row coordinate of the reference point of the surface.
Default: 256.0
Suggested values: Row ∈ {0.0, 128.0, 256.0, 512.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Column** (input_control) . . . . . number ⇝ *real*
Column coordinate of the reference point of the surface.
Default: 256.0
Suggested values: Column ∈ {0.0, 128.0, 256.0, 512.0}
Minimum increment: 0.000001
Recommended increment: -0.005

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of image.
Default: 512
Suggested values: Width ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of image.
Default: 512
Suggested values: Height ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

Example

```
* Adjust an inhomogeneous illumination
* using gen_image_surface_second_order
read_image (Image, 'cap_illumination/cap_illumination_01')
get_image_size (Image, Width, Height)
gen_circle (Circle, 495, 630, 350.5)
difference (Image, Circle, RegionDifference)
fit_surface_second_order (RegionDifference, Image, 'regression', 5, 2, \
                          Alpha, Beta, Gamma, Delta, Epsilon, Zeta)
area_center (RegionDifference, Area, Row, Column)
gen_image_surface_second_order (ImageSurface, 'byte', Alpha, Beta, \
                                Gamma, Delta, Epsilon, Zeta, Row, Column, \
                                Width, Height)
sub_image (Image, ImageSurface, ImageSub, 1, 128)
```

Result

If the parameter values are correct gen_image_surface_second_order returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fit_surface_second_order

Possible Successors

sub_image

See also

gen_image_gray_ramp, gen_image_surface_first_order, gen_image_const

Module

Foundation

interleave_channels (
MultichannelImage : InterleavedImage : PixelFormat, RowBytes,
Alpha :)

Create an interleaved image from a multichannel image.

The operator interleave_channels creates an InterleavedImage from the input MultichannelImage containing three or four channels. This is useful to prepare a color image for visualization, e.g., by converting it to a .NET Bitmap.

Pixel format

The format of the pixels in InterleavedImage is specified with PixelFormat. Possible formats are *'rgb'*, *'rgba'*, *'argb'*, *'bgr'*, *'bgra'*, and *'abgr'* for byte images and *'rgb48'*, *'bgr48'*, *'rgba64'*, *'bgra64'*, *'argb64'* and *'abgr64'* for uint2 images, where 'a' denotes the 4th ('alpha') channel. Be aware that, in contrast to gen_image_interleaved, LittleEndian byte ordering is used in PixelFormat.

Alpha channel

The alpha channel can be defined in the following ways:

channels	domain	
3	not specified/full	Value of `Alpha` is set for the complete image.
3	reduced	Inside the domain: alpha is set to `Alpha`. Outside the domain: alpha is set to zero.
4		Alpha is set to the respective value of the fourth channel, domain and `Alpha` are ignored.

Row alignment

The number of bytes between the beginning of two rows in the output image can be set in `RowBytes`, e.g., to fulfill specific alignment criteria. Set `RowBytes` to *'match'* if no additional padding is required. `RowBytes` must be at least the number of bytes in `PixelFormat` times the width of `MultichannelImage`.

Parameters

▷ **`MultichannelImage`** (input_object) . . . . . multichannel-image ⇝ *object :* byte
Input multichannel image.

▷ **`InterleavedImage`** (output_object) . . . . . singlechannelimage ⇝ *object :* byte
Output interleaved image.

▷ **`PixelFormat`** (input_control) . . . . . string ⇝ *string*
Target format for InterleavedImage.
Default: 'rgba'
List of values: `PixelFormat` ∈ {'rgb', 'bgr', 'rgba', 'abgr', 'argb', 'bgra', 'rgb48', 'bgr48', 'rgba64', 'bgra64', 'argb64', 'abgr64'}

▷ **`RowBytes`** (input_control) . . . . . integer ⇝ *string* / integer
Number of bytes in a row of the output image.
Default: 'match'
Suggested values: `RowBytes` ∈ {'match'}

▷ **`Alpha`** (input_control) . . . . . integer ⇝ *integer*
Alpha value for three channel input images.
Default: 255
Suggested values: `Alpha` ∈ {255, 0}

Example

```
HTuple type, width, height;
HImage patras = new HImage("patras");

HImage interleaved = patras.InterleaveChannels("argb", "match", 255);
IntPtr ptr = interleaved.GetImagePointer1(out type, out width, out height);

Image img = new Bitmap(width/4, height, width,
                       PixelFormat.Format32bppArgb, ptr);

pictureBox.Image = img;
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

See also

`gen_image_interleaved`

Module

Foundation

region_to_bin (Region : BinImage : ForegroundGray, BackgroundGray, Width, Height :)

Convert a region into a binary byte-image.

region_to_bin converts the input region given in Region into a byte-image and assigns a gray value of ForegroundGray to all pixels in the region. If the input region is larger than the generated image, it is clipped at the image borders. The background is set to BackgroundGray.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be converted.

▷ **BinImage** (output_object) ..... image ⇝ *object :* byte
Result image of dimension Width × Height containing the converted regions.

▷ **ForegroundGray** (input_control) ..... integer ⇝ *integer*
Gray value in which the regions are displayed.
Default: 255
Suggested values: ForegroundGray ∈ {0, 1, 50, 100, 128, 150, 200, 254, 255}
Value range: 0 ≤ ForegroundGray ≤ 255 (lin)
Recommended increment: 1

▷ **BackgroundGray** (input_control) ..... integer ⇝ *integer*
Gray value in which the background is displayed.
Default: 0
Suggested values: BackgroundGray ∈ {0, 1, 50, 100, 128, 150, 200, 254, 255}
Value range: 0 ≤ BackgroundGray ≤ 255 (lin)
Recommended increment: 1

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the image to be generated.
Default: 512
Suggested values: Width ∈ {256, 512, 1024}
Value range: 1 ≤ Width ≤ 1024 (lin)
Minimum increment: 1
Recommended increment: 16
Restriction: Width >= 1

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the image to be generated.
Default: 512
Suggested values: Height ∈ {256, 512, 1024}
Value range: 1 ≤ Height ≤ 1024 (lin)
Minimum increment: 1
Recommended increment: 16
Restriction: Height >= 1

Complexity

$O(2 * \texttt{Height} * \texttt{Width})$.

Result

region_to_bin always returns 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

get_grayval

Alternatives

region_to_label, paint_region, set_grayval

See also

gen_image_proto, paint_gray

Module

Foundation

region_to_label (Region : ImageLabel : Type, Width, Height :)

Convert regions to a label image.

region_to_label converts the input regions into a label image according to their index ($1..n$), i.e., the first region is painted with the gray value 1, the second the gray value 2, etc. Only positive gray values are used. For byte-images the index is entered modulo 256.

Regions larger than the generated image are clipped appropriately. If regions overlap the regions with the higher image are entered (i.e., they are painted in the order in which they are contained in the input regions). If so desired, the regions can be made non-overlapping by calling expand_region.

The background, i.e., the area not covered by any regions, is set to 0. This can be used to test in which image range no region is present.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be converted.

▷ **ImageLabel** (output_object) . . . . . image ⇝ *object :* byte / int2 / int4
Result image of dimension Width × Height containing the converted regions.

▷ **Type** (input_control) . . . . . string ⇝ *string*
Pixel type of the result image.
Default: 'int2'
List of values: Type ∈ {'byte', 'int2', 'int4', 'int8'}

▷ **Width** (input_control) . . . . . extent.y ⇝ *integer*
Width of the image to be generated.
Default: 512
Suggested values: Width ∈ {64, 128, 256, 512, 1024}
Value range: 1 ≤ Width ≤ 1024 (lin)
Minimum increment: 1
Recommended increment: 16
Restriction: Width >= 1

▷ **Height** (input_control) . . . . . extent.x ⇝ *integer*
Height of the image to be generated.
Default: 512
Suggested values: Height ∈ {64, 128, 256, 512, 1024}
Value range: 1 ≤ Height ≤ 1024 (lin)
Minimum increment: 1
Recommended increment: 16
Restriction: Height >= 1

Complexity

$O(2 * \text{Height} * \text{Width})$.

Result

region_to_label always returns 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, regiongrowing, connection, expand_region

Possible Successors

get_grayval, get_image_pointer1

Alternatives

region_to_bin, paint_region

See also

label_to_region

Module

Foundation

region_to_mean (Regions, Image : ImageMean : :)

Paint regions with their average gray value.

region_to_mean returns an image in which the regions Regions are painted with their average gray value based on the image Image. This operator is mainly intended to visualize segmentation results.

Parameters

▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
Input regions.

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
original gray-value image.

▷ **ImageMean** (output_object) . . . . . image ⇝ *object :* byte / uint2
Result image with painted regions.

Example

```
read_image(Image,'fabrik')
regiongrowing(Image,Regions,3,3,6,100)
region_to_mean(Regions,Image,Disp)
dev_display(Disp)
dev_set_draw('margin')
dev_display(Regions)
```

Result

region_to_mean returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on channel level.

Possible Predecessors

regiongrowing, connection

Possible Successors

disp_image

Alternatives

paint_region, intensity

Module

Foundation

15.5 Domain

add_channels (Regions, Image : GrayRegions : :)

Add gray values to regions.

The operator add_channels creates an array of images GrayRegions with an element for every input region in Regions. These images correspond to the input image Image with a reduced domain, namely the intersection of the definition domain of the input image with the region. Thus, the new definition domain can be a subset of the region. Thereby the size of the image matrix is not changed and all channels of Image are adopted.

Parameters

▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
Input regions (without pixel values).

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Input image with pixel values for regions.

▷ **GrayRegions** (output_object) . . . . . image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Output image(s) with regions and pixel values (one image per input region).
Number of elements: Regions == GrayRegions

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, gen_circle, draw_region

Possible Successors

threshold, regiongrowing, get_domain

Alternatives

change_domain, reduce_domain

See also

full_domain, get_domain, intersection

Module

Foundation

change_domain (Image, NewDomain : ImageNew : :)

Change definition domain of an image.

The operator change_domain uses the indicated region as the new definition domain. Unlike the operator reduce_domain it does not form the intersection of the previous definition domain, i.e., the size of the matrix is not changed. This implies in particular, that the region must not exceed the image matrix, otherwise using such inconsistent iconic objects during subsequent operations will likely lead to errors or system crashes.

Attention

Due to running time the transferred region is not checked for consistency (i.e., whether it fits with the image matrix). Incorrect regions lead to system hang-ups during subsequent operations.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image.

▷ **NewDomain** (input_object) ..... region ⇝ *object*

New definition domain.

▷ **ImageNew** (output_object) ..... image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Image with new definition domain.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`get_domain`

Alternatives

`reduce_domain`

See also

`full_domain`, `get_domain`, `intersection`

Module

Foundation

full_domain (Image : ImageFull : :)

Expand the domain of an image to maximum.

The operator `full_domain` enters a rectangle with the edge length of the image as new definition domain. This means that all pixels of the matrix are included in further operations. Thus the same definition domain is obtained as by reading or generating an image. The size of the matrix is not changed.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image.

▷ **ImageFull** (output_object) ..... image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Image with maximum definition domain.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`get_domain`

Alternatives

`change_domain`, `reduce_domain`

See also

get_domain, gen_rectangle1

Module

Foundation

get_domain (Image : Domain : :)

Get the domain of an image.

The operator get_domain returns the definition domains of all input images as a region.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input images.

▷ **Domain** (output_object) ..... region(-array) ⇝ *object*

Definition domains of input images.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

change_domain, reduce_domain, full_domain

See also

change_domain, reduce_domain, full_domain

Module

Foundation

rectangle1_domain (Image : ImageReduced : Row1, Column1, Row2, Column2 :)

Reduce the domain of an image to a rectangle.

The operator rectangle1_domain reduces the definition domain of the given image to the specified rectangle. The size of the matrix is not changed.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image.

▷ **ImageReduced** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Image with reduced definition domain.

▷ **Row1** (input_control) ..... rectangle.origin.y ⇝ *integer*

Line index of upper left corner of image area.
Default: 100
Suggested values: Row1 ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ Row1 ≤ 1024

▷ **Column1** (input_control) ..... rectangle.origin.x ⇝ *integer*
Column index of upper left corner of image area.
Default: 100
Suggested values: Column1 ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ Column1 ≤ 1024

▷ **Row2** (input_control) ..... rectangle.origin.y ⇝ *integer*
Line index of lower right corner of image area.
Default: 200
Suggested values: Row2 ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ Row2 ≤ 1024

▷ **Column2** (input_control) ..... rectangle.origin.x ⇝ *integer*
Column index of lower right corner of image area.
Default: 200
Suggested values: Column2 ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ Column2 ≤ 1024

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

get_domain

Alternatives

change_domain, reduce_domain, add_channels

See also

full_domain, get_domain, intersection

Module

Foundation

reduce_domain (Image, Region : ImageReduced : :)

Reduce the domain of an image.

The operator reduce_domain reduces the definition domain of the given image to the indicated region. The new definition domain is calculated as the intersection of the old definition domain with the region. Thus, the new definition domain can be a subset of the region. The size of the matrix is not changed.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Input image.

▷ **Region** (input_object) ..... region ⇝ *object*
New definition domain.

▷ **ImageReduced** (output_object) ..... image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Image with reduced definition domain.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.

Possible Predecessors

get_domain

Alternatives

change_domain, rectangle1_domain, add_channels

See also

full_domain, get_domain, intersection

Module

Foundation

15.6 Features

area_center_gray (Regions, Image : : : Area, Row, Column)

Compute the area and center of gravity of a region in a gray value image.

area_center_gray computes the area and center of gravity of the regions Regions that have gray values which are defined by the image Image. This operator is similar to area_center, but in contrast to that operator, the gray values of the image are taken into account while computing the area and center of gravity.

The area A of a region R in the image with the gray values $g(r, c)$ is defined as

$$A = \sum_{(r,c) \in R} g(r, c).$$

This means that the area is defined by the volume of the gray value function $g(r, c)$. The center of gravity is defined by the first two normalized moments of the gray values $g(r, c)$, i.e., by $(m_{1,0}, m_{0,1})$, where

$$m_{p,q} = \frac{1}{A} \sum_{(r,c) \in R} r^p c^q g(r, c).$$

Note, that in the case where the Area is zero the row and column coordinates of the center of gravity are also set to zero.

Attention

Note that the operator area_center_gray only considers the given Regions and ignores any previously set domain of the input image Image. area_center_gray can be executed on OpenCL devices if the device supports the cl_khr_fp64 and cl_khr_int64_base_atomics OpenCL extensions.

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Region(s) to be examined.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
  Gray value image.
- ▷ **Area** (output_control) ..... real(-array) ⇝ *real*
  Gray value volume of the region.
- ▷ **Row** (output_control) ..... point.y(-array) ⇝ *real*
  Row coordinate of the gray value center of gravity.
- ▷ **Column** (output_control) ..... point.x(-array) ⇝ *real*
  Column coordinate of the gray value center of gravity.

Result

area_center_gray returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

area_center

See also

area_center_xld, elliptic_axis_gray

Module

Foundation

cooc_feature_image (Regions, Image : : LdGray, Direction : Energy, Correlation, Homogeneity, Contrast)

Calculate a co-occurrence matrix and derive gray value features thereof.

The call of cooc_feature_image corresponds to the consecutive execution of the operators gen_cooc_matrix and cooc_feature_matrix. If several direction matrices of the co-occurrence matrix are to be evaluated consecutively, it is more efficient to generate the matrix via gen_cooc_matrix and then call the operator cooc_feature_matrix for the resulting matrix. The parameter Direction transfers the direction of the neighborhood in angle or *'mean'*. In the case of *'mean'* the mean value is calculated in all four directions.

Attention

Note that the operator cooc_feature_image only considers the given Regions and ignores any previously set domain of the input image Image.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region to be examined.

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte
Corresponding gray values.

▷ **LdGray** (input_control) ..... integer ⇝ *integer*
Number of gray values to be distinguished (2^{LdGray}).
Default: 6
List of values: LdGray ∈ {1, 2, 3, 4, 5, 6, 7, 8}

▷ **Direction** (input_control) ..... integer ⇝ *integer* / string
Direction in which the matrix is to be calculated.
Default: 0
List of values: Direction ∈ {0, 45, 90, 135, 'mean'}

▷ **Energy** (output_control) ..... real(-array) ⇝ *real*
Gray value energy.

▷ **Correlation** (output_control) ..... real(-array) ⇝ *real*
Correlation of gray values.

▷ **Homogeneity** (output_control) ..... real(-array) ⇝ *real*
Local homogeneity of gray values.

▷ **Contrast** (output_control) ..... real(-array) ⇝ *real*
Gray value contrast.

Result

The operator cooc_feature_image returns the value 2 (H_MSG_TRUE) if an image with defined gray values (byte) is entered and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:), the behavior in case of empty

region is set via set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_cooc_matrix

Alternatives

cooc_feature_matrix

See also

intensity, min_max_gray, entropy_gray, select_gray

Module

Foundation

cooc_feature_matrix (CoocMatrix : : : Energy, Correlation, Homogeneity, Contrast)

Calculate gray value features from a co-occurrence matrix.

The operator cooc_feature_matrix calculates from a co-occurrence matrix (CoocMatrix) the energy (Energy), correlation (Correlation), local homogeneity (Homogeneity) and contrast (Contrast).

The operator cooc_feature_matrix calculates the gray value features from the part of the input matrix generated by gen_cooc_matrix corresponding to the direction matrix indicated by the parameters LdGray and Direction according to the following formulae:

- Energy (Measure for image homogeneity):
  $\texttt{Energy} = \sum_{i,j=0}^{width} c_{ij}^2$
- Correlation (Measure for gray value dependencies):
  $\texttt{Correlation} = \frac{\sum_{i,j=0}^{width} (i-u_x)(j-u_y)c_{ij}}{s_x s_y}$
- Local homogeneity:
  $\texttt{Homogeneity} = \sum_{i,j=0}^{width} \frac{1}{1+(i-j)^2} c_{ij}$
- Contrast (Measure for the size of the intensity differences):
  $\texttt{Contrast} = \sum_{i,j=0}^{width} (i-j)^2 c_{ij}$

where

$width =$ Width of CoocMatrix,

c_{ij} Entry of co-occurrence matrix,

$u_x = \sum_{i,j=0}^{width} i * c_{ij}$,

$u_y = \sum_{i,j=0}^{width} j * c_{ij}$,

$s_x^2 = \sum_{i,j=0}^{width} (i-u_x)^2 * c_{ij}$,

$s_y^2 = \sum_{i,j=0}^{width} (i-u_y)^2 * c_{ij}$.

Attention

The region of the input image is disregarded.

Parameters

▷ **CoocMatrix** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
Co-occurrence matrix.

▷ **Energy** (output_control) . . . . . real ⇝ *real*
Homogeneity of the gray values.

▷ **Correlation** (output_control) . . . . . real ⇝ *real*
Correlation of gray values.

▷ **Homogeneity** (output_control) . . . . . real ⇝ *real*
Local homogeneity of gray values.

▷ **Contrast** (output_control) . . . . . real ⇝ *real*
Gray value contrast.

Result

The operator `cooc_feature_matrix` returns the value `2` (`H_MSG_TRUE`) if an image with defined gray values is passed and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`gen_cooc_matrix`

Alternatives

`cooc_feature_image`

See also

`intensity`, `min_max_gray`, `entropy_gray`, `select_gray`

Module

Foundation

elliptic_axis_gray (Regions, Image : : : Ra, Rb, Phi)

Compute the orientation and major axes of a region in a gray value image.

The operator `elliptic_axis_gray` calculates the length of the axes `Ra` and `Rb` and the orientation `Phi` of the ellipse having the "same orientation" and the "aspect ratio" as the input region in `Regions`. Therefore, gray value moments which are derived from the `Image` are used. Several input regions can be passed in `Regions` as tuples. The length of the major axis `Ra` and the minor axis `Rb` as well as the orientation of the major axis with regard to the x-axis (`Phi`) are determined. The angle is returned in radians. The calculation is done analogously to `elliptic_axis`. The only difference is that in `elliptic_axis_gray` the gray value moments are used instead of the region moments. For the definition of the gray value moments, see `area_center_gray`.

Note, that in the case where the gray value area is zero the length of the axes `Ra` and `Rb` as well as the orientation `Phi` are also set to zero.

Attention

Note that the operator `elliptic_axis_gray` only considers the given `Regions` and ignores any previously set domain of the input image `Image`.

Parameters

▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Gray value image.

▷ **Ra** (output_control) . . . . . real(-array) ⇝ *real*
Major axis of the region.

▷ **Rb** (output_control) . . . . . real(-array) ⇝ *real*
Minor axis of the region.

▷ **Phi** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Angle enclosed by the major axis and the x-axis.

Result

`elliptic_axis_gray` returns `2` (`H_MSG_TRUE`) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Possible Successors

`gen_ellipse`

Alternatives

`elliptic_axis`

See also

`area_center_gray`

Module

Foundation

entropy_gray (Regions, Image : : : Entropy, Anisotropy)

Determine the entropy and anisotropy of images.

The operator `entropy_gray` creates the histogram of relative frequencies of the gray values in the input image and calculates from these frequencies the entropy and the anisotropy coefficient for each region from `Regions` according to the following formulae:

Entropy:

$$Entropy = -\sum_{0}^{255} rel[i] * \log_2(rel[i])$$

Anisotropiy coefficient:

$$Anisotropy = \frac{\sum_{0}^{k} rel[i] * \log_2(rel[i])}{Entropy}$$

where

$rel[i]$ = Histogram of relative gray value frequencies
i = Gray value of input image $(0 \ldots 255)$
k = Smallest possible gray value with $\sum_{0}^{k} rel[i] \geq 0.5$

Attention

Note that the operator `entropy_gray` only considers the given `Regions` and ignores any previously set domain of the input image `Image`.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Regions where the features are to be determined.

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte
Gray value image.

▷ **Entropy** (output_control) ..... real(-array) ⇝ *real*
Information content (entropy) of the gray values.
Assertion: 0 <= Entropy && Entropy <= 8

▷ **Anisotropy** (output_control) ..... real(-array) ⇝ *real*
Measure of the symmetry of gray value distribution.

Complexity

If F is the area of the region the runtime complexity is $O(F + 255)$.

Result

The operator `entropy_gray` returns the value 2 (`H_MSG_TRUE`) if an image with defined gray values is entered and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)`, the behavior in case of empty region is set via `set_system(::'empty_region_result',<Result>:)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Alternatives

`select_gray`

See also

`entropy_image`, `gray_histo`, `gray_histo_abs`, `fuzzy_entropy`, `fuzzy_perimeter`

Module

Foundation

estimate_noise (Image : : Method, Percent : Sigma)

Estimate the image noise from a single image.

The operator `estimate_noise` estimates the standard deviation of additive noise within the domain of the image that is passed in `Image`. The standard deviation is returned in `Sigma`.

To estimate the noise, one of the following four methods can be selected in `Method`:

- *'foerstner'*: If `Method` is set to *'foerstner'*, first for each pixel a homogeneity measure is computed based on the first derivatives of the gray values of `Image`. By thresholding the homogeneity measure one obtains the homogeneous regions in the image. The threshold is computed based on a starting value for the image noise. The starting value is obtained by applying the method *'immerkaer'* (see below) in the first step. It is assumed that the gray value fluctuations within the homogeneous regions are solely caused by the image noise. Furthermore it is assumed that the image noise is Gaussian distributed. The average homogeneity measure within the homogeneous regions is then used to calculate a refined estimate for the image noise. The refined estimate leads to a new threshold for the homogeneity. The described process is iterated until the estimated image noise remains constant between two successive iterations. Finally, the standard deviation of the estimated image noise is returned in `Sigma`.

  Note that in some cases the iteration falsely converges to the value *0*. This happens, for example, if the gray value histogram of the input image contains gaps that are caused either by an automatic radiometric scaling of the camera or frame grabber, respectively, or by a manual spreading of the gray values using a scaling factor > 1.

  Also note that the result obtained by this method is independent of the value passed in `Percent`.

- *'immerkaer'*: If Method is set to *'immerkaer'*, first the following filter mask is applied to the input image:

$$M = \begin{vmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{vmatrix} .$$

  The advantage of this method is that M is almost insensitive to image structure but only depends on the noise in the image. Assuming a Gaussian distributed noise, its standard deviation is finally obtained as

$$\texttt{Sigma} = \sqrt{\frac{\pi}{2}} \frac{1}{6N} \sum_{\texttt{Image}} |\texttt{Image} * M| ,$$

  where N is the number of image pixels to which M is applied. Note that the result obtained by this method is independent of the value passed in Percent.
- *'least_squares'*: If Method is set to *'least_squares'*, the fluctuations of the gray values with respect to a locally fitted gray value plane are used to estimate the image noise. First, a homogeneity measure is computed based on the first derivatives of the gray values of Image. Homogeneous image regions are determined by selecting the Percent percent most homogeneous pixels in the domain of the input image, i.e., pixels with small magnitudes of the first derivatives. For each homogeneous pixel a gray value plane is fitted to its 3×3 neighborhood. The differences between the gray values within the 3×3 neighborhood and the locally fitted plane are used to estimate the standard deviation of the noise. Finally, the average standard deviation over all homogeneous pixels is returned in Sigma.
- *'mean'*: If Method is set to *'mean'*, the noise estimation is based on the difference between the input image and a noiseless version of the input image. First, a homogeneity measure is computed based on the first derivatives of the gray values of Image. Homogeneous image regions are determined by selecting the Percent percent most homogeneous pixels in the domain of the input image, i.e., pixels with small magnitudes of the first derivatives. A mean filter is applied to the homogeneous image regions in order to eliminate the noise. It is assumed that the difference between the input image and the thus obtained noiseless version of the image represents the image noise. Finally, the standard deviation of the differences is returned in Sigma. It should be noted that this method requires large connected homogeneous image regions to be able to reliably estimate the noise.

Note that the methods *'foerstner'* and *'immerkaer'* assume a Gaussian distribution of the image noise, whereas the methods *'least_squares'* and *'mean'* can be applied to images with arbitrarily distributed noise. In general, the method *'foerstner'* returns the most accurate results while the method *'immerkaer'* shows the fastest computation.

If the image noise could not be estimated reliably, the error 3175 is raised. This may happen if the image does not contain enough homogeneous regions, if the image was artificially created, or if the noise is not of Gaussian type. In order to avoid this error, it might be useful in some cases to try one of the following modifications in dependence of the estimation method that is passed in Method:

- Increase the size of the input image domain (useful for all methods).
- Increase the value of the parameter Percent (useful for methods *'least_squares'* and *'mean'*).
- Use the method *'immerkaer'*, instead of the methods *'foerstner'*, *'least_squares'*, or *'mean'*. The method *'immerkaer'* does not rely on the existence of homogeneous image regions, and hence is almost always applicable.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2
Input image.

▷ **Method** (input_control) ..... string ⇝ *string*
Method to estimate the image noise.
Default: 'foerstner'
List of values: Method ∈ {'foerstner', 'immerkaer', 'least_squares', 'mean'}

▷ **Percent** (input_control) ..... number ⇝ *real* / integer
Percentage of used image points.
Default: 20
Suggested values: Percent ∈ {1, 2, 5, 7, 10, 15, 20, 30, 40, 50}
Restriction: 0 < Percent && Percent <= 50.

▷ **Sigma** (output_control) ..... real(-array) ⇝ *real*
Standard deviation of the image noise.
Assertion: Sigma >= 0

Example

```
read_image (Image, 'combine')
estimate_noise (ImageNoise, 'foerstner', 20, SigmaFoerstner)
estimate_noise (ImageNoise, 'immerkaer', 20, SigmaImmerkaer)
estimate_noise (ImageNoise, 'least_squares', 20, SigmaLeastSquares)
estimate_noise (ImageNoise, 'mean', 20, SigmaMean)
```

Result

If the parameters are valid, the operator estimate_noise returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the image noise could not be estimated reliably, the error 3175 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

grab_image, grab_image_async, read_image, reduce_domain

Possible Successors

binomial_filter, gauss_filter, mean_image, smooth_image

Alternatives

noise_distribution_mean, intensity, min_max_gray

See also

gauss_distribution, add_noise_distribution

References

W. Förstner: "Image Preprocessing for Feature Extraction in Digital Intensity, Color and Range Images", Springer Lecture Notes on Earth Sciences, Summer School on Data Analysis and the Statistical Foundations of Geomatics, 1999

J. Immerkaer: "Fast Noise Variance Estimation", Computer Vision and Image Understanding, Vol. 64, No. 2, pp. 300-302, 1996

Module

Foundation

fit_surface_first_order (Regions, Image : : Algorithm, Iterations, ClippingFactor : Alpha, Beta, Gamma)

Calculate gray value moments and approximation by a first order surface (plane).

The operator fit_surface_first_order calculates the gray value moments and the parameters of the approximation of the gray values by a first order surface. The calculation is done by minimizing the distance between the gray values and the surface. A first order surface is described by the following formula:

$$\begin{aligned} \texttt{Image}(r,c) &= \texttt{Alpha}(r - r_{center}) \\ &+ \texttt{Beta}(c - c_{center}) \\ &+ \texttt{Gamma} \end{aligned}$$

r_{center} and c_{center} are the center coordinates of intersection of the input region with the full image domain. By the minimization process the parameters from Alpha to Gamma is calculated.

The algorithm used for the fitting can be selected via Algorithm:

'regression' Standard 'least squares' line fitting.

'huber' Weighted 'least squares' fitting, where the impact of outliers is decreased based on the approach of Huber.

'tukey' Weighted 'least squares' fitting, where outliers are ignored based on the approach of Tukey.

The parameter ClippingFactor (a scaling factor for the standard deviation) controls the amount of damping outliers: The smaller the value chosen for ClippingFactor the more outliers are detected. The detection of outliers is repeated. The parameter Iterations specifies the number of iterations. If *'regression'* is set for Algorithm Iterations is ignored.

Attention

Note that the operator fit_surface_first_order only considers the given Regions and ignores any previously set domain of the input image Image.

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be checked.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / direction / cyclic / real
  Corresponding gray values.
- ▷ **Algorithm** (input_control) ..... string ⇝ *string*
  Algorithm for the fitting.
  Default: 'regression'
  List of values: Algorithm ∈ {'regression', 'huber', 'tukey'}
- ▷ **Iterations** (input_control) ..... integer ⇝ *integer*
  Maximum number of iterations (unused for 'regression').
  Default: 5
  Restriction: Iterations >= 0
- ▷ **ClippingFactor** (input_control) ..... real ⇝ *real*
  Clipping factor for the elimination of outliers.
  Default: 2.0
  List of values: ClippingFactor ∈ {1.0, 1.5, 2.0, 2.5, 3.0}
  Restriction: ClippingFactor > 0
- ▷ **Alpha** (output_control) ..... real(-array) ⇝ *real*
  Parameter Alpha of the approximating surface.
- ▷ **Beta** (output_control) ..... real(-array) ⇝ *real*
  Parameter Beta of the approximating surface.
- ▷ **Gamma** (output_control) ..... real(-array) ⇝ *real*
  Parameter Gamma of the approximating surface.

Result

The operator fit_surface_first_order returns the value 2 (H_MSG_TRUE) if an image with the defined gray values (byte) is entered and the parameters are correct. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Successors

gen_image_surface_first_order

See also

moments_gray_plane, fit_surface_second_order

Module

Foundation

fit_surface_second_order (Regions, Image : : Algorithm, Iterations, ClippingFactor : Alpha, Beta, Gamma, Delta, Epsilon, Zeta)

Calculate gray value moments and approximation by a second order surface.

The operator fit_surface_second_order calculates the gray value moments and the parameters of the approximation of the gray values by a second order surface. The calculation is done by minimizing the distance between the gray values and the surface. A second order surface is described by the following formula:

$$\begin{aligned} \texttt{Image}(r,c) &= \texttt{Alpha}(r-r_{center})^2 \\ &+ \texttt{Beta}(c-c_{center})^2 \\ &+ \texttt{Gamma}(r-r_{center})*(c-c_{center}) \\ &+ \texttt{Delta}(r-r_{center}) \\ &+ \texttt{Epsilon}(c-c_{center}) \\ &+ \texttt{Zeta} \end{aligned}$$

r_{center} and c_{center} are the center coordinates of the intersection of the input region with the full image domain. By the minimization process the parameters from Alpha to Zeta is calculated.

The algorithm used for the fitting can be selected via Algorithm:

'regression' Standard 'least squares' fitting.

'huber' Weighted 'least squares' fitting, where the impact of outliers is decreased based on the approach of Huber.

'tukey' Weighted 'least squares' fitting, where outliers are ignored based on the approach of Tukey.

The parameter ClippingFactor (a scaling factor for the standard deviation) controls the amount of damping outliers: The smaller the value chosen for ClippingFactor the more outliers are detected. The detection of outliers is repeated. The parameter Iterations specifies the number of iterations. If *'regression'* is set for Algorithm Iterations is ignored.

Attention

Note that the operator fit_surface_second_order only considers the given Regions and ignores any previously set domain of the input image Image.

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be checked.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / direction / cyclic / real
  Corresponding gray values.
- ▷ **Algorithm** (input_control) ..... string ⇝ *string*
  Algorithm for the fitting.
  Default: 'regression'
  List of values: Algorithm ∈ {'regression', 'tukey', 'huber'}
- ▷ **Iterations** (input_control) ..... integer ⇝ *integer*
  Maximum number of iterations (unused for 'regression').
  Default: 5
  Restriction: Iterations >= 0
- ▷ **ClippingFactor** (input_control) ..... real ⇝ *real*
  Clipping factor for the elimination of outliers.
  Default: 2.0
  List of values: ClippingFactor ∈ {1.0, 1.5, 2.0, 2.5, 3.0}
  Restriction: ClippingFactor > 0
- ▷ **Alpha** (output_control) ..... real(-array) ⇝ *real*
  Parameter Alpha of the approximating surface.
- ▷ **Beta** (output_control) ..... real(-array) ⇝ *real*
  Parameter Beta of the approximating surface.

▷ **Gamma** (output_control) . . . . . real(-array) ⇝ *real*
Parameter Gamma of the approximating surface.
▷ **Delta** (output_control) . . . . . real(-array) ⇝ *real*
Parameter Delta of the approximating surface.
▷ **Epsilon** (output_control) . . . . . real(-array) ⇝ *real*
Parameter Epsilon of the approximating surface.
▷ **Zeta** (output_control) . . . . . real(-array) ⇝ *real*
Parameter Zeta of the approximating surface.

Result

The operator `fit_surface_second_order` returns the value `2` (`H_MSG_TRUE`) if an image with the defined gray values (byte) is entered and the parameters are correct. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Successors

`gen_image_surface_second_order`

See also

`moments_gray_plane`, `fit_surface_first_order`

Module

Foundation

fuzzy_entropy (Regions, Image : : Apar, Cpar : Entropy)

Determine the fuzzy entropy of regions.

`fuzzy_entropy` calculates the fuzzy entropy of a fuzzy set. To do so, the image is regarded as a fuzzy set. The entropy then is a measure of how well the image approximates a white or black image. It is defined as follows:

$$H(X) \quad = \quad \frac{1}{MNln2} \sum_l T_e(l) h(l)$$

where $M \times N$ is the size of the image, and $h(l)$ is the histogram of the image. Furthermore,

$$T_e(l) \quad = \quad -\mu(l) \ln \mu(l) - (1 - \mu(l)) \ln(1 - \mu(l))$$

Here, $u(x(m,n))$ is a fuzzy membership function defining the fuzzy set (see `fuzzy_perimeter`). The same restrictions hold as in `fuzzy_perimeter`.

Attention

Note that for `fuzzy_entropy`, the `Regions` must lie completely within the previously defined domain. Otherwise an exception is raised.

Parameters

▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
Regions for which the fuzzy entropy is to be calculated.
▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte
Input image containing the fuzzy membership values.
▷ **Apar** (input_control) . . . . . integer ⇝ *integer*
Start of the fuzzy function.
Default: 0
Suggested values: `Apar` ∈ {0, 5, 10, 20, 50, 100}
Value range: 0 ≤ `Apar` ≤ 255 (lin)
Minimum increment: 1
Recommended increment: 5

▷ **Cpar** (input_control) . . . . . integer ⇝ *integer*
End of the fuzzy function.
Default: 255
Suggested values: Cpar ∈ {50, 100, 150, 200, 220, 255}
Value range: 0 ≤ Cpar ≤ 255 (lin)
Minimum increment: 1
Recommended increment: 5
Restriction: Apar <= Cpar

▷ **Entropy** (output_control) . . . . . real(-array) ⇝ *real*
Fuzzy entropy of a region.

Example

```
* To find a Fuzzy Entropy from an Image
read_image(Image,'monkey')
fuzzy_entropy(Trans,Trans,0,255,Entro)
```

Result

The operator fuzzy_entropy returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

See also

fuzzy_perimeter

References

M.K. Kundu, S.K. Pal: "Automatic selection of object enhancement operator with quantitative justification based on fuzzy set theoretic measures"; Pattern Recognition Letters 11; 1990; pp. 811-829.

Module

Foundation

fuzzy_perimeter (Regions, Image : : Apar, Cpar : Perimeter)

Calculate the fuzzy perimeter of a region.

The operator fuzzy_perimeter is used to determine the differences of fuzzy membership between an image point and its neighbor points. The right and lower neighbor are taken into account. The fuzzy perimeter is then defined as follows:

$$p(X) = \sum_{m=1}^{M-1}\sum_{n=1}^{N-1} |\mu_X(x_{m,n}) - \mu_X(x_{m,n+1})| + \sum_{m=1}^{M-1}\sum_{n=1}^{N-1} |\mu_X(x_{m,n}) - \mu_X(x_{m+1,n})|$$

where $M \times N$ is the size of the image, and $u(x(m,n))$ is the fuzzy membership function (i.e., the input image). This implementation uses Zadeh's Standard-S function, which is defined as follows:

$$\mu_X(x) = \begin{cases} 0, & x \le a \\ 2\left(\frac{x-a}{c-a}\right)^2, & a < x \le b \\ 1 - 2\left(\frac{x-a}{c-a}\right)^2, & b < x \le c \\ 1, & c \le x \end{cases}$$

The parameters a, b and c obey the following restrictions: $b = \frac{a+c}{2}$ is the inflection point of the function, $\Delta b = b - a = c - b$ is the bandwidth, and for $x = b$ $\mu(x) = 0.5$ holds. In `fuzzy_perimeter`, the parameters `Apar` and `Cpar` are defined as follows: b is $\frac{\texttt{Apar}+\texttt{Cpar}}{2}$.

Attention

Note that for `fuzzy_perimeter`, the `Regions` must lie completely within the previously defined domain. Otherwise an exception is raised.

Parameters

- ▷ **`Regions`** (input_object) ..... region(-array) ⇝ *object*
  Regions for which the fuzzy perimeter is to be calculated.
- ▷ **`Image`** (input_object) ..... singlechannelimage ⇝ *object :* byte
  Input image containing the fuzzy membership values.
- ▷ **`Apar`** (input_control) ..... integer ⇝ *integer*
  Start of the fuzzy function.
  Default: 0
  Suggested values: `Apar` ∈ {0, 5, 10, 20, 50, 100}
  Value range: 0 ≤ `Apar` ≤ 255 (lin)
  Minimum increment: 1
  Recommended increment: 5
- ▷ **`Cpar`** (input_control) ..... integer ⇝ *integer*
  End of the fuzzy function.
  Default: 255
  Suggested values: `Cpar` ∈ {50, 100, 150, 200, 220, 255}
  Value range: 0 ≤ `Cpar` ≤ 255 (lin)
  Minimum increment: 1
  Recommended increment: 5
  Restriction: Apar <= Cpar
- ▷ **`Perimeter`** (output_control) ..... real(-array) ⇝ *real*
  Fuzzy perimeter of a region.

Example

```
* To find a Fuzzy Entropy from an Image
read_image(Image,'monkey')
fuzzy_perimeter(Trans,Trans,0,255,Per)
```

Result

The operator `fuzzy_perimeter` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

See also

`fuzzy_entropy`

References

M.K. Kundu, S.K. Pal: “Automatic selection of object enhancement operator with quantitative justification based on fuzzy set theoretic measures”; Pattern Recognition Letters 11; 1990; pp. 811-829.

Module

Foundation

`gen_cooc_matrix` (Regions, Image : Matrix : LdGray, Direction :)

Calculate the co-occurrence matrix of a region in an image.

The operator gen_cooc_matrix determines from the input regions how often the gray values i and j are located next to each other in a certain direction (*0*, *45*, *90*, *135* degrees), stores this number in the co-occurrence matrix at the locations (i, j) and (j, i) (the matrix is symmetrical), and finally scales the matrix with the number of entries. LdGray indicates the number of gray values to be distinguished (namely 2^{LdGray}).

Example: Input image with gray values (scaled with LdGray=2)

$$\begin{matrix} 0 & 0 & 3 \\ 1 & 1 & 2 \\ 1 & 2 & 3 \end{matrix}$$

Co-occurrence matrix (not scaled):

0:
$$\begin{matrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 2 & 0 \\ 0 & 2 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{matrix}$$

135:
$$\begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

90:
$$\begin{matrix} 0 & 2 & 0 & 0 \\ 2 & 2 & 1 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 2 & 0 \end{matrix}$$

45:
$$\begin{matrix} 0 & 1 & 0 & 0 \\ 1 & 2 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

Attention

Note that the operator gen_cooc_matrix only considers the given Regions and ignores any previously set domain of the input image Image.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region to be checked.

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte
Image providing the gray values.

▷ **Matrix** (output_object) ..... image(-array) ⇝ *object :* real
Co-occurrence matrix (matrices).

▷ **LdGray** (input_control) ..... integer ⇝ *integer*
Number of gray values to be distinguished (2^{LdGray}).
Default: 6
List of values: LdGray ∈ {1, 2, 3, 4, 5, 6, 7, 8}
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Direction** (input_control) ..... integer ⇝ *integer*
Direction of neighbor relation.
Default: 0
List of values: Direction ∈ {0, 45, 90, 135}

Result

The operator gen_cooc_matrix returns the value 2 (H_MSG_TRUE) if an image with defined gray values is entered and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:), the behavior in case of empty region is set via set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

draw_region, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, threshold, erosion_circle, binomial_filter, gauss_filter, smooth_image, sub_image

Alternatives

cooc_feature_image

See also

cooc_feature_matrix

Module

Foundation

gray_features (Regions, Image : : Features : Value)

Calculates gray value features for a set of regions.

gray_features has a set of regions (Regions) as input. For each of these regions the features (Features) are calculated and returned in Value.

Possible values for Features:

- *'area'*: Gray value volume of region (see area_center_gray)
- *'row'*: Row index of the center of gravity (see area_center_gray)
- *'column'*: Column index of the center of gravity (see area_center_gray)
- *'ra'*: Major axis of equivalent ellipse (see elliptic_axis_gray)
- *'rb'*: Minor axis of equivalent ellipse (see elliptic_axis_gray)
- *'phi'*: Orientation of equivalent ellipse (see elliptic_axis_gray)
- *'min'*: Minimum gray value (see min_max_gray)
- *'max'*: Maximum gray value (see min_max_gray)
- *'median'*: Median gray value (see min_max_gray, with Percent=*50*)
- *'mean'*:Mean gray value (see intensity)
- *'deviation'*: Deviation of gray values (see intensity)
- *'plane_deviation'*: Deviation from the approximating plane (see plane_deviation)
- *'anisotropy'*: Anisotropy (see entropy_gray)
- *'entropy'*: Entropy (see entropy_gray)
- *'fuzzy_entropy'*: Fuzzy entropy of region (see fuzzy_entropy, with a fuzzy function from Apar=*0* to Cpar=*255*)
- *'fuzzy_perimeter'*: Fuzzy perimeter of region (see fuzzy_perimeter, with a fuzzy function from Apar=*0* to Cpar=*255*)
- *'moments_row'*: Mixed moments along a row (see moments_gray_plane)
- *'moments_column'*: Mixed moments along a column (see moments_gray_plane)
- *'alpha'*: Approximating plane, parameter Alpha (see moments_gray_plane)
- *'beta'*: Approximating plane, parameter Beta (see moments_gray_plane)

Attention

Several features are processed in the order in which they are entered.

Note that the operator gray_features only considers the given Regions and ignores any previously set domain of the input image Image.

Parameters

▷ **Regions** (input_object) . . . . . region-array ⇝ *object*
Regions to be examined.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Gray value image.

▷ **Features** (input_control) . . . . . string(-array) ⇝ *string*
Names of the features.
Default: 'mean'
List of values: Features ∈ {'area', 'row', 'column', 'ra', 'rb', 'phi', 'min', 'max', 'median', 'mean', 'deviation', 'plane_deviation', 'anisotropy', 'entropy', 'fuzzy_entropy', 'fuzzy_perimeter', 'moments_row', 'moments_column', 'alpha', 'beta'}

▷ **Value** (output_control) . . . . . real(-array) ⇝ *real*
Values of the features.

Complexity

If F is the area of the region and N the number of features the runtime complexity is $O(F * N)$.

Result

The operator `gray_features` returns the value `2` (`H_MSG_TRUE`) if the input image has the defined gray values and the parameters are correct. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`connection`, `mean_image`, `entropy_image`, `sobel_amp`, `median_separate`

Possible Successors

`select_gray`, `shape_trans`, `reduce_domain`, `count_obj`

See also

`select_gray`, `deviation_image`, `entropy_gray`, `intensity`, `mean_image`, `min_max_gray`, `select_obj`

Module

Foundation

gray_histo (Region, Image : : : AbsoluteHisto, RelativeHisto)

Calculate the gray value distribution.

The operator `gray_histo` calculates for the image (`Image`) within `Region` the absolute (`AbsoluteHisto`) and relative (`RelativeHisto`) histogram of the gray values.

Both histograms are tuples of 256 values, which — beginning at 0 — contain the frequencies of the individual gray values of the image.

`AbsoluteHisto` indicates the absolute frequencies of the gray values in integers, and `RelativeHisto` indicates the relative, i.e. the absolute frequencies divided by the area of the image as floating point numbers.

'real'-, *'int2'*-, *'uint2'*-, and *'int4'*-images are transformed into *'byte'*-images (first the largest and smallest gray value in the image are determined, and then the original gray values are mapped linearly into the area 0..255) and then processed as mentioned above.

Attention

Note that the operator `gray_histo` only considers the given `Region` and ignores any previously set domain of the input image `Image`.

Real, int2, uint2, and int4 images are reduced to 256 gray values.

`gray_histo` can be executed on OpenCL devices for byte, int1, directional and cyclic images if the OpenCL device supports the cl_khr_local_int32_base_atomics and cl_khr_global_int32_base_atomics extensions (which are optional in OpenCL 1.0). Unlike most other HALCON operators, `gray_histo` can be substantially slower on an OpenCL device if the `Image` is not completely included in the `Region`.

OpenCL support is not available in HALCON XL, as 64 bit atomic integer arithmetic would be required.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
Region in which the histogram is to be calculated.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / cyclic / direction / int1 / int2 / uint2 / int4 / real
Image the gray value distribution of which is to be calculated.

▷ **AbsoluteHisto** (output_control) . . . . . histogram-array ⇝ *integer*
Absolute frequencies of the gray values.

▷ **RelativeHisto** (output_control) . . . . . histogram-array ⇝ *real*
Frequencies, normalized to the area of the region.

Complexity

If F is the area of the region the runtime complexity is $O(F + 255)$.

Result

The operator gray_histo returns the value 2 (H_MSG_TRUE) if the image has defined gray values and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:), the behavior in case of empty region is set via set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

histo_to_thresh, gen_region_histo

Alternatives

min_max_gray, intensity, gray_histo_abs, gray_histo_range

See also

set_paint, histo_2dim, scale_image_max, entropy_gray

Module

Foundation

gray_histo_abs (Region, Image : : Quantization : AbsoluteHisto)

Calculate the gray value distribution.

The operator gray_histo_abs calculates for the image (Image) within Region the absolute (AbsoluteHisto)) histogram of the gray values.

The parameter Quantization defines, how many frequencies of neighbored gray values are added for one frequency value. The resulting histogram AbsoluteHisto is a tuple, whose indices are mapped on the gray values of the input image Image and whose elements contain the frequencies of the gray values. The indices i of the frequency value are calculated from the gray values g and the quantization q as follows:

- For unsigned image types:

$$i = \left\lceil \frac{g + 0.5}{q} \right\rceil$$

- For signed image types:

$$i = \left\lceil \frac{g - (MIN - 0.5)}{q} \right\rceil$$

whereas MIN denotes the minimal gray value, e.g., -128 for an *int1* image type. Therefore, the size of the tuple results from the ratio of the full domain of gray values and the quantization, e.g., for images of *int2* in $\lceil \frac{65536}{3.0} \rceil = 21846$. The origin gray value of the signed image types *int1 resp. int2* is mapped on the index 128 resp. 32768, negative resp. positive gray values have smaller resp. greater indices.

Attention

Note that the operator gray_histo_abs only considers the given Region and ignores any previously set domain of the input image Image.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
Region in which the histogram is to be calculated.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object* : byte / cyclic / direction / int1 / int2 / uint2
Image the gray value distribution of which is to be calculated.

▷ **Quantization** (input_control) . . . . . number ⇝ *real* / integer
Quantization of the gray values.
Default: 1.0
Suggested values: Quantization ∈ {1.0, 2.0, 3.0, 5.0, 10.0}
Restriction: Quantization >= 1.0

▷ **AbsoluteHisto** (output_control) . . . . . histogram-array ⇝ *integer*
Absolute frequencies of the gray values.

Result

The operator gray_histo_abs returns the value 2 (H_MSG_TRUE) if the image has defined gray values and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:), the behavior in case of empty region is set via set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

histo_to_thresh, gen_region_histo

Alternatives

min_max_gray, intensity, gray_histo, gray_histo_range

See also

disp_image, histo_2dim, scale_image_max, entropy_gray

Module

Foundation

gray_histo_range (Region, Image : : Min, Max, NumBins : Histo, BinSize)

Calculate the gray value distribution of a single channel image within a certain gray value range.

gray_histo_range calculates the gray value distribution Histo of the single channel Image within Region and the gray value range [Min,Max]. The values for Min and Max are rounded down to the next integer if the Image is not of type *'real'*. The gray value range is divided into NumBins bins of the same size, which is returned in BinSize. If a gray value lies between two bins the gray value is assigned to the smaller bin. If the Image is of type *'real'* $\texttt{BinSize} = \frac{\texttt{Max}-\texttt{Min}}{\texttt{NumBins}}$. If the Image has discrete gray values the size of a bin is computed with $\texttt{BinSize} = \frac{\texttt{Max}-\texttt{Min}+1}{\texttt{NumBins}}$. Since the accuracy of the histogram Histo can be adjusted with NumBins, the calculation of the gray value histogram with gray_histo_range is most useful for images of type *'real'* and images of type *'integer'*, which have a high bit depth.

Attention

Note that the operator gray_histo_range only considers the given Region and ignores any previously set domain of the input image Image.

If the Image has discrete gray values the value of BinSize may cause the following effects: For $\texttt{BinSize} > 1$ multiple neighboring gray values are assigned to the same bin. If BinSize is no integer the gray values are distributed uneven among the bins, e.g., for $\texttt{BinSize} = 1.5$ the first and second gray value are assigned to the first bin, the third gray value is assigned to the second bin, and the fourth and fifth gray value are assigned to the third bin. This becomes noticeable in several peeks in the histogram Histo. If $\texttt{BinSize} < 1$ some classes are not assigned by any gray value, e.g., for $\texttt{BinSize} = 0.5$ the first gray value is assigned to the first bin and the second

gray value is assigned to the third bin. The histogram `Histo` shows some gaps, which resembles the structure of a comb.

If the `Image` is of type *'real'* and `Min` = `Max`, all pixels of the corresponding gray value are assigned only to the first bin.

Parameters

▷ **`Region`** (input_object) ..... region ⇝ *object*
Region in which the histogram is to be calculated.

▷ **`Image`** (input_object) ..... singlechannelimage ⇝ *object :* byte / cyclic / direction / int1 / uint2 / int2 / int4 / int8 / real
Input image.

▷ **`Min`** (input_control) ..... real ⇝ *real* / integer
Minimum gray value.
Default: 0
Suggested values: `Min` ∈ {0}

▷ **`Max`** (input_control) ..... real ⇝ *real* / integer
Maximum gray value.
Default: 255
Suggested values: `Max` ∈ {255}
Restriction: Max >= Min

▷ **`NumBins`** (input_control) ..... integer ⇝ *integer*
Number of bins.
Default: 256
Suggested values: `NumBins` ∈ {16, 32, 64, 128, 256}
Restriction: NumBins >= 1

▷ **`Histo`** (output_control) ..... histogram(-array) ⇝ *integer*
Histogram to be calculated.

▷ **`BinSize`** (output_control) ..... real ⇝ *real*
Bin size.

Result

If the parameters are valid, the operator `gray_histo_range` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`min_max_gray`

Possible Successors

`create_funct_1d_array`

Alternatives

`gray_histo`, `gray_histo_abs`

See also

`histo_2dim`, `scale_image_max`

Module

Foundation

`gray_projections` (Region, Image : : Mode : HorProjection, VertProjection)

Calculate horizontal and vertical gray-value projections.

gray_projections calculates the horizontal and vertical gray-value projections, i.e., the mean values in the horizontal and vertical direction of the gray values of the input image Image within the input region Region.

If Mode = *'simple'* is selected the projection is performed in the direction of the coordinate axes of the image, i.e.:

$$\begin{aligned} \texttt{HorProjection}(r) &= \frac{1}{n(r+r')} \sum_{(r+r',c+c') \in \texttt{Region}} \texttt{Image}(r+r',c+c') \\ \texttt{VertProjection}(c) &= \frac{1}{n(c+c')} \sum_{(r+r',c+c') \in \texttt{Region}} \texttt{Image}(r+r',c+c') \end{aligned}$$

Here, (r', c') denotes the upper left corner of the smallest enclosing axis-parallel rectangle of the input region (see smallest_rectangle1), and $n(x)$ denotes the number of region points in the corresponding row $r + r'$ or column $c + c'$. Hence, the horizontal projection returns a one-dimensional function that reflects the vertical gray value changes. Likewise, the vertical projection returns a function that reflects the horizontal gray value changes.

If Mode = *'rectangle'*is selected the projection is performed in the direction of the major axes of the smallest enclosing rectangle of arbitrary orientation of the input region (see smallest_rectangle2). Here, the horizontal projection direction corresponds to the larger axis, while the vertical direction corresponds to the smaller axis. In this mode, all gray values within the smallest enclosing rectangle of arbitrary orientation of the input region are used to compute the projections.

Attention

The operator gray_projections only considers the given Region and ignores any previously set domain of the input image Image.

If $n(x) = 0$, i.e., if there are no region points in the corresponding row $r + r'$ or column $c + c'$, the respective value of HorProjection or VertProjection is set to -1.

gray_projections can be executed on an OpenCL device for the *'simple'* mode if the OpenCL device supports the cl_khr_global_int32_base_atomics OpenCL extension. For processing images of type real, the OpenCL device must support the cl_khr_fp64 and cl_khr_int64_base_atomics extensions.

Parameters

- ▷ **Region** (input_object) ..... region ⇝ *object*
  Region to be processed.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / int2 / uint2 / real
  Grayvalues for projections.
- ▷ **Mode** (input_control) ..... string ⇝ *string*
  Method to compute the projections.
  Default: 'simple'
  List of values: Mode ∈ {'simple', 'rectangle'}
- ▷ **HorProjection** (output_control) ..... real-array ⇝ *real*
  Horizontal projection.
- ▷ **VertProjection** (output_control) ..... real-array ⇝ *real*
  Vertical projection.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

1D Metrology

histo_2dim (Regions, ImageCol, ImageRow : Histo2Dim : :)

Calculate the histogram of two-channel gray value images.

The operator histo_2dim calculates the 2-dimensional histogram of two images within Regions. The gray values of channel 1 (ImageCol) are interpreted as row index, those of channel 2 (ImageRow) as column index. The gray value at one point $P(g1, g2)$ in the output image Histo2Dim indicates the frequency of the gray value combination ($g1$,$g2$) with $g1$ indicating the line index and $g2$ the column index.

Attention

Note that the operator histo_2dim only considers the given Regions and ignores any previously set domain of the input images.

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Region in which the histogram is to be calculated.
- ▷ **ImageCol** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1
  Channel 1.
- ▷ **ImageRow** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1
  Channel 2.
- ▷ **Histo2Dim** (output_object) ..... image ⇝ *object :* int4
  Histogram to be calculated.

Example

```
read_image(Image,'monkey')
get_domain (Image, Domain)
gauss_filter (Image, ImageGauss, 7)
histo_2dim(Domain,ImageGauss, Image,Histo2Dim)
dev_display(Histo2Dim)
```

Complexity

If F is the plane of the region, the runtime complexity is $O(F + 256^2)$.

Result

The operator histo_2dim returns the value 2 (H_MSG_TRUE) if both images have defined gray values. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:), the behavior in case of empty region is set via set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

decompose3, decompose2, draw_region

Possible Successors

threshold, class_2dim_sup, pouring, local_max, gray_skeleton

Alternatives

gray_histo, gray_histo_abs

See also

get_grayval

Module

Foundation

intensity (Regions, Image : : : Mean, Deviation)

Calculate the mean and deviation of gray values.

The operator intensity calculates the mean and the deviation of the gray values in the input image within Regions. If R is a region, p a pixel from R with the gray value $g(p)$ and F the plane ($F = |R|$), the features are defined by:

$$\texttt{Mean} = \frac{\sum_{p \in R} g(p)}{F}$$

$$\texttt{Deviation} = \sqrt{\frac{\sum_{p \in R} (g(p) - \texttt{Mean})^2}{F}}$$

Attention

Note that the operator intensity only considers the given Regions and ignores any previously set domain of the input image Image. The calculation of Deviation does not follow the usual definition if the region of the image contains only one pixel. In this case 0.0 is returned.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Regions in which the features are calculated.

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Gray value image.

▷ **Mean** (output_control) ..... real(-array) ⇝ *real*
Mean gray value of a region.

▷ **Deviation** (output_control) ..... real(-array) ⇝ *real*
Deviation of gray values within a region.

Complexity

If F is the area of the region, the runtime complexity is $O(F)$.

Result

The operator intensity returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:), the behavior in case of empty region is set via set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

threshold

Alternatives

select_gray, min_max_gray

See also

mean_image, gray_histo, gray_histo_abs

Module

Foundation

min_max_gray (Regions, Image : : Percent : Min, Max, Range)

Determine the minimum and maximum gray values within regions.

The operator min_max_gray creates the histogram of the absolute frequencies of the gray values within Regions in the input image Image (see gray_histo) and calculates the number of pixels Percent corresponding to the area of the input image. Then it goes inwards on both sides of the histogram by this number of pixels and determines the smallest and the largest gray value:

Example:

With area = 60, Percent = *5*, d.h. i.e. 3 pixels, histogram = [2,8,0,7,13,0,0,...,0,10,10,5,3,1,1],
⇒ Minimum = 0, Maximum = 255, range = 255
min_max_gray returns: Min = *1*, Max = *253*, Range = *252*.

For images of type int4, int8, and real, the above calculation is not performed via histograms, but using a rank selection algorithm. If Percent is set to *50*, Min = Max = Median. If Percent is *0* no histogram is calculated in order to enhance the runtime.

Attention

Note that the operator min_max_gray only considers the given Regions and ignores any previously set domain of the input image Image.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Regions, the features of which are to be calculated.

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real
Gray value image.

▷ **Percent** (input_control) ..... number ⇝ *real* / integer
Percentage below (above) the absolute maximum (minimum).
Default: 0
Suggested values: Percent ∈ {0, 1, 2, 5, 7, 10, 15, 20, 30, 40, 50}
Restriction: 0 <= Percent && Percent <= 50

▷ **Min** (output_control) ..... real(-array) ⇝ *real*
"Minimum" gray value.

▷ **Max** (output_control) ..... real(-array) ⇝ *real*
"Maximum" gray value.
Assertion: Max >= Min

▷ **Range** (output_control) ..... real(-array) ⇝ *real*
Difference between Max and Min.
Assertion: Range >= 0

Example

```
* Threshold segmentation with training region:
read_image(Image,'fabrik')
draw_region(Region,WindowHandle)
min_max_gray(Region,Image,5,Min,Max,Range)
threshold(Image,SegmentedRegion,Min,Max)
dev_display(SegmentedRegion)
```

Result

The operator min_max_gray returns the value 2 (H_MSG_TRUE) if the input image has the defined gray values and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:). The behavior in case of an empty region is set via the operator set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

draw_region, gen_circle, gen_ellipse, gen_rectangle1, threshold, regiongrowing

Possible Successors

threshold

Alternatives

select_gray, intensity

See also

gray_histo, scale_image, scale_image_max, learn_ndim_norm

Module

Foundation

moments_gray_plane (Regions, Image : : : MRow, MCol, Alpha, Beta, Mean)

Calculate gray value moments and approximation by a plane.

The operator moments_gray_plane calculates the gray value moments and the parameters of the approximation of the gray values by a plane. The calculation is carried out according to the following formula:

$$\begin{aligned}
\texttt{MRow} &= \frac{1}{F^2} \sum_{(r,c)\in\texttt{Regions}} (r-\overline{r})(\texttt{Image}(r,c)-\texttt{Mean}) \\
\texttt{MCol} &= \frac{1}{F^2} \sum_{(r,c)\in\texttt{Regions}} (c-\overline{c})(\texttt{Image}(r,c)-\texttt{Mean}) \\
\texttt{Alpha} &= \frac{\texttt{MRow}Fm_{02}-m_{11}\texttt{MCol}}{F} m_{20}m_{02}-m_{11}^2 \\
\texttt{Beta} &= \frac{m_{20}\texttt{MCol}F-\texttt{MRow}Fm_{11}}{m_{20}m_{02}-m_{11}^2}
\end{aligned}$$

where F is the plane, $\overline{r}$, $\overline{c}$ the center, and m_{11}, m_{20}, and m_{02} the scaled moments of Regions.

The parameters Alpha, Beta and Mean describe a plane above the region:

$$\texttt{Image}'(r,c) = \texttt{Alpha}(r-\overline{r}) + \texttt{Beta}(c-\overline{c}) + \texttt{Mean}$$

Thus Alpha indicates the gradient in the direction of the line axis (“down”), Beta the gradient in the direction of the column axis (to the “right”).

Attention

Note that the operator moments_gray_plane only considers the given Regions and ignores any previously set domain of the input image Image.

Parameters

- ▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
  Regions to be checked.
- ▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / direction / cyclic / uint2 / real
  Corresponding gray values.
- ▷ **MRow** (output_control) . . . . . real(-array) ⇝ *real*
  Mixed moments along a line.
- ▷ **MCol** (output_control) . . . . . real(-array) ⇝ *real*
  Mixed moments along a column.
- ▷ **Alpha** (output_control) . . . . . real(-array) ⇝ *real*
  Parameter Alpha of the approximating plane.
- ▷ **Beta** (output_control) . . . . . real(-array) ⇝ *real*
  Parameter Beta of the approximating plane.
- ▷ **Mean** (output_control) . . . . . real(-array) ⇝ *real*
  Mean gray value.

Result

The operator moments_gray_plane returns the value 2 (H_MSG_TRUE) if an image with the defined gray values (byte) is entered and the parameters are correct. The behavior in case of empty input (no input images

available) is set via the operator set_system(::'no_object_result',<Result>:), the behavior in case of empty region is set via set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

draw_region, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, threshold, regiongrowing

See also

intensity, moments_region_2nd

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, pp 75-76

Module

Foundation

plane_deviation (Regions, Image : : : Deviation)

Calculate the deviation of the gray values from the approximating image plane.

The operator plane_deviation calculates the deviation of the gray values in Image from the approximation of the gray values through a plane. Contrary to the standard deviation in case of intensity slanted gray value planes also receive the value zero. The gray value plane is calculated according to gen_image_gray_ramp. If F is the area of the region, α, β, and μ the parameters of the image plane, and (r', c') the center of the region, Deviation is defined by:

$$\texttt{Deviation} = \sqrt{\frac{sum_{(r,c) \in \texttt{Regions}}((\alpha(r-r') + \beta(c-c') + \mu) - \texttt{Image}(r,c))^2}{F}} \; .$$

Attention

Note that the operator plane_deviation only considers the given Regions and ignores any previously set domain of the input image Image. It should be noted that the calculation of Deviation does not follow the usual definition. It is defined to return the value 0.0 for an image with only one pixel.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Regions, of which the plane deviation is to be calculated.

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / cyclic
Gray value image.

▷ **Deviation** (output_control) ..... real(-array) ⇝ *real*
Deviation of the gray values within a region.

Complexity

If F is the area of the region the runtime complexity amounts to $O(F)$.

Result

The operator plane_deviation returns the value 2 (H_MSG_TRUE) if Image is of the type byte. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:), the behavior in case of empty region is set via set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Alternatives

intensity, gen_image_gray_ramp, sub_image

See also

moments_gray_plane

Module

Foundation

select_gray (Regions, Image : SelectedRegions : Features, Operation, Min, Max :)

Select regions based on gray value features.

The operator select_gray has a number of regions (Regions) as input. For each of these regions the (Features) are calculated. The region is transferred (duplicated) into the output SelectedRegions, if each (Operation = *'and'*) or at least one (Operation = *'or'*) of the calculated features is within the limits. The limits are defined by the parameters Min and Max. Thereby, Min and Max can also be set to *'min'* or *'max'*, respectively, corresponding to setting the smallest or largest value possible for this feature. The parameter Image contains an image which returns the gray values for calculating the features.

Restriction:

$$\mathtt{Min}_i \leq \mathtt{Features}_i(\mathtt{Regions}, \mathtt{Image}) \leq \mathtt{Max}_i$$

Possible values for Features:

- *'area'*: Gray value volume of region (see area_center_gray)
- *'row'*: Row index of the center of gravity (see area_center_gray)
- *'column'*: Column index of the center of gravity (see area_center_gray)
- *'ra'*: Major axis of equivalent ellipse (see elliptic_axis_gray)
- *'rb'*: Minor axis of equivalent ellipse (see elliptic_axis_gray)
- *'phi'*: Orientation of equivalent ellipse (see elliptic_axis_gray)
- *'min'*: Minimum gray value (see min_max_gray)
- *'max'*: Maximum gray value (see min_max_gray)
- *'median'*: Median gray value (see min_max_gray, with Percent=*50*)
- *'mean'*: Mean gray value (see intensity)
- *'deviation'*: Deviation of gray values (see intensity)
- *'plane_deviation'*: Deviation from the approximating plane (see plane_deviation)
- *'anisotropy'*: Anisotropy (see entropy_gray)
- *'entropy'*: Entropy (see entropy_gray)
- *'fuzzy_entropy'*: Fuzzy entropy of region (see fuzzy_entropy, with a fuzzy function from Apar=*0* to Cpar=*255*)
- *'fuzzy_perimeter'*: Fuzzy perimeter of region (see fuzzy_perimeter, with a fuzzy function from Apar=*0* to Cpar=*255*)
- *'moments_row'*: Mixed moments along a row (see moments_gray_plane)
- *'moments_column'*: Mixed moments along a column (see moments_gray_plane)
- *'alpha'*: Approximating plane, parameter Alpha (see moments_gray_plane)
- *'beta'*: Approximating plane, parameter Beta (see moments_gray_plane)

Attention

Note that the operator `select_gray` only considers the given `Regions` and ignores any previously set domain of the input image `Image`. If only one feature is used the value of `Operation` is meaningless. Several features are processed in the order in which they are entered. The maximum number of features is limited to 100.

Parameters

▷ **`Regions`** (input_object) . . . . . region-array ⇝ *object*
Regions to be examined.

▷ **`Image`** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Gray value image.

▷ **`SelectedRegions`** (output_object) . . . . . region-array ⇝ *object*
Regions having features within the limits.

▷ **`Features`** (input_control) . . . . . string(-array) ⇝ *string*
Names of the features.
Default: 'mean'
List of values: `Features` ∈ {'area', 'row', 'column', 'ra', 'rb', 'phi', 'min', 'max', 'median', 'mean', 'deviation', 'plane_deviation', 'anisotropy', 'entropy', 'fuzzy_entropy', 'fuzzy_perimeter', 'moments_row', 'moments_column', 'alpha', 'beta'}

▷ **`Operation`** (input_control) . . . . . string ⇝ *string*
Logical connection of features.
Default: 'and'
List of values: `Operation` ∈ {'and', 'or'}

▷ **`Min`** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Lower limit(s) of features or 'min'.
Default: 128.0
Suggested values: `Min` ∈ {0.5, 1.0, 10.0, 20.0, 50.0, 128.0, 255.0, 1000.0, 'min'}

▷ **`Max`** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Upper limit(s) of features or 'max'.
Default: 255.0
Suggested values: `Max` ∈ {0.5, 1.0, 10.0, 20.0, 50.0, 128.0, 255.0, 1000.0, 'max'}

Complexity

If F is the area of the region and N the number of features the runtime complexity is $O(F * N)$.

Result

The operator `select_gray` returns the value `2` (`H_MSG_TRUE`) if the input image has the defined gray values and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result',<Result>:)`, the behavior in case of empty region is set via `set_system(::'empty_region_result',<Result>:)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`connection`, `mean_image`, `entropy_image`, `sobel_amp`, `median_separate`

Possible Successors

`select_shape`, `shape_trans`, `reduce_domain`, `count_obj`

See also

`deviation_image`, `entropy_gray`, `intensity`, `mean_image`, `min_max_gray`, `select_obj`

Module

Foundation

shape_histo_all (Region, Image : : Feature : AbsoluteHisto, RelativeHisto)

Determine a histogram of features along all threshold values.

The operator shape_histo_all carries out 255 threshold operations within Region with the gray values of Image. The entry i in the histogram corresponds to the number of connected components/holes of this image segmented with the threshold i (Feature = *'connected_components'*, *'holes'*) or the mean value of the feature values of the regions segmented in this way (Feature = *'convexity'*, *'compactness'*, *'anisometry'*), respectively.

Attention

The operator shape_histo_all expects a region and exactly one gray value image as input. Because of the power of this operator the runtime of shape_histo_all is relatively large!

Note that the operator shape_histo_all only considers the given Region and ignores any previously set domain of the input image Image.

Parameters

- ▷ **Region** (input_object) ..... region ⇝ *object*
  Region in which the features are to be examined.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object : byte*
  Gray value image.
- ▷ **Feature** (input_control) ..... string ⇝ *string*
  Feature to be examined.
  Default: 'connected_components'
  List of values: Feature ∈ {'connected_components', 'convexity', 'compactness', 'anisometry', 'holes'}
- ▷ **AbsoluteHisto** (output_control) ..... histogram-array ⇝ *real / integer*
  Absolute distribution of the feature.
- ▷ **RelativeHisto** (output_control) ..... histogram-array ⇝ *real*
  Relative distribution of the feature.

Example

```
* Simulation of shape_histo_all with feature 'connected_components':
* my_shape_histo_all(Region,Image,AbsHisto,RelHisto):
reduce_domain(Region,Image,RegionGray)
for i := 0 to 255 by 1
    threshold(RegionGray,Seg,i,255)
    connect_and_holes (Seg, NumConnected, _)
    AbsHisto[i] := NumConnected
endfor
Sum := 0
for i := 0 to 255 by 1
    Sum := Sum+AbsHisto[i]
endfor
for i := 0 to 255 by 1
    RelHisto[i] := AbsHisto[i]/Sum
endfor
```

Complexity

If F is the area of the input region and N the mean number of connected components the runtime complexity is $O(255(F + \sqrt{F}\sqrt{N}))$.

Result

The operator shape_histo_all returns the value 2 (H_MSG_TRUE) if an image with the defined gray values is entered. The behavior in case of empty input (no input images) is set via the operator set_system('no_object_result',<Result>:), the behavior in case of empty region is set via set_system('empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

histo_to_thresh, threshold, gen_region_histo

Alternatives

shape_histo_point

See also

compactness, connection, connect_and_holes, convexity, count_obj, entropy_gray, gray_histo

Module

Foundation

shape_histo_point (Region, Image : : Feature, Row, Column : AbsoluteHisto, RelativeHisto)

Determine a histogram of features along all threshold values.

Like shape_histo_all the operator shape_histo_point carries out 255 threshold value operations within Region with the gray values of Image. Contrary to shape_histo_all only the segmented region containing the pixel (Row, Column) is taken into account here. The entry i in the histogram then corresponds to the number of holes of this region segmented with the threshold i (Feature = *'holes'*) or the feature value of the region (Feature = *'convexity'*, *'compactness'*, *'anisometry'*), respectively.

Attention

Note that the operator shape_histo_point only considers the given Region and ignores any previously set domain of the input image Image.

Parameters

▷ **Region** (input_object) ..... region ⇝ *object*
Region in which the features are to be examined.

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte
Gray value image.

▷ **Feature** (input_control) ..... string ⇝ *string*
Feature to be examined.
Default: 'convexity'
List of values: Feature ∈ {'convexity', 'compactness', 'anisometry', 'holes'}

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Row of the pixel which the region must contain.
Default: 256
Suggested values: Row ∈ {10, 50, 100, 200, 300, 400}

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column of the pixel which the region must contain.
Default: 256
Suggested values: Column ∈ {10, 50, 100, 200, 300, 400}

▷ **AbsoluteHisto** (output_control) ..... histogram-array ⇝ *real* / integer
Absolute distribution of the feature.

▷ **RelativeHisto** (output_control) ..... histogram-array ⇝ *real*
Relative distribution of the feature.

Result

The operator shape_histo_point returns the value 2 (H_MSG_TRUE) if an image with defined gray values is entered. The behavior in case of empty input (no input images available) is set via the operator set_system(::'no_object_result',<Result>:), the behavior in case of empty region is set via set_system(::'empty_region_result',<Result>:). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_mbutton, area_center

Possible Successors

histo_to_thresh, threshold, gen_region_histo

Alternatives

shape_histo_all

See also

connection, connect_and_holes, convexity, compactness

Module

Foundation

15.7 Format

add_image_border (Image : ImageBorder : Size, Value :)

Add a border to an image.

Adds a border to the input image Image and returns the resulting image in ImageBorder.

The size of the border in pixels must be passed in Size. It can be set as follows:

- Single number: Border size in all four directions left/right/top/bottom.
- Two numbers: Border size on left/right and top/bottom side: [l/r, t/b].
- Four numbers: Border size on left, right, top, bottom side: [l,r,t,b].

The gray value of the added border must be passed in Value. It can be set as follows:

- Single number: Value will be applied to all channels of Image.
- Multiple numbers: Values will be applied to the channels of the image separately. Thus, the first value will be applied to the first channel, the second value will be applied to the second channel and so on.
  Restriction: Number of values must be equal to the number of channels.

Note that add_image_border will ignore the input domain and will always return an image with a full domain in ImageBorder.

Parameters

▷ **Image** (input_object) ..... multichannel-image-array ⇝ *object :* byte / direction / int1 / int2 / uint2 / int4 / int8 / real
Input image.

▷ **ImageBorder** (output_object) ..... multichannel-image-array ⇝ *object :* byte / direction / int1 / int2 / uint2 / int4 / int8 / real
Output image.

▷ **Size** (input_control) ..... attribute.name(-array) ⇝ *integer*
Size of the border in pixels.
Default: 10

▷ **Value** (input_control) ..... attribute.name(-array) ⇝ *integer* / string
Gray value of the border.
Default: 100

Example

```
* Add a border of 20 pixels with a constant gray value of 255
* on all four sides of Image
read_image (Image, 'printer_chip/printer_chip_01')
add_image_border (Image, ImageBorder, 20, 255)

* Add a 10-pixel border with a constant gray value of 255 to
* the left and right side and a 20-pixel border with the same
* gray value to the top and bottom of Image.
read_image (Image, 'printer_chip/printer_chip_01')
add_image_border (Image, ImageBorder, [10, 20], 255)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Alternatives

`tile_images_offset`

Module

Foundation

change_format (Image : ImagePart : Width, Height :)

Change image size.

The operator `change_format` increases or decreases the size of the input images to the indicated height or width, respectively. If the image is reduced, parts are cut off at the "right" or "lower" edge of the image, respectively. If the image is enlarged, the additional areas are set to 0. The definition domain of the new image is equal to the domain of the input image, clipped to the size of the new image. No zooming is carried out.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Input image.

▷ **ImagePart** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Image with new format.

▷ **Width** (input_control) ..... extent.x ⇝ *integer*

Width of new image.
Default: 512
Suggested values: `Width` ∈ {32, 64, 128, 256, 512, 768, 1024}

▷ **Height** (input_control) ..... extent.y ⇝ *integer*

Height of new image.
Default: 512
Suggested values: `Height` ∈ {32, 64, 128, 256, 512, 525, 1024}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

disp_image

Alternatives

crop_part

See also

zoom_image_size, zoom_image_factor

Module

Foundation

crop_domain (Image : ImagePart : :)

Cut out of defined gray values.

The operator crop_domain cuts a rectangular area from the input images. This rectangle is the smallest surrounding rectangle of the domain of the input image. The new image matrix has the size of the rectangle.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Input image.

▷ **ImagePart** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Image area.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

disp_image

Alternatives

crop_part, crop_rectangle1, change_format, reduce_domain

See also

zoom_image_size, zoom_image_factor

Module

Foundation

crop_domain_rel (Image : ImagePart : Top, Left, Bottom, Right :)

Cut out an image area relative to the domain.

crop_domain_rel cuts a rectangular area from the input images. The area is determined by the surrounding rectangle of the domain of the input image. The rectangle can be influenced by the control parameters to modify at the top (Top), at the left (Left), at the bottom (Bottom), and at the right (Right). Positive values results in a smaller, negative values in a larger size. If all parameters are set to zero, the region remains unchanged.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real

Input image.

▷ **ImagePart** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image area.

▷ **Top** (input_control) ..... integer ⇝ *integer*
Number of rows clipped at the top.
Default: -1
Suggested values: Top ∈ {-20, -10, -5, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20}

▷ **Left** (input_control) ..... integer ⇝ *integer*
Number of columns clipped at the left.
Default: -1
Suggested values: Left ∈ {-20, -10, -5, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20}

▷ **Bottom** (input_control) ..... integer ⇝ *integer*
Number of rows clipped at the bottom.
Default: -1
Suggested values: Bottom ∈ {-20, -10, -5, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20}

▷ **Right** (input_control) ..... integer ⇝ *integer*
Number of columns clipped at the right.
Default: -1
Suggested values: Right ∈ {-20, -10, -5, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20}

Result

crop_domain_rel returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Possible Predecessors

reduce_domain, threshold, connection, regiongrowing, pouring

Alternatives

crop_domain, crop_rectangle1

See also

smallest_rectangle1, intersection, gen_rectangle1, clip_region

Module

Foundation

crop_part (Image : ImagePart : Row, Column, Width, Height :)

Cut out one or more rectangular image areas.

The operator crop_part cuts one or more rectangular area from each of the input images. The areas are indicated by rectangles, which are defined by the coordinates of their upper left corner and by their size. The upper left corners of the rectangles must be within the image. At the right side and at the bottom, the rectangles may exceed the image, but the domain of the output images is set so that only the part that can be derived from the input image is contained. If the rectangular areas fall completely within the image, then each of the resulting images has the size of its corresponding rectangle.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real
Input image.

▷ **ImagePart** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real
Image area.

▷ **Row** (input_control) ..... rectangle.origin.y(-array) ⇝ *integer*
Line index of upper left corner of image area.
Default: 100
Suggested values: `Row` ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ `Row` ≤ 1024

▷ **Column** (input_control) ..... rectangle.origin.x(-array) ⇝ *integer*
Column index of upper left corner of image area.
Default: 100
Suggested values: `Column` ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ `Column` ≤ 1024

▷ **Width** (input_control) ..... rectangle.extent.x(-array) ⇝ *integer*
Width of new image.
Default: 128
Suggested values: `Width` ∈ {32, 64, 128, 256, 512, 768}
Value range: 0 ≤ `Width` ≤ 1024

▷ **Height** (input_control) ..... rectangle.extent.y(-array) ⇝ *integer*
Height of new image.
Default: 128
Suggested values: `Height` ∈ {32, 64, 128, 256, 512, 525}
Value range: 0 ≤ `Height` ≤ 1024

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

`disp_image`

Alternatives

`crop_rectangle1`, `crop_domain`, `change_format`, `reduce_domain`

See also

`zoom_image_size`, `zoom_image_factor`

Module

Foundation

crop_rectangle1 (Image : ImagePart : Row1, Column1, Row2, Column2 :)

Cut out one or more rectangular image areas.

The operator `crop_rectangle1` cuts one or more rectangular areas from each of the input images. The areas are indicated by rectangles, which are defined by the coordinates of their upper left and their lower right corners. The upper left corners must be within the image. At the right side and at the bottom, the rectangles may exceed the image, but the domain of the output images is set so that only the part that can be derived from the input image is contained. If the rectangular areas fall within the image, then each of the resulting images has the size of its corresponding rectangle.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real
Input image.

▷ **ImagePart** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real
Image area.

▷ **Row1** (input_control) ..... rectangle.origin.y(-array) ⇝ *integer*
Line index of upper left corner of image area.
Default: 100
Suggested values: `Row1` ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ `Row1` ≤ 1024

▷ **Column1** (input_control) ..... rectangle.origin.x(-array) ⇝ *integer*
Column index of upper left corner of image area.
Default: 100
Suggested values: `Column1` ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ `Column1` ≤ 1024

▷ **Row2** (input_control) ..... rectangle.corner.y(-array) ⇝ *integer*
Line index of lower right corner of image area.
Default: 200
Suggested values: `Row2` ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ `Row2` ≤ 1024

▷ **Column2** (input_control) ..... rectangle.corner.x(-array) ⇝ *integer*
Column index of lower right corner of image area.
Default: 200
Suggested values: `Column2` ∈ {10, 20, 50, 100, 200, 300, 500}
Value range: 0 ≤ `Column2` ≤ 1024

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

`disp_image`

Alternatives

`crop_part`, `crop_domain`, `change_format`, `reduce_domain`

See also

`zoom_image_size`, `zoom_image_factor`

Module

Foundation

crop_rectangle2 (Image : ImagePart : Row, Column, Phi, Length1, Length2, AlignToAxis, Interpolation :)

Cut out one or more arbitrarily oriented rectangular image areas.

`crop_rectangle2` cuts one or more rectangular areas from each of the input images. The areas are indicated by rectangles, which are defined by the center (`Row`, `Column`), the orientation `Phi` and the half edge lengths `Length1` and `Length2`. `Phi` is given in arc measure in mathematically positive direction and indicates the angle between the horizontal axis and the first edge (with length `Length1`). The rectangle parameters use pixel centered, subpixel accurate coordinates, see Transformations / 2D Transformations.

The parameter `AlignToAxis` determines the alignment of the output image. Possible values:

- *'true'*: The cropped image part is rotated such that it is aligned with the coordinate axes. For this, a suitable interpolation must be performed. A detailed description of the interpolation modes can be found in the description of `affine_trans_image`. The rectangle may protrude the input image. Gray values of pixels lying outside the input image are set to 0.
- *'false'*: The smallest surrounding rectangle of the input rectangle is cut out and the domain is set accordingly. The rectangle may exceed the input image boundaries. However, the size of the resulting cropped image part is determined by the portion of the rectangle which lies within the input image boundaries. For this method no interpolation is required. Therefore, the parameter `Interpolation` has no effect.

Parameters

▷ **`Image`** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Input image(s).

▷ **`ImagePart`** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Cropped image part(s).

▷ **`Row`** (input_control) . . . . . rectangle2.center.y(-array) ⇝ *real* / integer
Row index of the image crop center.
Default: 300.0
Suggested values: `Row` ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
(lin)
Recommended increment: 10.0

▷ **`Column`** (input_control) . . . . . rectangle2.center.x(-array) ⇝ *real* / integer
Column index of the image crop center.
Default: 200.0
Suggested values: `Column` ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
(lin)
Recommended increment: 10.0

▷ **`Phi`** (input_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real* / integer
Orientation of the rectangle (arc measure).
Default: 0.0
Suggested values: `Phi` ∈ {-1.178097, -0.785398, -0.392699, 0.0, 0.392699, 0.785398, 1.178097}
Value range: -1.178097 ≤ `Phi` ≤ 1.178097 (lin)

▷ **`Length1`** (input_control) . . . . . rectangle2.hwidth(-array) ⇝ *real* / integer
First half edge length of the rectangle.
Default: 100.0
Suggested values: `Length1` ∈ {3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0, 300.0, 500.0}
(lin)
Recommended increment: 10.0

▷ **`Length2`** (input_control) . . . . . rectangle2.hheight(-array) ⇝ *real* / integer
Second half edge length of the rectangle.
Default: 20.0
Suggested values: `Length2` ∈ {1.0, 2.0, 3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0}
(lin)
Recommended increment: 10.0

▷ **`AlignToAxis`** (input_control) . . . . . string ⇝ *string*
Determines whether the cropped image part is aligned with the coordinate axes.
Default: 'true'
List of values: `AlignToAxis` ∈ {'true', 'false'}

▷ **`Interpolation`** (input_control) . . . . . string ⇝ *string*
Interpolation method.
Default: 'constant'
List of values: `Interpolation` ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'constant', 'weighted'}

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.

Possible Successors

disp_image

Alternatives

crop_domain, crop_rectangle1, crop_part, change_format, reduce_domain

See also

zoom_image_size, zoom_image_factor

Module

Foundation

tile_channels (Image : TiledImage : NumColumns, TileOrder :)

Tile multiple images into a large image.

tile_channels tiles an image consisting of multiple channels into a large single-channel image. The input image Image contains *Num* images of the same size, which are stored in the individual channels. The output image TiledImage contains a single channel image, where the *Num* input channels have been tiled into NumColumns columns. In particular, this means that tile_channels cannot tile color images. For this purpose, tile_images can be used. The parameter TileOrder determines the order in which the images are copied into the output in the cases in which this is not already determined by NumColumns (i.e., if NumColumns != 1 and NumColumns != *Num*). If TileOrder = *'horizontal'* the images are copied in the horizontal direction, i.e., the second channel of Image will be to the right of the first channel. If TileOrder = *'vertical'* the images are copied in the vertical direction, i.e., the second channel of Image will be below the first channel. The domain of TiledImage is obtained by copying the domain of Image to the corresponding locations in the output image. If *Num* is not a multiple of NumColumns the output image will have undefined gray values in the lower right corner of the image. The output domain will reflect this.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Input image.

▷ **TiledImage** (output_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Tiled output image.

▷ **NumColumns** (input_control) ..... integer ⇝ *integer*

Number of columns to use for the output image.
Default: 1
Suggested values: NumColumns ∈ {1, 2, 3, 4, 5, 6, 7}
Restriction: NumColumns >= 1

▷ **TileOrder** (input_control) ..... string ⇝ *string*

Order of the input images in the output image.
Default: 'vertical'
List of values: TileOrder ∈ {'horizontal', 'vertical'}

Example

```
* Grab 5 single-channel images and stack them vertically.
gen_rectangle1 (Image, 0, 0, Height-1, Width-1)
for I := 1 to 5 by 1
    grab_image_async (ImageGrabbed, AcqHandle, -1)
    append_channel (Image, ImageGrabbed, Image)
endfor
tile_channels (Image, TiledImage, 1, 'vertical')
```

Result

tile_channels returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution.

If the input is empty the behavior can be set via `set_system('no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`append_channel`

Alternatives

`tile_images`, `tile_images_offset`

See also

`change_format`, `crop_part`, `crop_rectangle1`

Module

Foundation

`tile_images` (Images : TiledImage : NumColumns, TileOrder :)

Tile multiple image objects into a large image.

`tile_images` tiles multiple input image objects, which must contain the same number of channels, into a large image. The input image object `Images` contains *Num* images, which may be of different size. The output image `TiledImage` contains as many channels as the input images. In the output image the *Num* input images have been tiled into `NumColumns` columns. Each tile has the same size, which is determined by the maximum width and height of all input images. If an input image is smaller than the tile size it is copied to the center of the respective tile. The parameter `TileOrder` determines the order in which the images are copied into the output in the cases in which this is not already determined by `NumColumns` (i.e., if `NumColumns` != 1 and `NumColumns` != *Num*). If `TileOrder` = *'horizontal'* the images are copied in the horizontal direction, i.e., the second image of `Images` will be to the right of the first image. If `TileOrder` = *'vertical'* the images are copied in the vertical direction, i.e., the second image of `Images` will be below the first image. The domain of `TiledImage` is obtained by copying the domains of `Images` to the corresponding locations in the output image. If *Num* is not a multiple of `NumColumns` the output image will have undefined gray values in the lower right corner of the image. The output domain will reflect this.

Parameters

▷ **`Images`** (input_object) ..... (multichannel-)image-array ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Input images.

▷ **`TiledImage`** (output_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Tiled output image.

▷ **`NumColumns`** (input_control) ..... integer ⇝ *integer*

Number of columns to use for the output image.
Default: 1
Suggested values: `NumColumns` ∈ {1, 2, 3, 4, 5, 6, 7}
Restriction: NumColumns >= 1

▷ **`TileOrder`** (input_control) ..... string ⇝ *string*

Order of the input images in the output image.
Default: 'vertical'
List of values: `TileOrder` ∈ {'horizontal', 'vertical'}

Example

```
* Grab 5 (multi-channel) images and stack them vertically.
gen_empty_obj (Images)
```

```
for I := 1 to 5 by 1
    grab_image_async (ImageGrabbed, AcqHandle, -1)
    concat_obj (Images, ImageGrabbed, Images)
endfor
tile_images (Images, TiledImage, 1, 'vertical')
```

Result

`tile_images` returns `2` (`H_MSG_TRUE`) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`append_channel`

Alternatives

`tile_channels`, `tile_images_offset`

See also

`change_format`, `crop_part`, `crop_rectangle1`

Module

Foundation

tile_images_offset (Images : TiledImage : OffsetRow, OffsetCol, Row1, Col1, Row2, Col2, Width, Height :)

Tile multiple image objects into a large image with explicit positioning information.

`tile_images_offset` tiles multiple input image objects, which must contain the same number of channels, into a large image. The input image object `Images` contains *Num* images, which may be of different size. The output image `TiledImage` contains as many channels as the input images. The size of the output image is determined by the parameters `Width` and `Height`. The position of the upper left corner of the input images in the output images is determined by the parameters `OffsetRow` and `OffsetCol`. Both parameters must contain exactly *Num* values. Optionally, each input image can be cropped to an arbitrary rectangle that is smaller than the input image. To do so, the parameters `Row1`, `Col1`, `Row2`, and `Col2` must be set accordingly. If any of these four parameters is set to *-1*, the corresponding input image is not cropped. In any case, all four parameters must contain *Num* values. If the input images are cropped the position parameters `OffsetRow` and `OffsetCol` refer to the upper left corner of the cropped image. If the input images overlap each other in the output image (while taking into account their respective domains), the image with the higher index in `Images` overwrites the image data of the image with the lower index. The domain of `TiledImage` is obtained by copying the domains of `Images` to the corresponding locations in the output image.

Attention

If the input images all have the same size and tile the output image exactly, the operator `tile_images` usually will be slightly faster.

Parameters

▷ **Images** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Input images.

▷ **TiledImage** (output_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Tiled output image.

▷ **OffsetRow** (input_control) ..... point.y(-array) ⇝ *integer*
Row coordinate of the upper left corner of the input images in the output image.
Default: 0
Suggested values: OffsetRow ∈ {0, 50, 100, 150, 200, 250}

▷ **OffsetCol** (input_control) ..... point.x(-array) ⇝ *integer*
Column coordinate of the upper left corner of the input images in the output image.
Default: 0
Suggested values: OffsetCol ∈ {0, 50, 100, 150, 200, 250}

▷ **Row1** (input_control) ..... rectangle.origin.y(-array) ⇝ *integer*
Row coordinate of the upper left corner of the copied part of the respective input image.
Default: -1
Suggested values: Row1 ∈ {-1, 0, 10, 20, 50, 100, 200, 300, 500}

▷ **Col1** (input_control) ..... rectangle.origin.x(-array) ⇝ *integer*
Column coordinate of the upper left corner of the copied part of the respective input image.
Default: -1
Suggested values: Col1 ∈ {-1, 0, 10, 20, 50, 100, 200, 300, 500}

▷ **Row2** (input_control) ..... rectangle.corner.y(-array) ⇝ *integer*
Row coordinate of the lower right corner of the copied part of the respective input image.
Default: -1
Suggested values: Row2 ∈ {-1, 0, 10, 20, 50, 100, 200, 300, 500}

▷ **Col2** (input_control) ..... rectangle.corner.x(-array) ⇝ *integer*
Column coordinate of the lower right corner of the copied part of the respective input image.
Default: -1
Suggested values: Col2 ∈ {-1, 0, 10, 20, 50, 100, 200, 300, 500}

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the output image.
Default: 512
Suggested values: Width ∈ {32, 64, 128, 256, 512, 768, 1024, 2048, 4096}

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the output image.
Default: 512
Suggested values: Height ∈ {32, 64, 128, 256, 512, 525, 1024, 2048, 4096}

Example

```
* Example 1
* Grab 2 (multi-channel) NTSC images, crop the bottom 5 lines off
* of each image, the right 5 columns off of the first image, and
* the left five lines off of the second image, and put the cropped
* images side-by-side.
gen_empty_obj (Images)
for I := 1 to 2 by 1
    grab_image_async (ImageGrabbed, AcqHandle, -1)
    concat_obj (Images, ImageGrabbed, Images)
endfor
tile_images_offset (Images, TiledImage, [0,635], [0,0], [0,0], \
                    [0,5], [474,474], [634,639], 635, 950)

* Example 2
* Enlarge image by 15 rows and columns on all sides
EnlargeColsBy := 15
EnlargeRowsBy := 15
get_image_pointer1 (Image, Pointer, Type, WidthImage, HeightImage)
tile_images_offset (Image, EnlargedImage, EnlargeRowsBy, EnlargeColsBy, \
                    -1, -1, -1, -1, WidthImage + EnlargeColsBy*2, \
                    HeightImage + EnlargeRowsBy*2)
```

Result

tile_images_offset returns 2 (H_MSG_TRUE) if all parameters are correct and no error oc-

curs during execution. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

append_channel

Alternatives

tile_channels, tile_images

See also

change_format, crop_part, crop_rectangle1

Module

Foundation

15.8 Manipulation

overpaint_gray (ImageDestination, ImageSource : : :)

Overpaint the gray values of an image.

overpaint_gray copies the gray values of the image given in ImageSource into the image in ImageDestination. Only the gray values of the domain of ImageSource are copied (see reduce_domain).

If you do not want to modify ImageDestination itself, you can use the operator paint_gray, which returns the result in a newly created image.

Attention

overpaint_gray modifies the content of an already existing image (ImageDestination). Besides, even other image objects may be affected: For example, if you created ImageDestination via copy_obj from another image object (or vice versa), overpaint_gray will also modify the image matrix of this other image object. Therefore, overpaint_gray should only be used to overpaint newly created image objects. Typical operators for this task are, e.g., gen_image_const (creates a new image with a specified size), gen_image_proto (creates an image with the size of a specified prototype image) or copy_image (creates an image as the copy of a specified image).

Parameters

▷ **ImageDestination** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / vector_field

Input image to be painted over.

▷ **ImageSource** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / vector_field

Input image containing the desired gray values.

Example

```
* Copy a circular part of the image 'monkey' into a new image (New1):

read_image(Image,'monkey')
gen_circle(Circle,200,200,150)
reduce_domain(Image,Circle,Mask)
* New image with black (0) background
gen_image_proto(Image,New1,0.0)
* Copy a part of the image 'monkey' into New1
overpaint_gray(New1,Mask)
```

Result

`overpaint_gray` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ImageDestination

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`read_image`, `gen_image_const`, `gen_image_proto`

Alternatives

`get_image_pointer1`, `paint_gray`, `set_grayval`, `copy_image`

See also

`paint_region`, `overpaint_region`

Module

Foundation

overpaint_region (Image, Region : : Grayval, Type :)

Overpaint regions in an image.

`overpaint_region` paints the regions given in `Region` with a constant gray value into the image given in `Image`. These gray values can either be specified for each channel once, valid for all regions, or for each region separately. To define the latter, group the channel gray values g of each region and concatenate them to a tuple according to the regions' order, e.g., for a three channel image:

$$[g(channel1,region1), g(channel2,region1), g(channel3,region1), g(channel1,region2), \ldots]$$

If the input image is of type `direction`, gray values that are not in the value range that is valid for `direction` images are set to the value 255 to mark them as invalid. The parameter `Type` determines whether the region should be painted filled (*'fill'*) or whether only its boundary should be painted (*'margin'*).

If you do not want to modify `Image` itself, you can use the operator `paint_region`, which returns the result in a newly created image.

Attention

`overpaint_region` modifies the content of an already existing image (`Image`). Besides, even other image objects may be affected: For example, if you created `Image` via `copy_obj` from another image object (or vice versa), `overpaint_region` will also modify the image matrix of this other image object. Therefore, `overpaint_region` should only be used to overpaint newly created image objects. Typical operators for this task are, e.g., `gen_image_const` (creates a new image with a specified size), `gen_image_proto` (creates an image with the size of a specified prototype image) or `copy_image` (creates an image as the copy of a specified image).

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex

Image in which the regions are to be painted.

▷ **Region** (input_object) ..... region(-array) ⇝ *object*

Regions to be painted into the input image.

▷ **Grayval** (input_control) ..... number(-array) ⇝ *real / integer*
Desired gray values of the regions.
Default: 255.0
Suggested values: Grayval ∈ {0.0, 1.0, 2.0, 5.0, 10.0, 16.0, 32.0, 64.0, 128.0, 253.0, 254.0, 255.0}

▷ **Type** (input_control) ..... string ⇝ *string*
Paint regions filled or as boundaries.
Default: 'fill'
List of values: Type ∈ {'fill', 'margin'}

Example

```
* Paint a rectangle into a new image (New1)

gen_rectangle1(Rectangle,100.0,100.0,300.0,300.0)
* generate a black image
gen_image_const(New1,'byte', 768, 576)
* paint a white rectangle
overpaint_region(New1,Rectangle,255.0,'fill')
```

Result

overpaint_region returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- Image

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

read_image, gen_image_const, gen_image_proto, reduce_domain

Alternatives

set_grayval, paint_region, paint_xld

See also

reduce_domain, set_draw, paint_gray, overpaint_gray, gen_image_const

Module

Foundation

paint_gray (ImageSource, ImageDestination : MixedImage : :)

Paint the gray values of an image into another image.

paint_gray paints the gray values of the image given in ImageSource into the image in ImageDestination and returns the resulting image in MixedImage. Only the gray values of the domain of ImageSource are copied (see reduce_domain).

As an alternative to paint_gray, you can use the operator overpaint_gray, which directly paints the gray values into ImageDestination.

Parameters

▷ **ImageSource** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / vector_field

Input image containing the desired gray values.

▷ **ImageDestination** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / vector_field

Input image to be painted over.

▷ **MixedImage** (output_object) ..... image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / vector_field

Result image.

Example

```
* Copy a circular part of the image 'monkey' into the image 'fabrik':

read_image(Image,'monkey')
gen_circle(Circle,200,200,150)
reduce_domain(Image,Circle,Mask)
read_image(Image2,'fabrik')
* Copy a part of the image 'monkey' into 'fabrik'
paint_gray(Mask,Image2,MixedImage)
```

Result

`paint_gray` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_image`, `gen_image_const`, `gen_image_proto`

Alternatives

`get_image_pointer1`, `set_grayval`, `copy_image`, `overpaint_gray`

See also

`paint_region`, `overpaint_region`

Module

Foundation

paint_region (Region, Image : ImageResult : Grayval, Type :)

Paint regions into an image.

`paint_region` paints the regions given in `Region` with a constant gray value into the image given in `Image` and returns the result in `ImageResult`. These gray values can either be specified for each channel once, valid for all regions, or for each region separately. To define the latter, group the channel gray values g of each region and concatenate them to a tuple according to the regions' order, e.g., for a three channel image:

$$[g(channel1,region1), g(channel2,region1), g(channel3,region1), g(channel1,region2), \ldots]$$

If the input image is of type `direction`, gray values that are not in the value range that is valid for `direction` images are set to the value 255 to mark them as invalid. The parameter `Type` determines whether the region should be painted filled (*'fill'*) or whether only its boundary should be painted (*'margin'*).

As an alternative to paint_region, you can use the operator overpaint_region, which directly paints the regions into Image.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be painted into the input image.

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex
Image in which the regions are to be painted.

▷ **ImageResult** (output_object) ..... image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex
Image containing the result.

▷ **Grayval** (input_control) ..... number(-array) ⇝ *real* / integer
Desired gray values of the regions.
Default: 255.0
Suggested values: Grayval ∈ {0.0, 1.0, 2.0, 5.0, 10.0, 16.0, 32.0, 64.0, 128.0, 253.0, 254.0, 255.0}

▷ **Type** (input_control) ..... string ⇝ *string*
Paint regions filled or as boundaries.
Default: 'fill'
List of values: Type ∈ {'fill', 'margin'}

Example

```
* Paint a rectangle into the image 'monkey'

read_image(Image,'monkey')
gen_rectangle1(Rectangle,100.0,100.0,300.0,300.0)
* paint a white rectangle
paint_region(Rectangle,Image,ImageResult,255.0,'fill')
```

Result

paint_region returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, gen_image_const, gen_image_proto, reduce_domain

Alternatives

set_grayval, overpaint_region, paint_xld

See also

reduce_domain, paint_gray, overpaint_gray, set_draw, gen_image_const

Module

Foundation

paint_xld (XLD, Image : ImageResult : Grayval :)

Paint XLD objects into an image.

paint_xld paints the XLD objects XLD of type contour or polygon with the constant gray values Grayval into each channel of the background image given in Image and returns the result in ImageResult. Open contours

of XLD objects are closed and their enclosed regions are filled up. The rim of the subpixel XLD objects is painted onto the background image using anti-aliasing.

Filling up XLD-objects that have crossings or segments that are very close (lying in neighboring pixels) does not work correctly and leads to artifacts. The `set_system` parameter *'neighborhood'* defines whether diagonal neighbors are filled as well as orthogonal neighbors.

`Grayval` contains the gray values for painting the XLD objects. These gray values can either be specified for each channel once, valid for all XLD objects, or for each XLD object separately. To define the latter, group the channel gray values g of each XLD object and concatenate them to a tuple according to the order of the XLD objects, e.g., for a three channel image:

$$[g(channel1,xld1), g(channel2,xld1), g(channel3,xld1), g(channel1,xld2), \ldots]$$

If the input image is of type `direction`, gray values that are not in the value range that is valid for `direction` images are set to the value 255 to mark them as invalid.

Parameters

▷ **XLD** (input_object) . . . . . xld(-array) ⇝ *object*
XLD objects to be painted into the input image.

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex
Image in which the xld objects are to be painted.

▷ **ImageResult** (output_object) . . . . . image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex
Image containing the result.

▷ **Grayval** (input_control) . . . . . number(-array) ⇝ *real* / integer
Desired gray value of the xld object.
Default: 255.0
Suggested values: `Grayval` ∈ {0.0, 1.0, 2.0, 5.0, 10.0, 16.0, 32.0, 64.0, 128.0, 253.0, 254.0, 255.0}

Example

```
* Paint colored xld objects into a gray image

* read and copy image to generate a three channel image
read_image(Image1,'green-dot')
copy_image(Image1,Image2)
copy_image(Image1,Image3)
compose3(Image1,Image2,Image3,Image)
* extract subpixel border
threshold_sub_pix(Image1,Border,128)
* select the circle and the arrows
select_obj(Border,circle,14)
select_obj(Border,arrows,16)
concat_obj(circle,arrows,green_dot)
* paint a green circle and white arrows (to paint all
* objects e.g., blue, pass [0,0,255] tuple for GrayVal)
paint_xld(green_dot,Image,ImageResult,[0,255,0,255,255,255])
```

Result

`paint_xld` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system(::'no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image, gen_image_const, gen_image_proto, gen_contour_polygon_xld, threshold_sub_pix

Alternatives

set_grayval, paint_gray, paint_region

See also

gen_image_const

Module

Foundation

set_grayval (Image : : Row, Column, Grayval :)

Set single gray values in an image.

set_grayval sets the gray values of the input image Image at the positions (Row,Column) to the values specified by Grayval. If Image is a multi-channel image, you can either set a single gray value for a pixel and thus set it for all available channels, or you can set the gray values g for all channels individually by concatenating them within Grayval. For example, for a three-channel image three gray values (according to the order of the channels) are specified for each pixel:

$$[g(\mathit{channel0},\mathit{pixel0}), g(\mathit{channel1},\mathit{pixel0}), g(\mathit{channel2},\mathit{pixel0}), g(\mathit{channel0},\mathit{pixel1}), \ldots]$$

Please note that for complex or vector field images, two gray values per pixel must be specified (per channel).

If the image is of type direction, gray values that are not in the value range that is valid for direction images are set to the value 255 to mark them as invalid.

Attention

The operator set_grayval produces quite some overhead. Typically, it is used to set single gray values of an image. It is not suitable for programming image processing operations such as filters. In this case it is more useful to use the operator get_image_pointer1 and to directly use the C or C++ interface for integrating own procedures.

Note also that set_grayval modifies the content of an already existing image (Image). Besides, even other image objects may be affected: For example, if you created Image via copy_obj from another image object, set_grayval will also modify the image matrix of this other image object. Therefore, set_grayval should only be used to overpaint newly created image objects.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field

Image to be modified.

▷ **Row** (input_control) ..... point.y(-array) ⇝ *integer*

Row coordinates of the pixels to be modified.
Default: 0
Suggested values: Row ∈ {0, 10, 50, 127, 255, 511}
Value range: 0 ≤ Row
Restriction: 0 <= Row && Row < height(Image)

▷ **Column** (input_control) ..... point.x(-array) ⇝ *integer*

Column coordinates of the pixels to be modified.
Default: 0
Suggested values: Column ∈ {0, 10, 50, 127, 255, 511}
Value range: 0 ≤ Column
Restriction: 0 <= Column && Column < width(Image)

▷ **Grayval** (input_control) ..... grayval(-array) ⇝ *real* / integer

Gray values to be used.
Default: 255.0
Suggested values: Grayval ∈ {0.0, 1.0, 10.0, 128.0, 255.0}

Result

`set_grayval` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- Image

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`read_image`, `get_image_pointer1`, `gen_image_proto`, `gen_image1`

Alternatives

`get_image_pointer1`, `paint_gray`, `paint_region`

See also

`get_grayval`, `gen_image_const`, `gen_image1`, `gen_image_proto`

Module

Foundation

15.9 Type Conversion

`complex_to_real` (ImageComplex : ImageReal, ImageImaginary : :)

Convert a complex image into two real images.

`complex_to_real` converts a complex image `ImageComplex` into two real images `ImageReal` and `ImageImaginary`, which contain the real and imaginary part of the complex image.

Parameters

- ▷ **`ImageComplex`** (input_object) ..... singlechannelimage(-array) ⇝ *object :* complex
  Complex image.
- ▷ **`ImageReal`** (output_object) ..... image(-array) ⇝ *object :* real
  Real part.
- ▷ **`ImageImaginary`** (output_object) ..... image(-array) ⇝ *object :* real
  Imaginary part.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

See also

`real_to_complex`

Module

Foundation

convert_image_type (Image : ImageConverted : NewType :)

Convert the type of an image.

convert_image_type converts images of an arbitrary type into an arbitrary new image type. If the conversion is done from a larger to a smaller gray value range (e.g., from 'int4' to 'byte'), too large or too small values are simply "clipped". If the result images are of type 'direction', gray values that are not in the value range that is valid for 'direction' images are set to the value 255 to mark them as invalid. It is therefore advisable to adapt the range of gray values by calling scale_image before calling this operator. For images of type complex, only the real part is converted. The imaginary part is ignored. This facilitates an efficient conversion of images that have been transformed back from the frequency domain. Such images always have an imaginary part of 0.

Attention

The conversion to the 'int8' image format is only available on 64 bit systems! If the source and destination image type are identical, no new image matrix is allocated.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex

Image whose image type is to be changed.

▷ **ImageConverted** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex

Converted image.

▷ **NewType** (input_control) ..... string ⇝ *string*

Desired image type (i.e., type of the gray values).
Default: 'byte'
List of values: NewType ∈ {'int1', 'int2', 'uint2', 'int4', 'int8', 'byte', 'real', 'direction', 'cyclic', 'complex'}

Result

convert_image_type returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

scale_image

See also

scale_image, abs_image

Module

Foundation

real_to_complex (ImageReal, ImageImaginary : ImageComplex : :)

Convert two real images into a complex image.

real_to_complex converts two real images ImageReal and ImageImaginary, which contain the real and imaginary part of a complex image, into a complex image ImageComplex.

Parameters

▷ **ImageReal** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
Real part.

▷ **ImageImaginary** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
Imaginary part.

▷ **ImageComplex** (output_object) . . . . . image(-array) ⇝ *object :* complex
Complex image.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

See also

`complex_to_real`

Module

Foundation

real_to_vector_field (Row, Col : VectorField : Type :)

Convert two real-valued images into a vector field image.

`real_to_vector_field` converts two real-valued images `Row` and `Col` into a vector field image `VectorField`. The input images contain the vector components in the row and column direction, respectively. The semantic type of `VectorField` is set with `Type`.

Parameters

▷ **Row** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
Vector component in the row direction.

▷ **Col** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* real
Vector component in the column direction.

▷ **VectorField** (output_object) . . . . . image(-array) ⇝ *object :* vector_field
Displacement vector field.

▷ **Type** (input_control) . . . . . string ⇝ *string*
Semantic kind of the vector field.
Default: 'vector_field_relative'
List of values: `Type` ∈ {'vector_field_relative', 'vector_field_absolute'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`vector_field_to_real`

Module

Foundation

vector_field_to_real (VectorField : Row, Col : :)

Convert a vector field image into two real-valued images.

vector_field_to_real converts the vector field image VectorField into two real-valued images Row and Col. The output images contain the vector components in the row and column direction, respectively.

Parameters

▷ **VectorField** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* vector_field
Vector field.

▷ **Row** (output_object) . . . . . image(-array) ⇝ *object :* real
Vector component in the row direction.

▷ **Col** (output_object) . . . . . image(-array) ⇝ *object :* real
Vector component in the column direction.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

optical_flow_mg

See also

optical_flow_mg

Module

Foundation

Chapter 16

Inspection

16.1 Bead Inspection

apply_bead_inspection_model (Image : LeftContour, RightContour, ErrorSegment : BeadInspectionModel : ErrorType)

Inspect beads in an image, as defined by the bead inspection model.

The operator apply_bead_inspection_model applies the bead inspection model BeadInspectionModel to the input Image. The output parameter ErrorSegment indicates the positions of detected errors. The parameter ErrorType indicates the type of the detected errors for each segment. For more details on the possible errors, refer to the parameters' description and figures in create_bead_inspection_model. The value of ErrorType can be:

'no bead': No bead could be detected at this position.

'incorrect position': The center of the bead found at this position is further away from the contour than allowed by PositionTolerance.

'too thin': The bead is thinner than allowed by TargetThickness and ThicknessTolerance.

'too thick': The bead is thicker than allowed by TargetThickness and ThicknessTolerance.

Values of the generic parameters set through create_bead_inspection_model or set_bead_inspection_param highly affect the inspection results of the same model. For more details on those parameters, please refer to create_bead_inspection_model.

Note that the operator ignores the image domain and applies the model to the full image

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
  Image to apply bead inspection on.
- ▷ **LeftContour** (output_object) ..... xld-array ⇝ *object*
  The detected left contour of the beads.
- ▷ **RightContour** (output_object) ..... xld-array ⇝ *object*
  The detected right contour of the beads.
- ▷ **ErrorSegment** (output_object) ..... xld-array ⇝ *object*
  Detected error segments
- ▷ **BeadInspectionModel** (input_control) ..... bead_inspection_model ⇝ *handle*
  Handle of the bead inspection model to be used.
- ▷ **ErrorType** (output_control) ..... string-array ⇝ *string*
  Types of detected errors.

Result

The operator apply_bead_inspection_model returns the value 2 (H_MSG_TRUE) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- BeadInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_bead_inspection_model`, `set_bead_inspection_param`

Possible Successors

`set_bead_inspection_param`, `clear_bead_inspection_model`

See also

`get_bead_inspection_param`, `create_bead_inspection_model`, `set_bead_inspection_param`

Module

2D Metrology

clear_bead_inspection_model (: : BeadInspectionModel :)

Delete the bead inspection model and free the allocated memory.

The operator `clear_bead_inspection_model` deletes a bead inspection model that was created by `create_bead_inspection_model`. All memory used by the model is freed. The handle of the model is passed in `BeadInspectionModel`. It is invalid after the operator call.

Parameters

▷ **BeadInspectionModel** (input_control) ..... bead_inspection_model ⇝ *handle*
Handle of the bead inspection model.

Result

The operator `clear_bead_inspection_model` returns the value `2` (`H_MSG_TRUE`) if a valid handle is passed and the referred bead inspection model can be freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- BeadInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_bead_inspection_model`

See also

`create_bead_inspection_model`

Module

2D Metrology

create_bead_inspection_model (BeadContour : : TargetThickness, ThicknessTolerance, PositionTolerance, Polarity, GenParamName, GenParamValue : BeadInspectionModel)

Create a model to inspect beads or adhesive in images.

create_bead_inspection_model creates a model to inspect beads or adhesive in images.

The basic principle of bead inspection

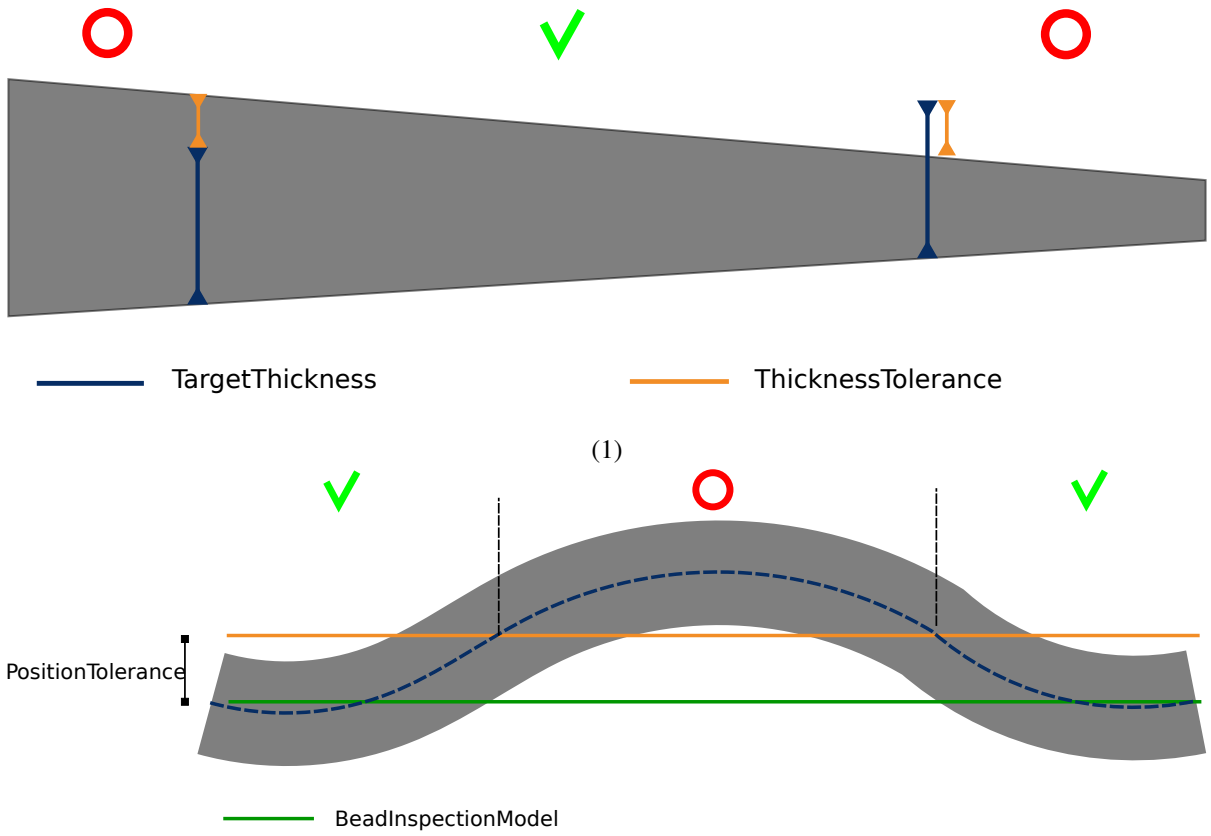
The bead inspection operators enable the user to define a reference contour that indicates the position and shape of a bead or adhesive on a certain product. Images of such products can then be inspected to verify that a bead or adhesive exists at the positions defined by the contour and that it is of acceptable thickness. The images must already be aligned with the given contour.

The operator create_bead_inspection_model returns a handle to the bead inspection model in BeadInspectionModel, which can later be used to perform other operations related to bead inspection. These operations include the modification of the bead inspection model parameters and performing the inspection on input images as defined by the model.

Providing the model contour

In order to inspect beads, a contour serving as the model for the bead must be provided in BeadContour. Such a contour can be generated, e.g., by using the operators gen_contour_polygon_xld or gen_contour_nurbs_xld. The contour must be defined to be aligned to the beads in the input images. To achieve this, the user can draw the XLD directly on a reference image in the graphics window (with draw_polygon or draw_nurbs). To inspect beads consisting of multiple parts, multiple models must be created.

Mandatory parameters



The mandatory parameters for a bead inspection model

In order to inspect beads, the following parameters must be set:

TargetThickness: defines the optimal bead thickness in pixels.

ThicknessTolerance: defines the tolerance to the bead's thickness in pixels. The bead is accepted if its thickness is within the range:
[TargetThickness-ThicknessTolerance, TargetThickness+ThicknessTolerance]

PositionTolerance: defines the tolerance to the bead's position in pixels. The 'bead position' is the center of the detected bead. The bead is accepted if the distance between its center and the contour's position is less than the PositionTolerance.

Polarity: defines the polarity of the bead in the images to inspect. *'light'* if the bead is lighter than its background, *'dark'* otherwise.

Edge extraction parameters

Bead inspection relies on edge extraction to identify the edges of beads. Accordingly, correct initialization of the measure_pos parameters helps improve bead inspection results.

The following parameters can be set for GenParamName.

'sigma': Sets the standard deviation of the Gaussian smoothing kernel. The value of *'sigma'* depends on the quality of the images to be inspected, namely on the amount of noise.
Default: *2.0*

'threshold': Sets the minimum edge amplitude to be extracted. Setting a higher value for *'threshold'* helps to avoid the extraction of noise edges and select salient edges. *'threshold'* cannot be less than *1*.
Default: *30*

The Measure Assistant can be used to find suitable values for these two parameters.

Parameters

▷ **BeadContour** (input_object) ..... xld ⇝ *object*
XLD contour specifying the expected bead's shape and position.

▷ **TargetThickness** (input_control) ..... integer ⇝ *integer* / real
Optimal bead thickness.
Default: 50
Value range: 5 ≤ TargetThickness

▷ **ThicknessTolerance** (input_control) ..... integer ⇝ *integer* / real
Tolerance of bead's thickness with respect to TargetThickness.
Default: 15
Value range: 0 ≤ ThicknessTolerance

▷ **PositionTolerance** (input_control) ..... integer ⇝ *integer* / real
Tolerance of the bead's center position.
Default: 15
Value range: 0 ≤ PositionTolerance

▷ **Polarity** (input_control) ..... string ⇝ *string*
The bead's polarity.
Default: 'light'
List of values: Polarity ∈ {'light', 'dark'}

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Names of the generic parameters that can be adjusted for the bead inspection model.
Default: []
List of values: GenParamName ∈ {'sigma', 'threshold'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *integer* / real / string
Values of the generic parameters that can be adjusted for the bead inspection model.
Default: []
Suggested values: GenParamValue ∈ {0.6, 1.0, 4.0, 5.0, 10.0, 50.0}

▷ **BeadInspectionModel** (output_control) ..... bead_inspection_model ⇝ *handle*
Handle for using and accessing the bead inspection model.

Example

```
* Read the image of the bead to be inspected.
read_image (Image, 'bead/adhesive_bead_01')
```

```
* Define the reference path of the adhesive beads.
gen_contour_nurbs_xld (ContourRef, \
                       [610.974,533.443,461.763,393.009,330.106,287.683, \
                        270.129,265.74,265.74,300.849,331.569,376.917, \
                        438.357,489.557,539.294], \
                       [418.581,424.27,439.441,473.574,526.67,574.078, \
                        644.241,708.715,765.604,818.7,866.107,915.411, \
                        966.611,998.848,993.159], \
                       'auto', \
                       [15,15,15,15,15,15,15,15,15,15,15,15,15,15,15], \
                       3, 1, 5)
*
* Create the bead inspection model, identifying the bead's parameters
create_bead_inspection_model (ContourRef, 14, 7, 30, 'dark', [], [], \
                              BeadInspectionModel)
*
* Apply the bead inspection model to the image
apply_bead_inspection_model (Image, LeftContour, RightContour, \
                             ErrorSegment, BeadInspectionModel, \
                             ErrorType)
```

Result

The operator `create_bead_inspection_model` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.This operator modifies the state of the following input parameter:

- BeadInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`gen_contour_nurbs_xld`, `gen_contour_polygon_xld`

Possible Successors

`apply_bead_inspection_model`, `set_bead_inspection_param`

See also

`clear_bead_inspection_model`

Module

2D Metrology

get_bead_inspection_param (: : BeadInspectionModel, GenParamName : GenParamValue)

Get the value of a parameter in a specific bead inspection model.

The operator `get_bead_inspection_param` is used to query the values of the different parameters of a bead inspection model. The names of the desired parameters are passed in the parameter `GenParamName`, the corresponding values are returned in `GenParamValue`. All these parameters can be set and changed at any time with the operator `set_bead_inspection_param`. Multiple parameters can be queried with a single call to `get_bead_inspection_param`.

The parameters that can be queried are *'target_thickness'*, *'thickness_tolerance'*, *'position_tolerance'*, *'polarity'*, *'sigma'* and *'threshold'*. Only the model's contour cannot be queried. Refer to `set_bead_inspection_param` and `create_bead_inspection_model` for a detailed description of all supported parameters.

Parameters

▷ **`BeadInspectionModel`** (input_control) . . . . . bead_inspection_model ⇝ *handle*
Handle of the bead inspection model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the model parameter that is queried.
Default: 'target_thickness'
List of values: `GenParamName` ∈ {'target_thickness', 'thickness_tolerance', 'position_tolerance', 'polarity', 'sigma', 'threshold'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.value(-array) ⇝ *integer* / real / string
Value of the queried model parameter.

Result

The operator `get_bead_inspection_param` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_bead_inspection_model`, `set_bead_inspection_param`

Possible Successors

`apply_bead_inspection_model`, `set_bead_inspection_param`

See also

`create_bead_inspection_model`, `set_bead_inspection_param`

Module

2D Metrology

`set_bead_inspection_param` (: : `BeadInspectionModel`, `GenParamName`, `GenParamValue` :)

Set parameters of the bead inspection model.

The operator `set_bead_inspection_param` is used to set parameters of a bead inspection model in order to adapt it to a particular inspection task. All parameters except for the model's contour can be set with this operator. A new bead inspection model should be used when a new contour is introduced. The current values of all parameters can be queried with the operator `get_bead_inspection_param`.

Parameters that can be set

The following parameters are essential and inspection model specific. It is vital for the inspection's success to identify accurate values for those parameters:

'target_thickness': sets the `TargetThickness`.

'thickness_tolerance': sets the `ThicknessTolerance`.

'position_tolerance': sets the `PositionTolerance`.

'polarity': sets the bead's `Polarity`.

'sigma'

'threshold'

For more details, refer to the parameters' description in `create_bead_inspection_model`.

Parameters

▷ **BeadInspectionModel** (input_control) . . . . . bead_inspection_model ⇝ *handle*
Handle of the bead inspection model.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the model parameter that shall be adjusted for the specified bead inspection model.
Default: 'target_thickness'
List of values: `GenParamName` ∈ {'target_thickness', 'thickness_tolerance', 'position_tolerance', 'polarity', 'sigma', 'threshold'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Value of the model parameter that shall be adjusted for the specified bead inspection model.
Default: 40
Suggested values: `GenParamValue` ∈ {40, 10, 15, 0.6, 1.0, 4.0, 5.0, 'light', 'dark'}

Result

The operator `set_bead_inspection_param` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- BeadInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_bead_inspection_model`

Possible Successors

`apply_bead_inspection_model`, `get_bead_inspection_param`

Alternatives

`create_bead_inspection_model`

See also

`create_bead_inspection_model`, `clear_bead_inspection_model`

Module

2D Metrology

16.2 OCV

close_ocv (: : OCVHandle :)

Clear an OCV tool.

`close_ocv` closes an open OCV tool and frees the memory. The OCV tool has been created using `create_ocv_proj` or `read_ocv`. The handle is after this call no longer valid.

Parameters

▷ **OCVHandle** (input_control) . . . . . ocv ⇝ *handle*
Handle of the OCV tool which has to be freed.

Example

```
read_ocv("ocv_file",&ocv_handle);
for (i=0; i<1000; i++)
{
  grab_image_async(&Image,fg_handle,-1);
  reduce_domain(Image,ROI,&Pattern);
  do_ocv_simple(Pattern,ocv_handle,"A",
                "true","true","false","true",10,
                &Quality);
}
close_ocv(ocv_handle);
```

Result

`close_ocv` returns `2` (`H_MSG_TRUE`), if the handle is valid. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCVHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`read_ocv`, `create_ocv_proj`

See also

`close_ocr`

Module

OCR/OCV

create_ocv_proj (: : PatternNames : OCVHandle)

Create a new OCV tool based on gray value projections.

`create_ocv_proj` creates a new OCV tool. This tool will be used to train good-patterns for the optical character verification. The training is done using the operator `traind_ocv_proj`. Thus `traind_ocv_proj` is normally called after `create_ocv_proj`.

The pattern comparison is based on the gray projections: For every training pattern the horizontal and vertical gray projections are calculated by summing up the gray values along the rows and columns inside the region of the pattern. This operation is applied to the training patterns and the test patterns. For the training patterns the result is stored inside the OCV tool to save runtime while comparing patterns. The OCV is done by comparing the corresponding projections. The Quality is the similarity of the projections.

Input for `create_ocv_proj` are the names of the patterns (`PatternNames`) which have to be trained. The number and the names can be chosen arbitrary. In most case only one pattern will be trained, thus only one name has to be specified. The names will be used when doing the OCV (`do_ocv_simple`). It is possible to specify more names than actually used. These might be trained later.

To close the OCV tool, i.e. to free the memory, the operator `close_ocv` is called.

Parameters

▷ **PatternNames** (input_control) . . . . . string(-array) ⇝ *string*
List of names for patterns to be trained.
Default: 'a'

▷ **OCVHandle** (output_control) . . . . . ocv ⇝ *handle*
Handle of the created OCV tool.

Example

```
create_ocv_proj("A",&ocv_handle);
draw_region(&ROI,window_handle);
reduce_domain(Image,ROI,&Sample);
traind_ocv_proj(Sample,ocv_handle,"A","single");
```

Result

`create_ocv_proj` returns `2` (`H_MSG_TRUE`), if the parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`traind_ocv_proj`, `write_ocv`, `close_ocv`

Alternatives

`read_ocv`

See also

`create_ocr_class_box`

Module

OCR/OCV

deserialize_ocv (: : SerializedItemHandle : OCVHandle)

Deserialize a serialized OCV tool.

`deserialize_ocv` deserializes an OCV tool, that was serialized by `serialize_ocv` (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The serialized OCV tool is defined by the handle `SerializedItemHandle`. The deserialized values are stored in an automatically created OCV tool with the handle `OCVHandle`. After deserializing the tool the training can be completed for those patterns which have not been trained so far. Otherwise, a pattern comparison can be applied directly by calling `do_ocv_simple`.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **OCVHandle** (output_control) ..... ocv ⇝ *handle*
Handle of the OCV tool.

Result

If the parameters are valid, the operator `deserialize_ocv` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_ocv

Possible Successors

do_ocv_simple, close_ocv

Module

OCR/OCV

do_ocv_simple (Pattern : : OCVHandle, PatternName, AdaptPos, AdaptSize, AdaptAngle, AdaptGray, Threshold : Quality)

Verification of a pattern using an OCV tool.

do_ocv_simple evaluates the pattern in (Pattern). Before the evaluation the good-pattern has to be trained by using the operator traind_ocv_proj. Both patterns should have roughly the same (relative) extent and shape. To specify which of the trained patterns is used as reference its name is specified in PatternName. The next four parameters influence the automatic adaption: AdaptPos and AdaptSize refer to the geometry of the pattern. AdaptPos specifies whether a shift of the position will be adapted automatically. AdaptSize is used to adapt to changes in the size of the pattern. AdaptAngle is not implemented. The parameter AdaptGray controls the adaption to changes of the gray values. This comprises additive and multiplicative changes of the intensity.

The parameter Threshold specifies the minimum difference of the gray values to be treated as an error. In this case the percentage of wrong pixels is returned. If the value is below 0 the sum of all errors normalized with respect to the size is returned.

The result of the operator is the Quality of the pattern with a value between 0 and 1. The value 1 corresponds to a pattern with no faults. The value 0 corresponds to a very big fault.

Parameters

- ▷ **Pattern** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Characters to be verified.
- ▷ **OCVHandle** (input_control) ..... ocv ⇝ *handle*
  Handle of the OCV tool.
- ▷ **PatternName** (input_control) ..... string(-array) ⇝ *string*
  Name of the character.
  Default: 'a'
- ▷ **AdaptPos** (input_control) ..... string ⇝ *string*
  Adaption to vertical and horizontal translation.
  Default: 'true'
  List of values: AdaptPos ∈ {'true', 'false'}
- ▷ **AdaptSize** (input_control) ..... string ⇝ *string*
  Adaption to vertical and horizontal scaling of the size.
  Default: 'true'
  List of values: AdaptSize ∈ {'true', 'false'}
- ▷ **AdaptAngle** (input_control) ..... string ⇝ *string*
  Adaption to changes of the orientation (not implemented).
  Default: 'false'
  List of values: AdaptAngle ∈ {'false'}
- ▷ **AdaptGray** (input_control) ..... string ⇝ *string*
  Adaption to additive and scaling gray value changes.
  Default: 'true'
  List of values: AdaptGray ∈ {'true', 'false'}
- ▷ **Threshold** (input_control) ..... number ⇝ *real*
  Minimum difference between objects.
  Default: 10
  Suggested values: Threshold ∈ {-1.0, 0.0, 1.0, 5.0, 10.0, 15.0, 20.0, 30.0, 40.0, 50.0, 60.0, 80.0, 100.0, 150.0}

▷ **Quality** (output_control) . . . . . real(-array) ⇝ *real*
Evaluation of the character.
Value range: 0.0 ≤ Quality ≤ 1.0

Result

do_ocv_simple returns 2 (H_MSG_TRUE), if the handle and the characters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

traind_ocr_class_box, trainf_ocr_class_box, read_ocv, threshold, connection, select_shape

Possible Successors

close_ocv

See also

create_ocv_proj

Module

OCR/OCV

read_ocv (: : FileName : OCVHandle)

Reading an OCV tool from file.

read_ocv reads an OCV tool from file. The tool will contain the same information that it contained when saving it with write_ocv. After reading the tool the training can be completed for those patterns which have not been trained so far. Otherwise a pattern comparison can be applied directly by calling do_ocv_simple.

As extension '.ocv' is used. If this extension is not given with the file name it will be added automatically.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Name of the file which has to be read.
Default: 'test_ocv'
File extension: .ocv

▷ **OCVHandle** (output_control) . . . . . ocv ⇝ *handle*
Handle of read OCV tool.

Example

```
read_ocv("ocv_file",&ocv_handle);
for (i=0; i<1000; i++)
{
  grab_image_async(&Image,fg_handle,-1);
  reduce_domain(Image,ROI,&Pattern);
  do_ocv_simple(Pattern,ocv_handle,"A",
                "true","true","false","true",10,
                &Quality);
}
close_ocv(ocv_handle);
```

Result

read_ocv returns 2 (H_MSG_TRUE), if the file is correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

write_ocv

Possible Successors

do_ocv_simple, close_ocv

See also

read_ocr

Module

OCR/OCV

serialize_ocv (: : OCVHandle : SerializedItemHandle)

Serialize an OCV tool.

serialize_ocv serializes the data of an OCV tool (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_ocv is converted to a serialized item. The OCV tool is defined by the handle OCVHandle. The serialized OCV tool is returned by the handle SerializedItemHandle and can be deserialized by deserialize_ocv.

Parameters

▷ **OCVHandle** (input_control) ..... ocv ⇝ *handle*
Handle of the OCV tool.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_ocv returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

traind_ocv_proj

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_ocv

Module

OCR/OCV

traind_ocv_proj (Pattern : : OCVHandle, Name, Mode :)

Training of an OCV tool.

traind_ocv_proj trains patterns for an OCV tool that has been created using the operators create_ocv_proj or read_ocv. For this training one or multiple patterns are provided to the system. Such a pattern consists of an image with a reduced domain (ROI) for the area of the pattern. Note that the pattern should

not only contain foreground pixels (e.g., dark pixels of a character) but also background pixels. This can be implemented e.g., by the smallest surrounding rectangle of the pattern. Without this context an evaluation of the pattern is not possible.

If more than one pattern has to be trained this can be achieved by multiple calls (one for each pattern) or by calling `traind_ocv_proj` once with all patterns and a tuple of the corresponding names. The result will be in both cases the same. However using multiple calls will normally result in a longer execution time than using one call with all patterns.

Parameters

▷ **`Pattern`** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
Pattern to be trained.

▷ **`OCVHandle`** (input_control) ..... ocv ⇝ *handle*
Handle of the OCV tool to be trained.

▷ **`Name`** (input_control) ..... string(-array) ⇝ *string*
Name(s) of the object(s) to analyze.
Default: 'a'

▷ **`Mode`** (input_control) ..... string ⇝ *string*
Mode for training (only one mode implemented).
Default: 'single'
List of values: `Mode` ∈ {'single'}

Example

```
create_ocv_proj("A",&ocv_handle);
draw_region(&ROI,window_handle);
reduce_domain(Image,ROI,&Sample);
traind_ocv_proj(Sample,ocv_handle,"A","single");
```

Result

`traind_ocv_proj` returns `2` (`H_MSG_TRUE`), if the handle and the training pattern(s) are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCVHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`write_ocr_trainf`, `create_ocv_proj`, `read_ocv`, `threshold`, `connection`, `select_shape`

Possible Successors

`close_ocv`

See also

`traind_ocr_class_box`

Module

OCR/OCV

`write_ocv` (: : `OCVHandle`, `FileName` :)

Saving an OCV tool to file.

write_ocv writes an OCV tool to file. This can be used to save the result of a training (traind_ocv_proj). The whole information contained in the OCV tool is stored in the file. The file can be reloaded afterwards using the operator read_ocv.

As file extension '.ocv' is used. If this extension is not given with the file name, it will be added automatically.

Parameters

▷ **OCVHandle** (input_control) . . . . . ocv ⇝ *handle*
Handle of the OCV tool to be written.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the file where the tool has to be saved.
Default: 'test_ocv'
File extension: .ocv

Result

write_ocv returns 2 (H_MSG_TRUE), if the data is correct and the file can be written. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

traind_ocv_proj

Possible Successors

close_ocv

See also

write_ocr

Module

OCR/OCV

16.3 Structured Light

This chapter describes the usage of structured light for surface inspection (deflectometry).

Concept of Structured Light

The basic concept behind structured light is to use a structured illumination, i.e. an illumination showing well known patterns. The way those patterns appear in the scene after hitting surfaces helps to further analyze or reconstruct the surfaces (see 3D Reconstruction / Structured Light).

Deflectometry is the procedure of analyzing the reflections of known patterns from specular or semi-specular surfaces. In such a setup, every pattern image must be shown by the display or monitor. It is then reflected by the specular surface under test, and a camera image of the reflection is acquired. Deformations of the pattern in the camera image are caused by the form of the specular surface which implies that defects on the specular surface can be detected.

In the following, the steps that are required to use structured light are described briefly.

Create a structured light model: In the first step, a structured light model is created with

- create_structured_light_model (ModelType='*deflectometry*')

or read with

- read_structured_light_model.

Set the model parameters: The different structured light model parameters can then be set with

- set_structured_light_model_param

or queried with

- get_structured_light_model_param.

The pattern parameters *'pattern_width'*, *'pattern_height'*, *'pattern_orientation'*, and *'pattern_type'* specify along with the stripe parameters *'min_stripe_width'* and *'single_stripe_width'* the specifications of the pattern images to be used to illuminate the surface. Finally, the *'persistence'* parameter can be enabled to debug intermediate results.

Generate the pattern images: The pattern images are to be generated with gen_structured_light_pattern after setting all relevant parameters. Please ensure that the output images are as needed in the particular setup.

Use the patterns to illuminate the surface and acquire the camera images: At this stage, the pattern images are shown on the display. The respective image of the illuminated surface is acquired by the camera for each pattern image.

Decode the acquired images: The acquired CameraImages can be decoded with decode_structured_light_pattern. Upon calling this operator, the correspondence images are created and stored in the model StructuredLightModel.

Get the results: The decoded *'correspondence_image'*, as well as other results can be queried with get_structured_light_object. For more details of the different objects that can be queried, please refer to the operator's documentation.

The *'defect_image'* can also be generated and queried with get_structured_light_object.

Further operators

The structured light model offers various other operators that help access and update the various parameters of the model.

The operator write_structured_light_model enables writing the structured light model to a file. Please note that previously generated pattern images are not written in this file. A structured light model file can be read using read_structured_light_model.

Furthermore, it is possible to serialize and deserialize the structured light model using serialize_structured_light_model and deserialize_structured_light_model.

Further Information

See also the "Solution Guide Basics" for further details.

clear_structured_light_model (: : StructuredLightModel :)

Clear a structured light model and free the allocated memory.

The operator clear_structured_light_model deletes a structured light model that was created by create_structured_light_model. All memory used by the model is freed. The handle of the model is passed in StructuredLightModel. It is invalid after the operator call.

For an explanation of the concept of structured light and its supported applications, see the introduction of chapter Inspection / Structured Light.

Parameters

▷ **StructuredLightModel** (input_control) ..... structured_light_model(-array) ⇝ *handle*
Handle of the structured light model.

Result

The operator clear_structured_light_model returns the value 2 (H_MSG_TRUE) if a valid handle is passed and the referred structured light model can be freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- StructuredLightModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

`create_structured_light_model`

Module

3D Metrology

create_structured_light_model (: : ModelType : StructuredLightModel)

Create a structured light model.

`create_structured_light_model` creates a new structured light model of type `ModelType`. Currently, the types *'deflectometry'* and *'3d_reconstruction'* are supported.

The parameters of the structured light model can be queried with `get_structured_light_model_param` and manipulated by calls to `set_structured_light_model_param`.

For an explanation of the concept of structured light and its supported applications, see the introduction of chapter Inspection / Structured Light.

Parameters

▷ **ModelType** (input_control) . . . . . string ⇝ *string*
The type of the created structured light model.
Default: 'deflectometry'
List of values: `ModelType` ∈ {'3d_reconstruction', 'deflectometry'}

▷ **StructuredLightModel** (output_control) . . . . . structured_light_model ⇝ *handle*
Handle for using and accessing the structured light model.

Example

```
* Create the model
create_structured_light_model ('deflectometry', StructuredLightModel)
* Generate the patterns to project
gen_structured_light_pattern (PatternImages, StructuredLightModel)
* Decode the camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
```

Result

The operator `create_structured_light_model` returns the value `2` (`H_MSG_TRUE`) if the structured light model can be allocated correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_structured_light_model_param, get_structured_light_model_param

Module

3D Metrology

decode_structured_light_pattern (
CameraImages : : StructuredLightModel :)

Decode the camera images acquired with a structured light setup.

decode_structured_light_pattern decodes the camera images CameraImages that have been previously acquired with a structured light setup. The correspondence images and other intermediate results that are created by the decoding process are stored in the model StructuredLightModel and can be accessed afterwards using the operator get_structured_light_object.

In the following, the decoding process is explained in detail:

As mentioned in gen_structured_light_pattern the first purpose is to find out whether a pixel is in a region illuminated by a light stripe or illuminated by a dark stripe. To simplify this decision process the normalization images are used and a locally varying threshold is determined that is able to cope with objects of varying reflectance or brightness and lighting conditions. During the decoding of the acquired camera images all Gray code images are then compared with the previously calculated threshold. A pixel within the image is classified as bright if its gray value is greater or equal this threshold.

Furthermore, the pattern region is segmented during the decoding process. The segmentation is controlled by the parameter *'min_gray_difference'* (see set_structured_light_model_param).

Assuming that n Gray code images have been processed, we get a n-bit binary code for each pixel. From this sequence the row and column coordinates up to $'min\_stripe\_width'/2$ of the monitor or projector can be derived.

If the StructuredLightModel is a hybrid system consisting not only of Gray code images but also of phase shift images (see gen_structured_light_pattern), the next step is to decode the latter ones. The result is a subpixel-precise correspondence image between the monitor or projector coordinates and the camera coordinates that contains the information which camera pixel observes which monitor or projector pixel.

If the *'pattern_type'* of the StructuredLightModel is set to *'single_stripe'*, the first step in the decoding process is to decide which single stripe shed its light on a camera pixel. The Gray code sequence and phase are then used to refine the position within the found single stripe.

In real world setups it may occur that the detected Gray code sequence of a pixel is wrong. This can then lead to values in the correspondence images which represent monitor or projector rows or columns larger than the monitor or projector width and height. To avoid these problems, the last step of the decoding process is to remove these values from the correspondence images.

Parameters

▷ **CameraImages** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2
Acquired camera images.

▷ **StructuredLightModel** (input_control) ..... structured_light_model ⇝ *handle*
Handle of the structured light model.

Example

```
* Create the model
create_structured_light_model ('deflectometry', StructuredLightModel)
* Set the size of the monitor
set_structured_light_model_param (StructuredLightModel, \
                                  'pattern_width', 1600)
set_structured_light_model_param (StructuredLightModel, \
                                  'pattern_height', 1200)
* Set the smallest width of the stripes in the pattern
set_structured_light_model_param (StructuredLightModel, \
                                  'min_stripe_width', 8)
* Generate the patterns to project
```

```
gen_structured_light_pattern (PatternImages, StructuredLightModel)
* Set the expected black/white contrast in the region of interest
set_structured_light_model_param (StructuredLightModel, \
                                  'min_gray_difference', 70)
* Decode the camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
* Get the computed correspondences and defects
get_structured_light_object (CorrespondenceImages, StructuredLightModel, \
                             'correspondence_image')
set_structured_light_model_param (StructuredLightModel, 'derivative_sigma', \
                                  Sigma)
get_structured_light_object (DefectImage, StructuredLightModel, \
                             'defect_image')
```

Result

The operator decode_structured_light_pattern returns the value 2 (H_MSG_TRUE) if the given parameters are valid. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- StructuredLightModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

gen_structured_light_pattern

Possible Successors

get_structured_light_object, reconstruct_surface_structured_light

See also

create_structured_light_model, set_structured_light_model_param

Module

3D Metrology

deserialize_structured_light_model (: : SerializedItemHandle : StructuredLightModel)

Deserialize a structured light model.

deserialize_structured_light_model deserializes a structured light model that was serialized by serialize_structured_light_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized structured light model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created structured light model with the handle StructuredLightModel.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **StructuredLightModel** (output_control) . . . . . structured_light_model ⇝ *handle*
Handle of the structured light model.

Example

```
* Deserialize the model
deserialize_structured_light_model (SerializedItemHandle, \
                                    StructuredLightModel)
* Get a previously decoded result
get_structured_light_object (CorrespondenceImages, StructuredLightModel, \
                             'correspondence_image')
* Decode new camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
* Get the decoded result
get_structured_light_object (CorrespondenceImagesNew, StructuredLightModel, \
                             'correspondence_image')
```

Result

The operator `deserialize_structured_light_model` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- StructuredLightModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_structured_light_model`

See also

`create_structured_light_model`, `write_structured_light_model`, `serialize_structured_light_model`

Module

3D Metrology

gen_structured_light_pattern (: PatternImages : StructuredLightModel :)

Generate the pattern images to be displayed in a structured light setup.

`gen_structured_light_pattern` generates the pattern images that need to be displayed or projected in a structured light setup. Several parameters such as the width and height of the images, the pattern type and the minimal stripe width need to be set in advance in the model `StructuredLightModel` using `set_structured_light_model_param`. Dependent on the set parameters of the `StructuredLightModel`, the number and the appearance of the pattern images vary.

In general, the pattern images can be divided into four groups of images:

- Normalization images
- Gray code images
- Phase shift images
- Single stripe images

General information about the image types

Normalization images:

Decoding of Gray code images relies upon being able to know whether a camera pixel observed a surface point illuminated by a white stripe or a dark stripe of the displayed or projected patterns. To allow a robust decoding even for large variations in surface reflectance or brightness, normalization images are used to simplify the process of identifying light and dark regions. The simplest but in most cases also sufficient approach is to generate one entirely black and one entirely white image. During the decoding of the acquired camera images (see `decode_structured_light_pattern`) all Gray code images are then compared with the mean of the dark and the bright image and a pixel within the image is classified as bright if its gray value is greater or equal the previously calculated mean.

Another approach is to use the Gray code images as explained in the next section and generate an inverted version of these images. A pixel is then classified as light, if its gray value in the first image is brighter than the gray value of the inverted version.

Gray code images:

Before performing any practical task such as detecting defects, the first step in any structured light measurement process is to find a mapping between the pixel plane of the camera and the pixel plane of the monitor or projector. This can be achieved by displaying a series of images that uniquely encode the rows and columns of the monitor or projector. Since Gray code images have some advantages compared to other pattern types, they are typically used for the encoding. An example of a series of vertical and horizontal Gray code images can be seen in the subsequent figure.

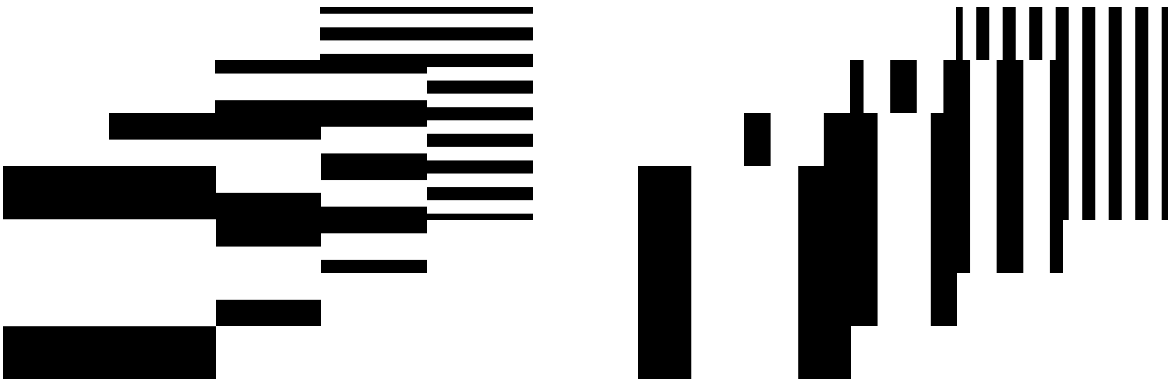


Image series of vertical and horizontal Gray code images as generated by `gen_structured_light_pattern`. The patterns are typically displayed on the monitor or projector from the coarsest pattern to the finest one while the stripe width is halved at each level.

Although using Gray code images is a very simple, yet suitable approach, it comes with certain limitations. One drawback is that often many Gray code images are required to achieve a spatial resolution sufficient for practical applications. However, with increasingly finer resolution of the stripes, a correct decoding becomes harder and harder because of blurring effects introduced by the optics involved in the measurement process. To overcome these limitations, the third group of pattern images (phase shift images) can be generated.

Phase shift images:

Phase shift images usually use periodic patterns such as sine or cosine waves to reconstruct the phase and in combination with Gray code images the correspondences between the monitor or projector and the camera coordinates. To allow a decoding that is robust despite variations in the surface reflectance or brightness and has a relaxed noise influence, four phase shift images per orientation (vertical or horizontal) are generated. Each image is a cosine wave with phase shifts of $k\pi/2$, where $k = 0, ..., 3$.

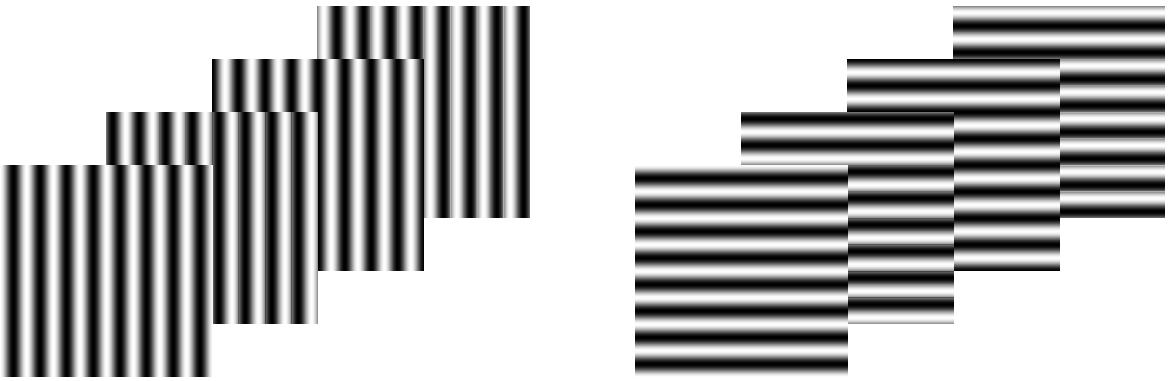


Image series of vertical and horizontal phase shift images as generated by `gen_structured_light_pattern`.

The main advantage of using phase shift images, is that in contrast to Gray code images subpixel-precise correspondences between the monitor or projector and the camera coordinates can be calculated. Since the images are periodic, the encoding is unique up to integer multiples of the period length. In order to achieve a unique encoding over the entire image, the combination with the Gray code image is necessary.

Single stripe images:

Single stripe images are dark with one bright stripe. The bright stripe is shifted over the images such that each monitor or projector pixel is contained in exactly one stripe. To decide which bright stripe is reflected in the area observed by a camera pixel, the brightest pixel value within the sequence of single stripe images is chosen during decoding. This allows a decoding that is robust despite variations in the surface reflectance.

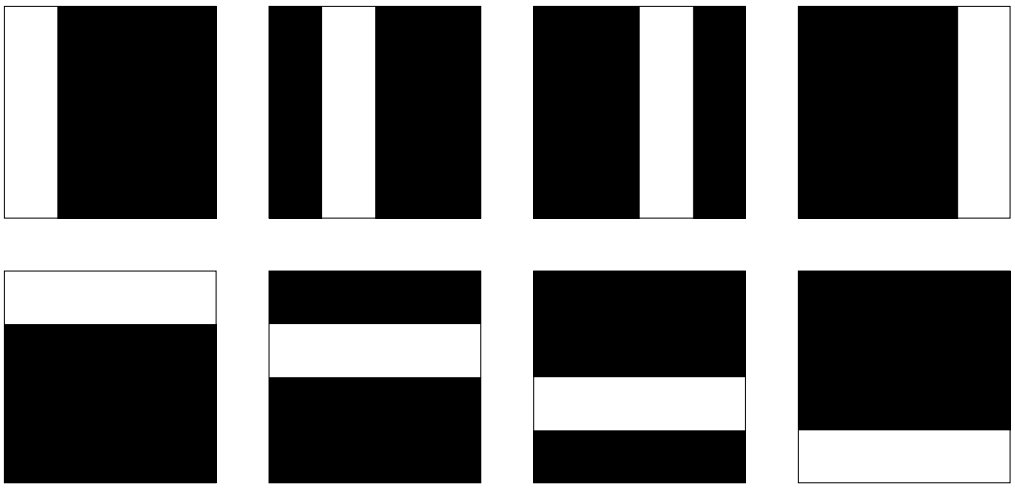


Image series of vertical and horizontal single stripe images as generated by `gen_structured_light_pattern` for *'pattern_type'* set to *'single_stripe'*.

The Gray code sequence and phase are then used to refine the position within the found single stripe. Thus, while additional single stripe images are generated, the number of created Gray code images will typically decrease slightly.

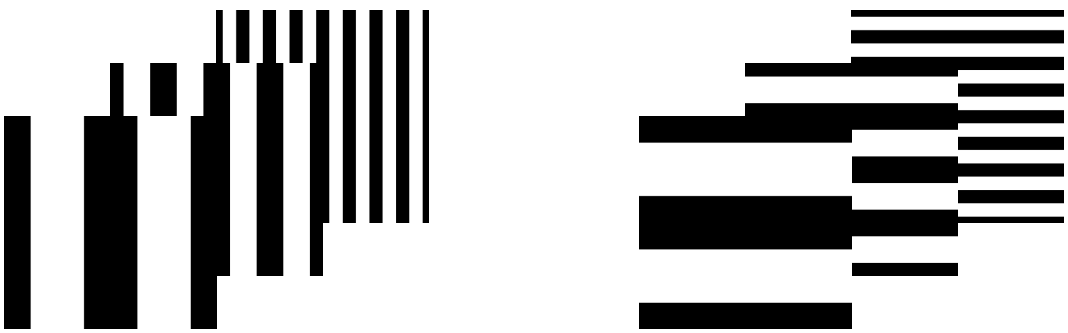


Image series of vertical and horizontal Gray code images as generated by gen_structured_light_pattern for *'pattern_type'* set to *'single_stripe'*.

Information about the generation of the image types

Generating the normalization images:

As mentioned above, HALCON offers two different kinds of normalization images. By default, the *'normalization'* method is set to *'global'* which results in the generation of an entirely black and an entirely white image. To change the method such that each Gray code is additionally inverted, set_structured_light_model_param has to be called using the method *'inverted_pattern'*.

The main advantage of using a global normalization is that the number of images is significantly smaller while simultaneously providing satisfying results. Only in cases of partially specular surfaces, the other method might be more robust and should be chosen if the global normalization leads to wrong decoding results.

Generating the Gray code images:

The size, the number and the appearance of the Gray code images is dependent on the following parameters (see set_structured_light_model_param): *'pattern_width'*, *'pattern_height'*, *'min_stripe_width'*, and *'pattern_orientation'*.

The first two parameters determine the size of the generated Gray code images and should usually be set to the size of the monitor or projector that is used to display the pattern images. The parameter *'min_stripe_width'* sets the width (in pixels) of the finest stripe of the pattern images that are generated. Additionally, all three parameters have a major influence on the number of created Gray code images.

Let $'max\_stripe\_width' = 2^{\lceil \log_2 ('pattern\_width') \rceil}$ be the maximum stripe width in vertical direction and $'max\_stripe\_width' = 2^{\lceil \log_2 ('pattern\_height') \rceil}$ be the maximum stripe width in horizontal direction. Then the number of Gray code images num_images with stripes oriented in one of the two directions (either horizontal or vertical) is given by

$$num\_images = \lceil \log_2 (2 \cdot max\_stripe\_width / min\_stripe\_width) \rceil$$

The parameter *'pattern_orientation'* sets the orientation of the Gray code (and phase shift) images that are generated. By default, images with vertical stripes as well as images with horizontal stripes are created. It is also possible to generate images with stripes having only one of the two directions. For structured light models of type 'deflectometry', this is not recommended, because changes in the surface can then only be detected in one direction. Solely, if changes on the surface should be ignored or if defects are to be expected in only one direction, this parameter should be changed using set_structured_light_model_param. In constrast, structured light models of type '3d_reconstruction' need both directions only during the calibration process (for further details please refer to the example program on structured light calibration). In the online process, setting *'pattern_orientation'* to *'vertical'* is recommended for speed, since fewer images are necessary than for *'both'* and reconstruct_surface_structured_light uses only the vertical correspondence image.

Generating the phase shift images:

Since phase shift images are suitable to overcome the practical limitations of Gray code images their generation is recommended and enabled by default.

In case that the accuracy using Gray code images only is sufficient, the creation of phase shift images can be deactivated using set_structured_light_model_param by setting the *'pattern_type'* to the value *'gray_code'*.

Generating the single stripe images:

In case that using Gray code and phase shift images only is not robust enough (this might happen, for example, on semi-specular surfaces), the creation of single stripe images can be activated using set_structured_light_model_param by setting the *'pattern_type'* to the value *'single_stripe'*.

The number and the appearance of the single stripe images depends on the parameter *'single_stripe_width'* which sets the width in pixels (see set_structured_light_model_param).

While additional single stripe images are generated, the number of created Gray code images will typically decrease slightly. To calculate the number of Gray code images, *'max_stripe_width'* is replaced by *'single_stripe_width'* in the formula above.

The use of these images is described in the introduction to the chapter Inspection / Structured Light.

Parameters

▷ **PatternImages** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Generated pattern images.

▷ **StructuredLightModel** (input_control) ..... structured_light_model ⇝ *handle*
Handle of the structured light model.

Example

```
* Create the model
create_structured_light_model ('deflectometry', StructuredLightModel)
* Set the size of the monitor
set_structured_light_model_param (StructuredLightModel, \
                                  'pattern_width', 1600)
set_structured_light_model_param (StructuredLightModel, \
                                  'pattern_height', 1200)
* Set the smallest width of the stripes in the pattern
set_structured_light_model_param (StructuredLightModel, \
                                  'min_stripe_width', 8)
* Generate the patterns to project
gen_structured_light_pattern (PatternImages, StructuredLightModel)
* Decode the camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
```

Result

The operator gen_structured_light_pattern returns the value 2 (H_MSG_TRUE) if the given parameters are valid. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- StructuredLightModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

set_structured_light_model_param

Possible Successors

decode_structured_light_pattern

See also

create_structured_light_model, get_structured_light_model_param, get_structured_light_object

Module

3D Metrology

get_structured_light_model_param (: : StructuredLightModel, GenParamName : GenParamValue)

Query parameters of a structured light model.

The operator get_structured_light_model_param is used to query the values of the different parameters of a structured light model (StructuredLightModel). The names of the desired parameters are passed

in the parameter GenParamName, the corresponding values are returned in GenParamValue. It is possible to query multiple parameters with a single call to get_structured_light_model_param.

All parameters which can be manipulated by set_structured_light_model_param can be queried. Please refer to the documentation of set_structured_light_model_param for an explanation of the individual parameters.

The following additional parameter can be queried:

'type': Type of the structured light model (currently either *'deflectometry'* or *'3d_reconstruction'*, as set in create_structured_light_model).

For an explanation of the concept of structured light and its supported applications, see the introduction of chapter Inspection / Structured Light.

Parameters

▷ **StructuredLightModel** (input_control) ..... structured_light_model ⇝ *handle*
Handle of the structured light model.

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Name of the queried model parameter.
Default: 'min_stripe_width'
List of values: GenParamName ∈ {'type', 'derivative_sigma', 'min_gray_difference', 'min_stripe_width', 'normalization', 'pattern_height', 'pattern_orientation', 'pattern_type', 'pattern_width', 'single_stripe_width', 'persistence', 'camera_setup_model'}

▷ **GenParamValue** (output_control) ..... attribute.value(-array) ⇝ *integer* / real / string
Value of the queried model parameter.

Example

```
* Create the model
create_structured_light_model ('deflectometry', StructuredLightModel)
* Get the default value
get_structured_light_model_param (StructuredLightModel, \
                                  'min_stripe_width', Default)
* Set the value
set_structured_light_model_param (StructuredLightModel, \
                                  'min_stripe_width', 64)
* Get the value
get_structured_light_model_param (StructuredLightModel, 'min_stripe_width', \
                                  MinStripeWidth)
* Generate the patterns to project
gen_structured_light_pattern (PatternImages, StructuredLightModel)
* Decode the camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
```

Result

The operator get_structured_light_model_param returns the value 2 (H_MSG_TRUE) if the given parameters are valid. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_structured_light_model

See also

set_structured_light_model_param

Module

3D Metrology

get_structured_light_object (: Object : StructuredLightModel, ObjectName :)

Get (intermediate) iconic results of a structured light model.

With the operator get_structured_light_object you can access and inspect (intermediate) iconic results of a structured light measurement setup that has already been decoded using decode_structured_light_pattern.

On the one hand, this is useful to query the results of the decoding process (*'correspondence_image'*) together with further processed results i.e. the defect image (*'defect_image'*). On the other hand, get_structured_light_object is also useful for troubleshooting the decoding process. Note that to collect the intermediate results, the *'persistence'* mode has to be enabled for the StructuredLightModel (see set_structured_light_model_param) *before* decoding the model.

Objects that can be accessed without persistence mode enabled:

'correspondence_image': Image that describes the correspondences between the monitor or projector and the camera. If the decoding has been performed in vertical and horizontal direction, both correspondence images are returned in an array. Otherwise, only the image of the appropriate direction is returned.

'pattern_region': Segmented region in which the camera images have been decoded. This region is identical to the domain of the correspondence images.

'defect_image' **(for structured light models of type** *'deflectometry'***):** Image that describes possible defects on the surface which reflects the displayed patterns towards the camera. Basically, high gray values in the defect image indicate that the gradients of the correspondence images differ significantly from gradients in the neighborhood. Note that this object is only available for structured light models of type *'deflectometry'*.

Objects that can be accessed only when in persistence mode:

If the decoding has been performed in vertical and horizontal direction, images of both directions are returned in an array. Otherwise, only the images of the appropriate direction are returned.

'binarized_image': Images of the Gray code sequence where white and black gray values indicate that the camera pixel observed a surface point illuminated by white or black regions of the displayed or projected patterns. If *'normalization'* was set to *'inverted_pattern'* using set_structured_light_model_param, this refers to the first image of an inverted pair.

'decoded_phase_shift_image': Decoded phase shift images containing the phase or angle corresponding to the displayed or projected phase images. One phase period covers stripes in the pattern of an extent *'min_stripe_width'*.

'decoded_single_stripe_image': Decoded single stripe image computed solely from the single stripe images. The correspondence image is a combination of this image with the Gray code and phase shift images, if *'pattern_type'* is set to *'single_stripe'*.

'gray_code_correspondence_image': Correspondence image computed solely from the Gray code images. If *'pattern_type'* is set to *'gray_code_and_phase_shift'*, the actual correspondence image is a combination of this image with the decoded phase shift images. If *'pattern_type'* is set to *'single_stripe'*, the actual correspondence image is a combination of this image with the decoded phase shift images and the decoded single stripe images.

Parameters

▷ **Object** (output_object) ..... (multichannel-)object(-array) ⇝ *object*
Iconic result.

▷ **StructuredLightModel** (input_control) ..... structured_light_model ⇝ *handle*
Handle of the structured light model.

▷ **ObjectName** (input_control) ..... string(-array) ⇝ *string*
Name of the iconic result to be returned.
Default: 'correspondence_image'
Suggested values: ObjectName ∈ {'correspondence_image', 'pattern_region', 'decoded_single_stripe_image', 'binarized_image', 'decoded_phase_shift_image', 'gray_code_correspondence_image', 'defect_image'}

Example

```
* Create the model
create_structured_light_model ('deflectometry', StructuredLightModel)
* Generate the patterns to project
gen_structured_light_pattern (PatternImages, StructuredLightModel)
* Decode the camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
* Get the computed correspondences and defects
get_structured_light_object (CorrespondenceImages, StructuredLightModel, \
                             'correspondence_image')
set_structured_light_model_param (StructuredLightModel, \
                                  'derivative_sigma', Sigma)
get_structured_light_object (DefectImage, StructuredLightModel, \
                             'defect_image')
```

Result

The operator `get_structured_light_object` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`decode_structured_light_pattern`

Possible Successors

`clear_structured_light_model`

See also

`gen_structured_light_pattern`

Module

3D Metrology

read_structured_light_model (
: : FileName : StructuredLightModel)

Read a structured light model from a file.

`read_structured_light_model` reads the structured light model `StructuredLightModel` that has been stored with `write_structured_light_model`. The default HALCON file extension for the structured light model is 'hslm'.

It can be helpful to write a model after setting all the optimal parameters for a particular setup. The model can be read later to easily regenerate pattern images. The objects already generated in the model before writing it can be also revisited later after reading the model.

For an explanation of the concept of structured light and its supported applications, see the introduction of chapter Inspection / Structured Light.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: `.hslm`

▷ **StructuredLightModel** (output_control) . . . . . structured_light_model ⇝ *handle*
Handle of the structured light model.

Example

```
* Read the model
read_structured_light_model ('ExampleModel.hslm', StructuredLightModel)
* Get a previously decoded result
get_structured_light_object (CorrespondenceImages, StructuredLightModel, \
                             'correspondence_image')
* Decode new camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
* Get the decoded result
get_structured_light_object (CorrespondenceImagesNew, StructuredLightModel, \
                             'correspondence_image')
```

Result

The operator `read_structured_light_model` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`write_structured_light_model`

See also

`create_structured_light_model`, `clear_structured_light_model`

Module

3D Metrology

reconstruct_surface_structured_light (: : StructuredLightModel : ObjectModel3D)

Reconstruct a surface from a decoded structured light setup.

`reconstruct_surface_structured_light` reconstructs a surface from a structured light setup. Prior to reconstruction, for every projected pattern image, a camera image of the projection on the surface is acquired. For an explanation of the concept of structured light and its supported applications, see the introduction of chapter Inspection / Structured Light. The reconstructed surface is stored in the handle `ObjectModel3D`.

Reconstruction is possible for non-specular surfaces, using a projector projecting light like an 'inverse camera'. In `create_structured_light_model`, the type of the model must have been set to *'3d_reconstruction'*.

The reconstruction process uses the vertical decoded correspondence between projector coordinates lighting the camera coordinates, as well as calibration information:

`decode_structured_light_pattern` must be called before `reconstruct_surface_structured_light`, storing the correspondence image between the projector coordinates and the camera coordinates in the model `StructuredLightModel`, i. e. for each camera pixel which projector pixel lighted (i.e. encoded) the observed point on the surface. `reconstruct_surface_structured_light` uses the vertical correspondence image, so the parameter *'pattern_orientation'* of `set_structured_light_model_param` must have been set to *'vertical'* or *'both'*. Note that in most cases *'vertical'* should be chosen for speed, since fewer images are necessary than for *'both'*.

For setting the calibration information, please refer to the parameter *'camera_setup_model'* of `set_structured_light_model_param`.

The reconstruction is performed only within the domain of the correspondence image. Reducing the domain of the input images of decode_structured_light_pattern (e.g., with reduce_domain) can be used to obtain a reduced reconstructed surface, i. e. for shorter runtime or to reduce noise. The resulting ObjectModel3D contains a mapping, so the coordinate images can be obtained by object_model_3d_to_xyz using *'from_xyz_map'*. Pixels for which no reconstruction was possible are excluded from its domain.

Parameters

▷ **StructuredLightModel** (input_control) ..... structured_light_model ⇝ *handle*
Handle of the structured light model.

▷ **ObjectModel3D** (output_control) ..... object_model_3d ⇝ *handle*
Handle of the 3D object model.

Example

```
* Set the calibration information
set_structured_light_model_param (StructuredLightModel, \
                                  'camera_setup_model', CameraSetupModelID)
* Decode the camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
* Reconstruct the surface
reconstruct_surface_structured_light (StructuredLightModel, ObjectModel3D)
```

Result

The operator reconstruct_surface_structured_light returns the value 2 (H_MSG_TRUE) if the given parameters are valid. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

decode_structured_light_pattern

Possible Successors

object_model_3d_to_xyz

See also

create_structured_light_model, set_structured_light_model_param

Module

3D Metrology

serialize_structured_light_model (
: : StructuredLightModel : SerializedItemHandle)

Serialize a structured light model.

serialize_structured_light_model serializes a structured light model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The structured light model is defined by the handle StructuredLightModel. The serialized structured light model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_structured_light_model.

Parameters

▷ **StructuredLightModel** (input_control) ..... structured_light_model ⇝ *handle*
Handle of the structured light model.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Example

```
* Create the model
create_structured_light_model ('deflectometry', StructuredLightModel)
* Generate the patterns to project
gen_structured_light_pattern (PatternImages, StructuredLightModel)
* Decode the camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
* Serialize the model
serialize_structured_light_model (StructuredLightModel, \
                                  SerializedItemHandle)
```

Result

The operator `serialize_structured_light_model` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`clear_structured_light_model`, `fwrite_serialized_item`, `send_serialized_item`, `deserialize_structured_light_model`

See also

`create_structured_light_model`, `read_structured_light_model`, `deserialize_structured_light_model`

Module

3D Metrology

set_structured_light_model_param (: : StructuredLightModel, GenParamName, GenParamValue :)

Set parameters of a structured light model.

The operator `set_structured_light_model_param` is used to manipulate the parameters of a structured light model `StructuredLightModel`. The current parameter settings can be queried with the operator `get_structured_light_model_param`.

For an explanation of the concept of structured light and its supported applications, see the introduction of chapter Inspection / Structured Light.

General parameters:

'persistence': Enables (`GenParamValue`=*'true'*) or disables (`GenParamValue`=*'false'*) the *'persistence'* mode of the structured light model. When in persistence mode, the model stores intermediate results of the decoding, which can be inspected later by `get_structured_light_object`.

Note that the model might need significant memory space in this mode. Therefore, we recommend to enable this mode only during the setup and for debugging a structured light model setup.

If *'persistence'* is changed, all results of the model are cleared.

Values: 'true', 'false', 1, 0

Default: 'false'

Parameters which influence the appearance of the generated pattern images:

'pattern_width': Sets the width of the pattern images that are generated while calling gen_structured_light_pattern. Usually, the value should be set to the width of the monitor or projector that is used to display the pattern images.

The value of *'pattern_width'* has to be larger than the value *'min_stripe_width'*/2. In case that *'pattern_type'* is *'single_stripe'*, it also has to be larger than *'single_stripe_width'*/2. In case that a *'camera_setup_model'* is set in the structured light model, the value of *'pattern_width'* has to match the width of the projector parameters stored in the camera setup model (at Index *1*).

If *'pattern_width'* is changed, the model is not decodable anymore and all possibly available results are cleared. Therefore, gen_structured_light_pattern has to be called before decoding the model using decode_structured_light_pattern.

Values: integer values larger or equal to 1

Default: 1024

'pattern_height': Sets the height of the pattern images that are generated while calling gen_structured_light_pattern. Usually, the value should be set to the height of the monitor or projector that is used to display the pattern images.

The value of *'pattern_height'* has to be larger than the value *'min_stripe_width'*/2. In case that *'pattern_type'* is *'single_stripe'*, it also has to be larger than *'single_stripe_width'*/2. In case that a *'camera_setup_model'* is set in the structured light model, the value of *'pattern_height'* has to match the height of the projector parameters stored in the camera setup model (at Index *1*).

If *'pattern_height'* is changed, the model is not decodable anymore and all possibly available results are cleared. Therefore, gen_structured_light_pattern has to be called before decoding the model using decode_structured_light_pattern.

Values: integer values larger or equal to 1

Default: 1024

'min_stripe_width': Sets the width (in pixels) of the finest stripe of the pattern images that are generated while calling gen_structured_light_pattern.

The value has to be a power of two. Furthermore, it has to be smaller or equal the minimum of the parameters *'pattern_width'* and *'pattern_height'* after rounding both up to their next power of two. In case that *'pattern_type'* is *'single_stripe'*, it also has to be smaller or equal the *'single_stripe_width'*.

If *'min_stripe_width'* is changed, the model is not decodable anymore and all possibly available results are cleared. Therefore, gen_structured_light_pattern has to be called before decoding the model using decode_structured_light_pattern.

Values: integer values larger or equal to 4 and a power of 2

Default: 32

'normalization': Sets the normalization mode of the structured light model. If the parameter is set to *'global'*, gen_structured_light_pattern generates two additional pattern images composed of a black and a white image. These images are then used in the decoding process to determine the actual pattern region. If instead, the value *'inverted_pattern'* is chosen, each Gray code image is additionally inverted and the pattern region is segmented iteratively using the Gray code images and their inverted version.

If *'normalization'* is changed, the model is not decodable anymore and all possibly available results are cleared. Therefore, gen_structured_light_pattern has to be called before decoding the model using decode_structured_light_pattern.

Values: 'global', *'inverted_pattern'*

Default: 'global'

'pattern_orientation': Sets the orientation of the pattern images that are generated while calling gen_structured_light_pattern. If the parameter is set to *'both'*, pattern images with vertical stripes as well as pattern images with horizontal stripes are created. For *'vertical'* the pattern images consist of vertical stripes only, whereas for *'horizontal'* only horizontal stripes are generated.

If *'pattern_orientation'* is changed, the model is not decodable anymore and all possibly available results are cleared. Therefore, gen_structured_light_pattern has to be called before decoding the model using decode_structured_light_pattern.

Values: 'both', *'vertical'*, *'horizontal'*

Default: 'both'

'pattern_type': Sets the type of the pattern images that are generated while calling gen_structured_light_pattern.

'pattern_type'	Generated Images
'gray_code'	Gray code images
'gray_code_and_phase_shift'	Gray code images and phase shift images
'single_stripe'	Single stripe images, Gray code images and phase shift images

Usually, *'gray_code_and_phase_shift'* leads to more accurate results than *'gray_code'* because the combination between Gray code and phase images yields sub-pixel precise monitor or projector coordinates. *'single_stripe'* can increase robustness in case of only partially specular surfaces.

If *'pattern_type'* is changed, the model is not decodable anymore and all possibly available results are cleared. Therefore, gen_structured_light_pattern has to be called before decoding the model using decode_structured_light_pattern.

Values: 'gray_code_and_phase_shift', *'gray_code'*, *'single_stripe'*

Default: 'gray_code_and_phase_shift'

'single_stripe_width': Sets the width (in pixels) of the single stripe that is generated while calling gen_structured_light_pattern using the *'pattern_type' 'single_stripe'*.

The value has to be a power of two and has to be greater or equal *'min_stripe_width'*. Furthermore, it has to be smaller or equal the minimum of the parameters *'pattern_width'* and *'pattern_height'* after rounding both up to their next power of two.

Note that *'single_stripe_width'* can only be set in case that *'pattern_type'* is *'single_stripe'*. When *'pattern_type'* is changed to *'single_stripe'*, *'single_stripe_width'* is set to some power of two roughly halfway between the currently set *'min_stripe_width'* and the minimum of *'pattern_width'* and *'pattern_height'*.

If *'single_stripe_width'* is changed, the model is not decodable anymore and all possibly available results are cleared. Therefore, gen_structured_light_pattern has to be called before decoding the model using decode_structured_light_pattern.

Values: integer values larger or equal to 4 and a power of 2

Default: power of two roughly halfway between the currently set *'min_stripe_width'* and *'pattern_width'*, *'pattern_height'*

Parameters which influence the segmentation of the pattern region:

'min_gray_difference': Sets the minimum gray value difference for the Gray code images. This value is used within decode_structured_light_pattern to determine the binarized images. Thereby, for each pixel of the camera images the gray value difference between the lighted and the unlighted case is calculated. If the gray value difference of a pixel is smaller than *'min_gray_difference'*, it is excluded from the domain of the binarized images (see get_structured_light_object) and thus also from further calculations.

If *'min_gray_difference'* is set to the value 0, the segmented region is identical to the input region of the camera images with the only exception that pixels with a decoding result outside the pattern images are excluded.

Values: integer values larger or equal to 0

Default: 30

Parameters for structured light models of type 'deflectometry':

'derivative_sigma': Sets the sigma of the Gaussian (i.e. the amount of smoothing) that is used for the convolution of the correspondence image(s) to calculate the defect image.

Values: float or integer value larger or equal 0.01

Default: 2

Parameters for structured light models of type '3d_reconstruction':

'camera_setup_model': Sets a copy of the calibrated camera setup model containing the calibration parameters of the structured light sensor, such as its camera and projector parameters and poses. Here, the projector of the sensor is modeled as an 'inverse camera'. reconstruct_surface_structured_light uses the calibration parameters to determine the reconstructed surface in world coordinates. Please refer to the

example program on structured light calibration in order to obtain the calibrated camera setup model (for further details on calibration of multiple cameras see Calibration / Multi-View).

The camera of the sensor has to correspond to Index 0 in the camera setup model, the projector of the sensor to Index 1. In order to delete a set camera setup model from `StructuredLightModel`, HNULL has to be passed.

Note that the width and height of the projector parameters stored in the camera setup model (at Index 1) have to match the set *'pattern_width'* and *'pattern_height'* of the structured light model. In subsequent calls to `decode_structured_light_pattern` and `reconstruct_surface_structured_light`, the width and height of the camera parameters stored in the camera setup model (at Index 0) have to match the dimensions of the input images of `decode_structured_light_pattern`.

So far, only projectors are supported which can be modeled as an inverse pinhole camera, i.e., only projectors of type *'area_scan_division'* and *'area_scan_polynomial'*. For the camera, all area scan types except hypercentric ones are allowed.

If *'camera_setup_model'* is changed, all possibly available results are cleared, so `decode_structured_light_pattern` has to be called before reconstructing the model using `reconstruct_surface_structured_light`.

Default: HNULL

Parameters

▷ **`StructuredLightModel`** (input_control) ..... structured_light_model ⇝ *handle*
Handle of the structured light model.

▷ **`GenParamName`** (input_control) ..... attribute.name(-array) ⇝ *string*
Name of the model parameter to be adjusted.
Default: 'min_stripe_width'
List of values: `GenParamName` ∈ {'camera_setup_model', 'derivative_sigma', 'min_gray_difference', 'min_stripe_width', 'normalization', 'pattern_height', 'pattern_orientation', 'pattern_type', 'pattern_width', 'single_stripe_width', 'persistence'}

▷ **`GenParamValue`** (input_control) ..... attribute.value(-array) ⇝ *integer* / real / string
New value of the model parameter.
Default: 32
Suggested values: `GenParamValue` ∈ {0, 0.01, 0.5, 0.7, 1, 1.4, 5, 50.0, 64, 128, 256, 1024, 'true', 'false', 'both', 'vertical', 'horizontal', 'global', 'inverted_pattern', 'gray_code_and_phase_shift', 'gray_code', 'single_stripe'}

Example

```
* Create the model
create_structured_light_model ('deflectometry', StructuredLightModel)
* Set the size of the monitor
set_structured_light_model_param (StructuredLightModel, \
                                  'pattern_width', 1600)
set_structured_light_model_param (StructuredLightModel, \
                                  'pattern_height', 1200)
* Set the smallest width of the stripes in the pattern
set_structured_light_model_param (StructuredLightModel, \
                                  'min_stripe_width', 8)
* Generate the patterns to project
gen_structured_light_pattern (PatternImages, StructuredLightModel)
* Set the expected black/white contrast in the region of interest
set_structured_light_model_param (StructuredLightModel, \
                                  'min_gray_difference', 70)
* Decode the camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
* Get the computed correspondences and defects
get_structured_light_object (CorrespondenceImages, StructuredLightModel, \
                             'correspondence_image')
set_structured_light_model_param (StructuredLightModel, 'derivative_sigma', \
                                  Sigma)
```

```
get_structured_light_object (DefectImage, StructuredLightModel, \
                             'defect_image')
```

Result

The operator `set_structured_light_model_param` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- StructuredLightModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_structured_light_model`

See also

`get_structured_light_model_param`

Module

3D Metrology

write_structured_light_model (: : StructuredLightModel, FileName :)

Write a structured light model to a file.

`write_structured_light_model` writes the structured light model `StructuredLightModel` to the file given by `FileName`. The default HALCON file extension for the structured light model is 'hslm'. The structured light model can be then read with `read_structured_light_model`. It should be noted that `write_structured_light_model` does *not* write any previously generated pattern images in the structured light model. In other words, only the parameters of the model and objects that were already generated will be written in the file. The list of structured light objects and how to generate each can be found in `get_structured_light_object`.

For an explanation of the concept of structured light and its supported applications, see the introduction of chapter Inspection / Structured Light.

Parameters

▷ **StructuredLightModel** (input_control) ..... structured_light_model ⇝ *handle*
Handle of the structured light model.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
File name.
File extension: `.hslm`

Example

```
* Create the model
create_structured_light_model ('deflectometry', StructuredLightModel)
* Generate the patterns to project
gen_structured_light_pattern (PatternImages, StructuredLightModel)
* Decode the camera images
decode_structured_light_pattern (CameraImages, StructuredLightModel)
* Write the model
write_structured_light_model (StructuredLightModel, 'ExampleModel.hslm')
```

Result

The operator `write_structured_light_model` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`clear_structured_light_model`

See also

`create_structured_light_model`, `clear_structured_light_model`

Module

3D Metrology

16.4 Texture Inspection

This chapter describes the operators for texture inspection.

Concept of the texture inspection model

The texture inspection model enables the inspection of textured surfaces with only having to set few parameters. The algorithm requires images of flawless texture for training. The training process extracts texture features from the training images and trains a texture inspection model that is based on Gaussian Mixture Model (GMM) classifiers. After a successful training, texture images can be compared to the texture inspection model and potential defects can be identified. Texture inspection works on image pyramids. This way, multiple frequency ranges of the texture are analyzed.

It is possible to use multi-channel images in the texture inspection pipeline.

In the following, the steps that are required to perform the texture inspection are described briefly.

Create a texture inspection model: In a first step, a texture inspection model is created with

- `create_texture_inspection_model`

or read with

- `read_texture_inspection_model`.

Add training samples: The operator

- `add_texture_inspection_model_image`

adds sample images to the texture inspection model.

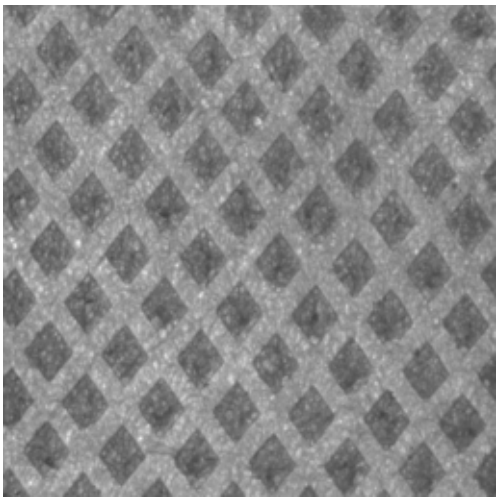


Image of a textured surface that is used to train a texture inspection model.

Images that have been added to the texture inspection model with the help of add_texture_inspection_model_image can be viewed with get_texture_inspection_model_image. If all or a part of the added images are not needed for the current texture inspection, they can be removed from the model using remove_texture_inspection_model_image.

Train a texture inspection model: The texture inspection model is trained with

- train_texture_inspection_model.

In the training process an image pyramid is created. For each pyramid level a Gaussian Mixture Model (GMM) is trained and a *'novelty_threshold'* is determined. The *'novelty_threshold'* helps to distinguish between flawless and defective texture. After the training, it is possible to query the novelty thresholds with get_texture_inspection_model_param.

Several parameters influence the training. They can be set with

- set_texture_inspection_model_param.

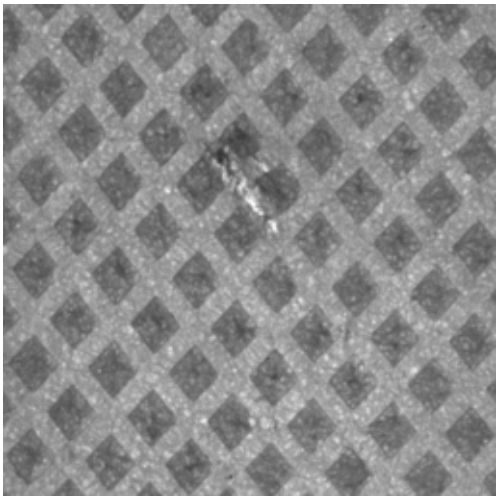
We recommend the following proceeding for the training:

- Set initial parameters for the training, if needed:
  - *'patch_normalization'*: Setting this parameter to *'weber'*, the features of the used texture patches are normalized making the texture inspection model more robust against illumination changes. Note, however, that sometimes a normalization might not be desired, e.g., if the brightness is needed to discriminate between flawless and defective parts. In that case, a controlled illumination is absolutely necessary for a successful classification.
  - *'patch_rotational_robustness'*: Setting this parameter to *'true'*, the features of the used texture patches are sorted in a way that is robust against rotational changes. This can be used if it is difficult to train all possible orientations of the textured surface via training samples.
- Train the texture inspection model (using the default parameters): The initial training of the texture inspection model may be very time consuming, depending on the number of training images and their resolution. The following time saving strategies can be considered:
  - Zoom the images: Often, the texture inspection works even better on low resolution images. If it is known that the full resolution of the images is not needed, you can significantly reduce the training time by zooming the images. Zooming the image down by a factor of 2, e.g., reduces the training time by factor 4.
  - Accelerate the training: Another, but more risky strategy is to increase the threshold for the stopping criterion of the classifier optimization by setting the parameter *'gmm_em_threshold'*, e.g., from *0.001* to *0.1*. This leads to shorter training times but also to a possibly less accurate texture inspection model. If this strategy is used, we strongly recommend to reduce *'gmm_em_threshold'* again after most of the parameters have been adjusted.

- Check the result of the training: Use a set of images with defective textures, classify them with the trained texture inspection model and check the results as is described for the step 'Apply the texture inspection model'. If the results are not satisfying, further parameters can be adjusted for the training. The most important parameters are:
  - *'num_levels'* or *'levels'*: The number of pyramid levels or the number of explicitly used pyramid levels. Higher pyramid levels work on coarser texture features. If the texture in your images is very coarse, the lower pyramid levels may not be needed for a successful inspection. Then runtime can be reduced by setting the pyramid levels of interest explicitly with *'levels'*. If for example the *'levels'* are set to [1,3], the training and the texture classification are only performed on the first and third level.
  - *'sensitivity'*: Controls how strict the novelty thresholds are set. This way, the sensitivity to novelties can be adjusted. In general, positive values lead to fewer detected defects and negative values lead to less detected defects.
  - *'novelty_threshold'*: Sets the novelty threshold manually. This can be used for fine tuning if the results of the automatic estimation are not optimal.
- Repeat the training, if needed: If parameters were adjusted, usually the texture inspection model has to be retrained.

Apply the texture inspection model: After successfully training the texture inspection model it can be used to classify textured surfaces. Each test image can be compared to the texture inspection model with

- `apply_texture_inspection_model`.



Test image with defects.

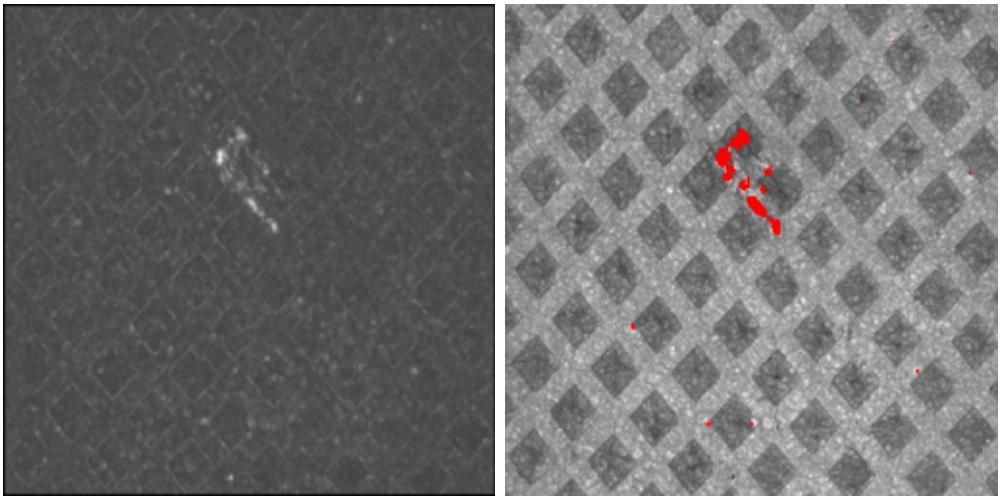
For each test image the operator returns a *'novelty_region'*, which indicates where the image has high deviations to the trained texture samples. The *'novelty_region'* is generated by combining the novelty regions of the different pyramid levels. The novelty regions of adjacent levels in the image pyramid are intersected with each other. This step helps to improve the robustness towards noise within the single pyramid level responses. The intersected novelty regions are then added to the returned *'novelty_region'*. If a pyramid level has no adjacent pyramid level, the region itself is added to the *'novelty_region'*. If, for example, *'num_levels'* is set to 1, no adjacent pyramid level exists and the *'novelty_region'* is the novelty region of the first pyramid level.

For debugging, the parameter *'gen_result_handle'* can be set to *'true'* with `set_texture_inspection_model_param`. This way, the novelty score images and novelty regions of the individual pyramid levels are stored in a result handle. They can then be read with

- `get_texture_inspection_result_object`.

In general, the novelty score images describe how well each pixel fits to the texture inspection model that was created within the training process. The novelty regions of the single pyramid levels are then calculated by applying the *'novelty_threshold'* to the corresponding novelty score images.

Inspecting the novelty score images and the novelty regions provides you with hints which kinds of errors are detectable on which pyramid levels. Thus, you can decide whether parameters for the training and the classification should be adjusted, e.g., whether it is reasonable to explicitly select only a subset of pyramid levels using the parameter *'levels'*.



(1) (2)

The resulting *'novelty_score_image'* (1) and the resulting *'novelty_region'* (2) for the test image above.

Further operators

The functionality of the texture inspection can be altered by various parameters. The parameters can be queried with `get_texture_inspection_model_param` and altered with calls to `set_texture_inspection_model_param`.

It is possible to write a texture inspection model to file with the operator `write_texture_inspection_model`. Thereby, only the texture inspection model is stored. The images which were possibly added can be queried with the help of `get_texture_inspection_model_image` and stored separately by calling `write_object`.

To help reduce the memory required by the texture inspection model, it is possible to delete previously added images from the model with `remove_texture_inspection_model_image`.

Furthermore, it is possible to serialize and deserialize the texture inspection model with `serialize_texture_inspection_model` and `deserialize_texture_inspection_model`.

Glossary

In the following, the most important terms that are used in the context of texture inspection are described:

Training images Images that are used for training.

Test images Images which are compared to the trained texture inspection model.

Patch A collection of neighboring pixels.

Texture feature The pixel values within a patch.

Sample A texture feature that is used for training.

Novelty Score In the test process the texture features of the test images are compared to the texture inspection model and their *novelty score* is calculated. The larger this value, the more probable it is that the individual texture feature does not fit to the texture inspection model.

Novelty Threshold The novelty threshold is determined during the training process. Texture features with a novelty score below the novelty threshold fit to the texture inspection model. Texture features with a higher novelty score do not.

add_texture_inspection_model_image (
Image : : TextureInspectionModel : Indices)

Add training images to the texture inspection model.

add_texture_inspection_model_image adds training images to the texture inspection model (TextureInspectionModel). The domain of the training images is considered in all further calculations. Since the feature extraction considers a certain neighborhood of each pixel, no features can be extracted for pixels at the border of the domain. The border size (*Border*) depends on the *'patch_size'* (set_texture_inspection_model_param) and computes to $Border = 'patch\_size'/2 - 1$.

All images which are added to the texture inspection model are used within the training process (see train_texture_inspection_model for more information). Furthermore, it is possible to query all of the images passed to the texture inspection model with get_texture_inspection_model_image.

Every added image receives an index (Indices), which allows the user to generate an assignment between the added images and his image data. By calls of remove_texture_inspection_model_image, the index can further be used to delete images from the model that are not needed anymore.

It is possible to add either gray-scale images or multichannel images. However, all of the passed images should have the same number of channels.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Image of flawless texture.
- ▷ **TextureInspectionModel** (input_control) ..... texture_inspection_model ⇝ *handle*
  Handle of the texture inspection model.
- ▷ **Indices** (output_control) ..... integer(-array) ⇝ *integer*
  Indices of the images that have been added to the texture inspection model.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Read and apply a test image
read_image (TrainImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModel, \
                                TextureInspectionResultID)
```

Result

The operator add_texture_inspection_model_image returns the value 2 (H_MSG_TRUE) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- TextureInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_texture_inspection_model`, `set_texture_inspection_model_param`, `get_texture_inspection_model_param`

Possible Successors

`train_texture_inspection_model`, `clear_texture_inspection_model`, `remove_texture_inspection_model_image`, `get_texture_inspection_model_image`

Module

Matching

apply_texture_inspection_model (Image : NoveltyRegion : TextureInspectionModel : TextureInspectionResultID)

Inspection of the texture within an image.

The operator `apply_texture_inspection_model` compares `Image` with the trained texture inspection model `TextureInspectionModel`. `Image` can either be a single image or a tuple of images. It is possible to pass either gray-scale or multichannel images. Hereby, the operator expects that the passed `Image` has the same number of channels than the model was trained with. The runtime of the `apply_texture_inspection_model` scales roughly linearly with the number of image channels. However, models using colored images are generally better at detecting colored texture defects.

Pixels that do not fit to the texture inspection model are returned in `NoveltyRegion`. Furthermore, if *'gen_result_handle'* has been set to *'true'* with `set_texture_inspection_model_param`, the operator also returns the result handle `TextureInspectionResultID` with more detailed information on the texture classification. If *'gen_result_handle'* is set to *'false'*, `TextureInspectionResultID` is empty.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

For each pyramid level texture features are extracted and classified with the corresponding GMM classifier. The resulting novelty score is then compared to the novelty threshold of the current pyramid level and classified as defective or not. The defective pixels are collected in a novelty region for each pyramid level. These novelty regions are then combined to the final novelty region returned in `NoveltyRegion`. The novelty regions of adjacent levels in the image pyramid are intersected with each other. This step helps to improve the robustness towards noise within the single pyramid level responses. The intersected novelty regions are then added to the returned *'novelty_region'*. If a pyramid level has no adjacent pyramid level the region itself is added to the *'novelty_region'*. If, for example, *'num_levels'* is set to 1, no adjacent pyramid level exists and the *'novelty_region'* is the novelty region of the first pyramid level.

If *'gen_result_handle'* is set to *'true'*, the result handle `TextureInspectionResultID` contains novelty score images and the resulting novelty regions for each pyramid level.

This information is useful for debugging and fine tuning of the model parameters (e.g., the novelty thresholds) and can be accessed with `get_texture_inspection_result_object`.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image-array ⇝ *object :* byte / uint2 / real
Image of the texture to be inspected.

▷ **NoveltyRegion** (output_object) . . . . . region-array ⇝ *object*
Novelty regions.

▷ **TextureInspectionModel** (input_control) . . . . . texture_inspection_model ⇝ *handle*
Handle of the texture inspection model.

▷ **TextureInspectionResultID** (output_control) . . . . . texture_inspection_result ⇝ *handle*
Handle of the inspection results.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Set parameters
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gen_result_handle', 'true')
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModel, \
                                TextureInspectionResultID)
```

Result

The operator `apply_texture_inspection_model` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.This operator modifies the state of the following input parameter:

- TextureInspectionResultID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`train_texture_inspection_model`

Possible Successors

`get_texture_inspection_result_object`, `get_texture_inspection_model_param`, `clear_texture_inspection_result`, `clear_texture_inspection_model`

Module

Matching

clear_texture_inspection_model (: : TextureInspectionModel :)

Clear a texture inspection model and free the allocated memory.

The operator `clear_texture_inspection_model` deletes a texture inspection model that was created by `create_texture_inspection_model`. All memory used by the model is freed. The handle of the model is passed in `TextureInspectionModel`. It is invalid after the operator call.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **TextureInspectionModel** (input_control) . . . . . texture_inspection_model(-array) ⇝ *handle*
Handle of the texture inspection model.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModel, \
                                TextureInspectionResultID)
```

Result

The operator `clear_texture_inspection_model` returns the value `2` (`H_MSG_TRUE`) if a valid handle is passed and the referred texture inspection model can be freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TextureInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

`create_texture_inspection_model`

Module

Matching

clear_texture_inspection_result (: : TextureInspectionResultID :)

Clear a texture inspection result handle and free the allocated memory.

The operator `clear_texture_inspection_result` clears the texture inspection result `TextureInspectionResultID` that was created by `apply_texture_inspection_model`. All memory used by the model is freed. The handle of the model is invalid after the operator call.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **TextureInspectionResultID** (input_control) . . . . . texture_inspection_result(-array) ⇝ *handle*
Handle of the texture inspection results.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Set parameter to generate a result handle
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gen_result_handle', 'true')
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModel, \
                                TextureInspectionResultID)
```

Result

The operator `clear_texture_inspection_result` returns the value `2` (`H_MSG_TRUE`) if a valid handle is passed and the referred texture inspection model can be freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TextureInspectionResultID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`apply_texture_inspection_model`

Module

Matching

create_texture_inspection_model (: : ModelType : TextureInspectionModel)

Create a texture inspection model.

`create_texture_inspection_model` creates a new texture inspection model of type `ModelType`. Currently, only the model type *'basic'* is supported.

The parameters of the texture inspection can be queried with `get_texture_inspection_model_param` and manipulated by calls to `set_texture_inspection_model_param`.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **ModelType** (input_control) . . . . . string ⇝ *string*
The type of the created texture inspection model.
Default: 'basic'
List of values: ModelType ∈ {'basic'}

▷ **TextureInspectionModel** (output_control) . . . . . texture_inspection_model ⇝ *handle*
Handle for using and accessing the texture inspection model.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModel, \
                                TextureInspectionResultID)
```

Result

The operator create_texture_inspection_model returns the value 2 (H_MSG_TRUE) in the texture inspection model can be allocated correctly. Otherwise, an exception will be raised..

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_texture_inspection_model_param, get_texture_inspection_model_param, add_texture_inspection_model_image

Module

Matching

deserialize_texture_inspection_model (: : SerializedItemHandle : TextureInspectionModel)

Deserialize a serialized texture inspection model.

deserialize_texture_inspection_model deserializes a texture inspection model, that was serialized by serialize_texture_inspection_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized texture inspection model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created texture inspection model with the handle TextureInspectionModel.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **TextureInspectionModel** (output_control) ..... texture_inspection_model ⇝ *handle*
Handle of the texture inspection model.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Set parameters
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gen_result_handle', 'true')
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Serialize texture inspection model
serialize_texture_inspection_model (TextureInspectionModel, \
                                    SerializedItemHandle)
* Deserialize in Model
deserialize_texture_inspection_model (SerializedItemHandle, \
                                      TextureInspectionModelSerialized)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModelSerialized, \
                                TextureInspectionResultID)
```

Result

If the parameters are valid, the operator `deserialize_texture_inspection_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TextureInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_texture_inspection_model`

Possible Successors

`apply_texture_inspection_model`

See also

`create_texture_inspection_model`, `write_texture_inspection_model`, `serialize_texture_inspection_model`

Module

Matching

get_texture_inspection_model_image (: ModelImages : TextureInspectionModel :)

Get the training images contained in a texture inspection model.

The operator get_texture_inspection_model_image returns all training images contained in a texture inspection model TextureInspectionModel that were added by add_texture_inspection_model_image.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **ModelImages** (output_object) ..... (multichannel-)image-array ⇝ *object :* byte / uint2
Training images contained in the texture inspection model.

▷ **TextureInspectionModel** (input_control) ..... texture_inspection_model ⇝ *handle*
Handle of the texture inspection model.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (ModelImages, TextureInspectionModel, \
                                    Indices)
* Get added training images
get_texture_inspection_model_image (ModelImagesOut, TextureInspectionModel)
```

Result

The operator get_texture_inspection_model_image returns the value 2 (H_MSG_TRUE) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

create_texture_inspection_model, set_texture_inspection_model_param, get_texture_inspection_model_param, add_texture_inspection_model_image

Possible Successors

train_texture_inspection_model, clear_texture_inspection_model, remove_texture_inspection_model_image

Module

Matching

get_texture_inspection_model_param (: : TextureInspectionModel, GenParamName : GenParamValue)

Query parameters of a texture inspection model.

The operator get_texture_inspection_model_param is used to query the values of the different parameters of a texture inspection model (TextureInspectionModel). The names of the desired parameters

are passed in the parameter GenParamName, the corresponding values are returned in GenParamValue. It is possible to query multiple parameters with a single call to get_texture_inspection_model_param. All parameters which can be manipulated by set_texture_inspection_model_param can be queried. Please refer to the documentation of set_texture_inspection_model_param for an explanation of the individual parameters. It is furthermore possible to query the following parameters:

'gmm_centers': Returns the number of Gaussian distributions which were determined as optimal within the training process. The value can be used as a reference for future texture inspection models. The range of the possible number of distributions is determined by the settings of the parameters *'gmm_pmincenters'* and *'gmm_pmaxcenters'*. Each image pyramid level can have a different number of Gaussian distributions. By adding an index, the number of Gaussian distributions for specific pyramid levels can be queried. For example, *'gmm_centers_3'* returns the number of Gaussian distributions for the third pyramid level.

'image_indices': Returns the indices of the images that have been added to the texture inspection model using add_texture_inspection_model_image. Only indices of images are returned that are currently in the texture inspection model and have not been deleted with remove_texture_inspection_model_image.

'trained_covartype': Returns the type of covariance matrix that was used during training. The user has the possibility to set the preferred type of covariance matrix by setting *'gmm_covartype'* with set_texture_inspection_model_param. However, if the training with the preferred covariance type fails, training is re-initialized with a less strict *'gmm_covartype'*. Hence, if training with *'diag'* fails, a further training with *'spherical'* is attempted.

In order to avoid confusions regarding the returned tuple length, multivalued parameters, such as *'image_indices'*, *'gmm_centers'*, *'levels'* and *'novelty_threshold'* can only be queried alone i.e. without passing any other parameter to get_texture_inspection_model_param.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **TextureInspectionModel** (input_control) . . . . . texture_inspection_model ⇝ *handle*
Handle of the texture inspection model.

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Name of the queried model parameter.
Default: 'novelty_threshold'
List of values: GenParamName ∈ {'num_levels', 'levels', 'gen_result_handle', 'novelty_threshold', 'sensitivity', 'patch_normalization', 'patch_rotational_robustness', 'patch_size', 'gmm_centers', 'gmm_sigma', 'gmm_pmincenters', 'gmm_pmaxcenters', 'gmm_preprocessing', 'gmm_ncomp', 'gmm_randseed', 'gmm_em_max_iter', 'gmm_em_threshold', 'gmm_em_regularize', 'gmm_covartype', 'image_indices', 'trained_covartype'}

▷ **GenParamValue** (output_control) . . . . . string(-array) ⇝ *integer* / real / string
Value of the queried model parameter.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Get parameters
get_texture_inspection_model_param (TextureInspectionModel, 'patch_size', \
                                    PatchSize)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Read and apply a test image
```

```
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModel, \
                                TextureInspectionResultID)
```

Result

The operator `get_texture_inspection_model_param` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_texture_inspection_model`, `train_texture_inspection_model`

Possible Successors

`add_texture_inspection_model_image`, `train_texture_inspection_model`

See also

`set_texture_inspection_model_param`

Module

Matching

get_texture_inspection_result_object (: Object : TextureInspectionResultID, ResultName :)

Query iconic results of a texture inspection.

`get_texture_inspection_result_object` queries the iconic result `ResultName` of the handle `TextureInspectionResultID` returned by `apply_texture_inspection_model`. The result handle `TextureInspectionResultID` was only created when *'gen_result_handle'* was set to *'true'* with `set_texture_inspection_model_param`.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

In the following the possible parameter values for `ResultName` are listed:

'novelty_region': Returns the novelty regions of all pyramid levels in an object tuple. To query the novelty regions of single pyramid levels the corresponding index has to be added, e.g., *'novelty_region_1'*. Please note that if a tuple of images was passed to `apply_texture_inspection_model`, it is only possible to query the novelty regions for the first image within the tuple.

'novelty_score_image': Returns the novelty score images of all pyramid levels in an object tuple. To query the novelty regions of single pyramid levels the corresponding index has to be added, e.g., *'novelty_score_image_1'*. Please note that if a tuple of images was passed to `apply_texture_inspection_model`, it is only possible to query the novelty score images for the first image within the tuple.

Parameters

▷ **Object** (output_object) ..... object(-array) ⇝ *object*
Returned iconic object.

▷ **TextureInspectionResultID** (input_control) ..... texture_inspection_result ⇝ *handle*
Handle of the texture inspection result.

▷ **ResultName** (input_control) ..... string(-array) ⇝ *string* / integer
Name of the iconic object to be returned.
Default: 'novelty_region'
List of values: `ResultName` ∈ {'novelty_score_image', 'novelty_region'}

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Set parameters
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gen_result_handle', 'true')
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModel, \
                                TextureInspectionResultID)
* Get result objects
get_texture_inspection_result_object (Object, TextureInspectionResultID, \
                                      'novelty_score_image')
```

Result

The operator `get_texture_inspection_result_object` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`apply_texture_inspection_model`

Possible Successors

`clear_texture_inspection_result`, `clear_texture_inspection_model`

Module

Matching

read_texture_inspection_model (
: : FileName : TextureInspectionModel)

Read a texture inspection model from a file.

`read_texture_inspection_model` reads a texture inspection model that has been stored with `write_texture_inspection_model`. Since the training of a texture inspection model can consume a relatively long time, the texture inspection model is typically trained in an offline process and written to a file with `write_texture_inspection_model`. In the online process the texture inspection model is read with `read_texture_inspection_model` and subsequently used for evaluation with `apply_texture_inspection_model`. The default HALCON file extension for the texture inspection model is 'htim'.

After reading a trained texture inspection model, it is possible to classify test images using `apply_texture_inspection_model`. However, retraining this model can only be done, if the read model has images. Furthermore, modifying parameters of the model leads to an entire retraining.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .htim

▷ **TextureInspectionModel** (output_control) . . . . . texture_inspection_model ⇝ *handle*
Handle of the texture inspection model.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)

* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Write out texture inspection model
write_texture_inspection_model (TextureInspectionModel, \
                                'ExampleModel.htim')
* Read in Model
read_texture_inspection_model ('ExampleModel.htim', \
                               TextureInspectionModelRead)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModelRead, \
                                TextureInspectionResultID)
```

Result

If the parameters are valid, the operator `read_texture_inspection_model` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`write_texture_inspection_model`

Possible Successors

`apply_texture_inspection_model`

See also

`create_texture_inspection_model`, `clear_texture_inspection_model`, `set_texture_inspection_model_param`, `get_texture_inspection_model_param`, `add_texture_inspection_model_image`

Module

Matching

remove_texture_inspection_model_image (
: : TextureInspectionModel, Indices : RemainingIndices)

Clear all or a user-defined subset of the images of a texture inspection model.

remove_texture_inspection_model_image clears all or certain selected images that have been stored in a texture inspection model TextureInspectionModel.

On the one hand, the usage of remove_texture_inspection_model_image is recommended to reduce the storage space of a saved texture inspection model. This is especially of advantage if the texture inspection model has been trained in an offline process and written with write_texture_inspection_model. In this case, test images can still be classified using apply_texture_inspection_model, while the required storage space is reduced to a minimum. On the other hand, it offers a simple way to delete certain selected images of the texture inspection model that are not needed (anymore) for a subsequent classification process. This allows the user to proceed with an entirely configured model, while only the used image data has to be tuned to solve the underlying inspection problem. Please note that after removing images from the model a retraining is required.

In order to specifically delete images from the texture inspection model, the indices of the respective images have to be specified in the parameter Indices. A validation of all images that are still in the texture inspection model can be made using the value *'image_indices'* in get_texture_inspection_model_param. If the user still passes unassigned indices to delete images, these indices are ignored. Besides the above mentioned option, remove_texture_inspection_model_image also allows to set the parameter Indices to the value *'all'*. In this case, all images of the texture inspection model are deleted.

It should be noted that after deleting all images, the model can only be (re)trained, if new images are added using add_texture_inspection_model_image. In the case that only a subset of all images has been deleted, a call of train_texture_inspection_model requires all steps of the training to be executed, since potentially still available training data does not match the used data.

After calling remove_texture_inspection_model_image, the remaining indices of the images that are still in the texture inspection model are returned in the parameter RemainingIndices.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **TextureInspectionModel** (input_control) ..... texture_inspection_model(-array) ⇝ *handle*
Handle of the texture inspection model.

▷ **Indices** (input_control) ..... integer-array ⇝ *integer*
Indices of the images to be deleted from the texture inspection model.

▷ **RemainingIndices** (output_control) ..... integer-array ⇝ *integer*
Indices of the images that remain in the texture inspection model.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Get added training images
get_texture_inspection_model_image (TrainImageOut, TextureInspectionModel)
* Remove training images
remove_texture_inspection_model_image (TextureInspectionModel, Indices, \
                                       RemainingIndices)
```

Result

If the parameters are valid, the operator remove_texture_inspection_model_image returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TextureInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

train_texture_inspection_model

See also

create_texture_inspection_model, clear_texture_inspection_model, set_texture_inspection_model_param, get_texture_inspection_model_param, add_texture_inspection_model_image

Module

Matching

serialize_texture_inspection_model (: : TextureInspectionModel : SerializedItemHandle)

Serialize a texture inspection model.

serialize_texture_inspection_model serializes a texture inspection model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The texture inspection model is defined by the handle TextureInspectionModel. The serialized texture inspection model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_texture_inspection_model.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **TextureInspectionModel** (input_control) ..... texture_inspection_model ⇝ *handle*
Handle of the texture inspection model.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Set parameters
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gen_result_handle', 'true')
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Serialize texture inspection model
serialize_texture_inspection_model (TextureInspectionModel, \
                                    SerializedItemHandle)
```

```
* Deserialize in Model
deserialize_texture_inspection_model (SerializedItemHandle, \
                                      TextureInspectionModelSerialized)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModelSerialized, \
                                TextureInspectionResultID)
```

Result

If the parameters are valid, the operator `serialize_texture_inspection_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`train_texture_inspection_model`

Possible Successors

`clear_texture_inspection_model`, `fwrite_serialized_item`, `send_serialized_item`, `deserialize_texture_inspection_model`

See also

`create_texture_inspection_model`, `read_texture_inspection_model`, `deserialize_texture_inspection_model`

Module

Matching

set_texture_inspection_model_param (: : TextureInspectionModel, GenParamName, GenParamValue :)

Set parameters of a texture inspection model.

The operator `set_texture_inspection_model_param` is used to manipulate the parameters of a texture inspection model `TextureInspectionModel`. The current parameter settings can be queried with the operator `get_texture_inspection_model_param`.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

General parameters:

'num_levels': Determines the number of pyramid levels generated for the texture inspection process. The number of pyramid levels affects the inspection results as well as the runtime of `train_texture_inspection_model` and `apply_texture_inspection_model`. Per default the number of levels is estimated by the size of the training images. It should be noted, that no more than 5 pyramid levels are used by the estimation process.

Changing this parameter can require a complete new training of the texture inspection model. No retraining is required if the number of levels of a trained texture inspection model are merely reduced.

If the texture in your images is very coarse, the lower pyramid levels may not be needed for a successful inspection. Then you can save a lot of runtime by setting the levels of interest explicitly with *'levels'*.

Values: integer values larger or equal to 1 or *'auto'*

Default: 'auto'

'levels': Determines the specific pyramid levels used within the texture inspection process. The number of pyramid levels affects the inspection results as well as the runtime of `train_texture_inspection_model` and

apply_texture_inspection_model. Per default the pyramid levels are determined by the value of *'num_levels'*. If for example *'num_levels'* is set to four, then the texture inspection is performed for levels 1 to 4. If all available levels should be trained, the value has to be set to *'auto'*.

Changing this parameter can require a complete new training of the texture inspection model. No retraining is required if the number of levels of a trained texture inspection model are merely reduced.

If the texture in your images is very coarse, the lower pyramid levels may not be needed for a successful inspection. Then you can save a lot of runtime by excluding unwanted levels by setting the pyramid levels with *'levels'* explicitly.

In order to avoid confusions regarding the returned tuple length, *'levels'* can only be set alone i.e. without passing any other parameter to set_texture_inspection_model_param.

Values: a monotonically increasing tuple of integer values larger or equal to 1, or *'auto'*

Default: 'auto'

'gen_result_handle': This parameter determines whether or not a result handle is returned by apply_texture_inspection_model. If set to *'true'*, the individual pyramid level results can be queried from the texture result handle with get_texture_inspection_result_object. This is useful for debugging and fine tuning.

If set to *'false'*, apply_texture_inspection_model only returns a novelty region.

Values: 'true', *'false'*

Default: 'false'

Parameters which influence the automatic novelty threshold calculation:

'novelty_threshold': This parameter sets the values of the novelty thresholds, which distinguish between defective and non-defective texture. The novelty thresholds are passed as a tuple and are set for all pyramid levels at once. Therefore, the number of elements must be equal to the number of used pyramid levels *'num_levels'* (or *'levels'*, if specified).

Setting the novelty thresholds prior to training has no effect since the training automatically determines novelty thresholds. Hence the novelty thresholds should be adapted after the training. A typical use case is the fine tuning to optimize the results of apply_texture_inspection_model. Please note, if the novelty thresholds have been set explicitly, a new call of train_texture_inspection_model will re-estimate the novelty thresholds automatically and overwrite any explicitly set novelty thresholds.

It is possible to add an index to set the novelty threshold of a specific pyramid level. For example, *'novelty_threshold_3'* sets the novelty threshold for the third pyramid level.

Values: integer or float values larger or equal to 0 and smaller or equal to 710

Default: Is determined by train_texture_inspection_model

'sensitivity': This parameter influences the sensitivity of the novelty thresholds which are determined during the training with train_texture_inspection_model. Per default *'sensitivity'* is set to 0.0, which means the novelty thresholds from the training are used directly when calling apply_texture_inspection_model. If a negative value for the parameter *'sensitivity'* is used, the value is added to the novelty boundaries. Hence, a sensitivity value below 0.0 increments the novelty thresholds and therefore leads to potentially fewer detected errors. If instead a positive value is set, the novelty thresholds are decremented which leads to potentially more detected errors.

The automatically determined novelty thresholds can be changed after the training with the parameter *'novelty_threshold'*.

Values: float or integer value

Default: 0.0

Parameters that influence the texture features:

'patch_normalization': Determines the method used to normalize the texture features. A normalization is required if the lighting between different images is not consistent. If set to *'weber'*, the patches are normalized in accordance with the Weber-Fechner law. If the parameter is set to *'none'*, no feature normalization is performed.

Changing this parameter requires a complete new training of the texture inspection model.

Values: 'none', *'weber'*

Default: 'none'

'patch_rotational_robustness': If set to *'true'*, the texture features are sorted in a way to obtain a certain amount of rotational invariance. Sorting the features leads to a loss of feature information and generally makes the texture inspection less sensitive. Also, the runtime is slightly increased. If the parameter is set to *'false'*, the texture features are not sorted.

Alternatively, it is possible to capture all possible rotations within the training images. Increasing the training images leads to an increase of the training time.

Changing this parameter requires a complete new training of the texture inspection model.

Values: 'true', *'false'*

Default: 'false'

'patch_size': This parameter determines the diameter of the extracted texture feature patches in pixels. Higher patch sizes increase the runtime significantly. Therefore this parameter should only be changed with great care. First try to change the image resolution or the number of pyramids used.

Changing this parameter requires a complete new training of the texture inspection model.

Values: odd integer values between 1 and 11

Default: 5

Advanced parameters that influence the GMM classifiers:

Internally, the texture inspection model uses a Gaussian Mixture Model (GMM) classifier. In general, it is recommended not to change the GMM parameters, but for experienced users this is still possible.

Changing these parameters requires a new training of the texture inspection model.

In the following, the parameters which influence the GMM classifiers are described. For more information on classification with GMM classifiers, see `create_class_gmm` and `train_class_gmm`.

'gmm_sigma': This parameter determines the standard deviation of the Gaussian noise which is added to the training samples (see `add_sample_class_gmm`).

Values: integer or float values larger or equal to 0.0

Default: 0.0

'gmm_pmincenters': This parameter determines the minimum number of Gaussian distributions per class. *'gmm_pmincenters'* cannot be set to a value larger than the current value of *'gmm_pmaxcenters'*.

Values: integer value larger or equal to 1 and less or equal to *'gmm_pmaxcenters'*

Default: 12

'gmm_pmaxcenters': This parameter determines the maximum number of Gaussian distributions per class. *'gmm_pmaxcenters'* cannot be set to a value smaller than the current value of *'gmm_pmincenters'*.

Values: integer value larger or equal to *'gmm_pmincenters'*

Default: 12

'gmm_preprocessing': This parameter determines the type of preprocessing used to transform the feature vectors.

Values: 'principal_components', *'normalization'*, *'none'*

Default: 'normalization'

'gmm_ncomp': This parameter manipulates the preprocessing in case *'principal_components'* is chosen as the preprocessing method. More specifically, it determines the dimension of the transformed feature vectors. It needs to be chosen such that it is smaller than the feature dimension, which is *'patch_size'* $\times$ *'patch_size'*. If the *'patch_size'* is changed such that *'patch_size'* $\times$ *'patch_size'* $<$ *'gmm_ncomp'*, the parameter is manipulated accordingly.

Values: integer value larger or equal to 1

Default: 15

'gmm_randseed': This parameter determines the initialization value for the random number generator, which is required to initialize the GMM with random values.

Values: integer value

Default: 42

'gmm_em_max_iter': This parameter determines the maximum number of iterations of the expectation maximization algorithm.

Values: integer values larger or equal to 0

Default: 100

'gmm_em_threshold': This parameter determines the threshold for the relative change of the error for the expectation maximization algorithm to terminate.

Values: integer or float value larger or equal to 0 and smaller than 1.0

Default: 0.001

'gmm_em_regularize': This parameter determines the regularization value for preventing covariance matrix singularity within the training process.

Values: integer or float values larger or equal to 0 and smaller than 1.0

Default: 0.0001

'gmm_covartype': This parameter determines the type of covariance matrix that is used internally.

Values: 'spherical', *'diag'*, *'full'*

Default: 'diag'

Changing the parameters of a trained texture inspection model

If parameters of a texture inspection model are manipulated after a successful call of `train_texture_inspection_model`, the texture inspection model usually needs to be retrained. Depending on which parameters are changed, different parts of the training need to be recalculated.

A manipulation of the parameters *'gmm_*'* requires a new training of the classifiers. Since a new classifier yields different novelty scores, the novelty thresholds also need to be recalculated. The parameter *'patch_*'* requires a complete new training of the texture inspection model. The parameters *'num_levels'* and *'levels'* require a retraining if the number of levels is increased or levels are set which were not trained beforehand. For example, setting *'levels'* from [1,2,3] to [1,3] does not require a retraining. The operator *'gen_result_handle'* has no effect on a trained texture inspection model and can be manipulated without requiring any retraining.

Parameters

▷ **`TextureInspectionModel`** (input_control) . . . . . texture_inspection_model ⇝ *handle*
Handle of the texture inspection model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the model parameter to be adjusted.
Default: 'gen_result_handle'
List of values: `GenParamName` ∈ {'num_levels', 'levels', 'gen_result_handle', 'novelty_threshold', 'sensitivity', 'patch_normalization', 'patch_rotational_robustness', 'patch_size', 'gmm_sigma', 'gmm_pmincenters', 'gmm_pmaxcenters', 'gmm_preprocessing', 'gmm_ncomp', 'gmm_randseed', 'gmm_em_max_iter', 'gmm_em_threshold', 'gmm_em_regularize', 'gmm_covartype'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value(-array) ⇝ *integer* / real / string
New value of the model parameter.
Default: 'true'
List of values: `GenParamValue` ∈ {5, 'auto', 'true', 'false', 'none', 'weber', 'principal_components', 'normalization'}

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Set parameters
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gen_result_handle', 'true')
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
```

```
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModel, \
                                TextureInspectionResultID)
```

Result

The operator `set_texture_inspection_model_param` returns the value `2` (`H_MSG_TRUE`) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TextureInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_texture_inspection_model`

Possible Successors

`add_texture_inspection_model_image`, `train_texture_inspection_model`

See also

`get_texture_inspection_model_param`

Module

Matching

`train_texture_inspection_model` (: : TextureInspectionModel :)

Train a texture inspection model.

`train_texture_inspection_model` trains a texture inspection model with all training images which were added by `add_texture_inspection_model_image`.

The complete texture inspection process works with image pyramids. The depth of the pyramid is determined by the parameter *'num_levels'* (or *'levels'* if the levels have been set explicitly). In the first step of the training, texture features are extracted and added to the set of training samples for each pyramid level. For each pyramid level a Gaussian Mixture Model (GMM) classifier is determined with all training samples of the corresponding pyramid level. In a third step, the training samples are used to determine a novelty threshold for each pyramid level. In the following, the three training steps are explained in detail:

1. **Feature extraction** extracts a feature for each pixel within the training images. The texture features are calculated according to the settings of the *'patch_*'* parameters, which can be manipulated with `set_texture_inspection_model_param`. Each texture feature is added to the training data of the GMM.
2. **During the training** of the GMMs, the optimal *'gmm_*'* parameters are calculated from the training data. The dimension of the single GMMs is determined by the size of the feature patches. Please refer to `set_texture_inspection_model_param` for details. For large patch sizes, it can be extremely difficult to obtain a good approximation of the optimal GMM parameters. Also, the run time and memory usage rises significantly. Therefore we recommend to alter the default patch size with care.

   More information on GMM classifiers can be found in the description of the operator `create_class_gmm`.

3. **The calculation of the novelty thresholds** is required to be able to distinguish between defective and non-defective texture. First, the novelty score of each training sample is determined with the GMMs determined in step 2. Then, based on the resulting novelty scores, novelty thresholds are determined for each pyramid level. The automatic determination of the novelty thresholds can be influenced with the parameter *'sensitivity'* that can be set with set_texture_inspection_model_param.

After the training was successful it is possible to classify images with apply_texture_inspection_model. Each pixel is assigned a *novelty score* that is compared to the novelty threshold, which was determined within the third step of the training. For optimal results it is possible to adapt the novelty threshold of the single pyramid levels with set_texture_inspection_model_param.

In general, the manipulation of all parameters, with the exception of *'gen_result_handle'* and *'sensitivity'*, requires a retraining of the texture inspection model. Depending on which part of the training is affected by the corresponding parameter, either the whole training or only single steps have to be recalculated. This can lead to strong variations in the time required for retraining a texture inspection model. A precise description which parameters influence which parts of the algorithm are illustrated in set_texture_inspection_model_param.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

▷ **TextureInspectionModel** (input_control) ..... texture_inspection_model ⇝ *handle*
Handle of the texture inspection model.

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Read and apply a test image
read_image (TestImage, 'carpet/carpet_02')
apply_texture_inspection_model (TestImage, DefectCandidates, \
                                TextureInspectionModel, \
                                TextureInspectionResultID)
```

Result

The operator train_texture_inspection_model returns the value 2 (H_MSG_TRUE) if the given parameters are valid and within acceptable range. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- TextureInspectionModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

add_texture_inspection_model_image, set_texture_inspection_model_param

Possible Successors

apply_texture_inspection_model, clear_texture_inspection_model, remove_texture_inspection_model_image, write_texture_inspection_model, serialize_texture_inspection_model

References

X. Xianghua, M. Mirmehdi: "TEXEMS: Texture Exemplars for Defect Detection on Random Textured Surfaces"; IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 29, No. 8; August 2007.

T. Boettger, M. Ulrich: "Real-time Texture Detection on Textured Surfaces with Compressed Sensing"; Pattern Recognition and Image Analysis, Vol. 26, No. 1, pp. 88-94; January 2016.

Module

Matching

write_texture_inspection_model (: : TextureInspectionModel, FileName :)

Write a texture inspection model to a file.

write_texture_inspection_model writes the texture inspection model TextureInspectionModel to the file given by FileName. The default HALCON file extension for the texture inspection model is 'htim'. write_texture_inspection_model is typically called after the texture inspection model has been trained with train_texture_inspection_model. However, it can also be used to save an untrained model. The texture inspection model can be read with read_texture_inspection_model. It should be noted that write_texture_inspection_model does *not* write any training samples that possibly have been stored in the texture inspection model. Instead, only the currently set parameters of the model, possibly added images and, if trained, the classifiers of the single pyramid levels are written.

For an explanation of the concept of the texture inspection see the introduction of chapter Inspection / Texture Inspection.

Parameters

- ▷ **TextureInspectionModel** (input_control) ..... texture_inspection_model ⇝ *handle*
  Handle of the texture inspection model.
- ▷ **FileName** (input_control) ..... filename.write ⇝ *string*
  File name.
  File extension: .htim

Example

```
* Create texture inspection model
create_texture_inspection_model ('basic', TextureInspectionModel)
* Make this short example fast:
set_texture_inspection_model_param (TextureInspectionModel, \
                                    'gmm_em_max_iter', 1)
* Read and add training images
read_image (TrainImage, 'carpet/carpet_01')
add_texture_inspection_model_image (TrainImage, TextureInspectionModel, \
                                    Indices)
* Train the model
train_texture_inspection_model (TextureInspectionModel)
* Write out texture inspection model
write_texture_inspection_model (TextureInspectionModel, 'ExampleModel.htim')
```

Result

If the parameters are valid, the operator write_texture_inspection_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_texture_inspection_model

Possible Successors

clear_texture_inspection_model

See also

create_texture_inspection_model, clear_texture_inspection_model, set_texture_inspection_model_param, get_texture_inspection_model_param, add_texture_inspection_model_image

Module

Matching

16.5 Variation Model

clear_train_data_variation_model (: : ModelID :)

Free the memory of the training data of a variation model.

clear_train_data_variation_model frees the memory of a variation model that was created by create_variation_model. clear_train_data_variation_model can be used to reduce the amount of memory required for the variation model (in main memory as well as when writing the model to file with write_variation_model). clear_train_data_variation_model can only be called if the model has been prepared for comparison with an image with prepare_variation_model. After the call to clear_train_data_variation_model the variation model can only be used for image comparison with compare_variation_model or compare_ext_variation_model. The model cannot be trained any further. Furthermore, the images used for the image comparison can no longer be read with get_variation_model. If they are required, get_variation_model must be called before clear_train_data_variation_model is called.

Parameters

▷ **ModelID** (input_control) ..... variation_model ⇝ *handle*
ID of the variation model.

Result

clear_train_data_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

prepare_variation_model

Possible Successors

compare_variation_model, compare_ext_variation_model, write_variation_model

Module

Matching

clear_variation_model (: : ModelID :)

Free the memory of a variation model.

clear_variation_model frees the memory of a variation model that was created by create_variation_model. After calling create_variation_model, the model can no longer be used. The handle ModelID becomes invalid.

Parameters

▷ **ModelID** (input_control) . . . . . variation_model ⇝ *handle*
ID of the variation model.

Result

clear_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_variation_model

Module

Matching

compare_ext_variation_model (Image : Region : ModelID, Mode :)

Compare an image to a variation model.

compare_ext_variation_model compares the input image Image to the variation model given by ModelID. compare_ext_variation_model is an extension of compare_variation_model that provides more modes for the image comparison. Before compare_ext_variation_model can be called, the two internal threshold images of the variation model must have been created with prepare_variation_model or prepare_direct_variation_model. Let $c(x, y)$ denote the input image Image and $t_{u,l}$ denote the two threshold images (see prepare_variation_model or prepare_direct_variation_model). Then, for Mode = *'absolute'* the output region Region contains all points that differ substantially from the model, i.e., the points that fulfill the following condition:

$$c(x, y) > t_u(x, y) \vee c(x, y) < t_l(x, y) \ .$$

This mode is identical to compare_variation_model. For Mode = *'light'*, Region contains all points that are too bright:

$$c(x, y) > t_u(x, y) \ .$$

For Mode = *'dark'*, Region contains all points that are too dark:

$$c(x, y) < t_l(x, y) \ .$$

Finally, for Mode = *'light_dark'* two regions are returned in Region. The first region contains the result of Mode = *'light'*, while the second region contains the result of Mode = *'dark'*. The respective regions can be selected with select_obj.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2
Image of the object to be compared.

▷ **Region** (output_object) . . . . . region(-array) ⇝ *object*
Region containing the points that differ substantially from the model.

▷ **ModelID** (input_control) . . . . . variation_model ⇝ *handle*
ID of the variation model.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Method used for comparing the variation model.
Default: 'absolute'
Suggested values: Mode ∈ {'absolute', 'light', 'dark', 'light_dark'}

Example

```
read_shape_model ('model.shm', TemplateID)
read_variation_model ('model.var', ModelID)
for K := 1 to 10 by 1
    read_image (Image, 'pen-' + K$'02')
    find_generic_shape_model (Image, TemplateID, MatchResultID, \
                              NumMatchResult)
    get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                    HomMat2D)
    dev_display (Image)
    if (NumMatchResult == 1)
        affine_trans_image (Image, ImageTrans, HomMat2D, 'constant', \
                            'false')
        compare_ext_variation_model (ImageTrans, RegionDiff, ModelID, \
                                     'absolute')
        dev_display (RegionDiff)
    endif
endfor
```

Result

compare_ext_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct and if the internal threshold images have been generated with prepare_variation_model or prepare_direct_variation_model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

prepare_variation_model, prepare_direct_variation_model

Possible Successors

select_obj, connection

Alternatives

compare_variation_model, dyn_threshold

See also

get_thresh_images_variation_model

Module

Matching

compare_variation_model (Image : Region : ModelID :)

Compare an image to a variation model.

compare_variation_model compares the input image Image to the variation model given by ModelID. Before compare_variation_model can be called, the two internal threshold images of the variation model must have been created with prepare_variation_model or prepare_direct_variation_model. Let $c(x, y)$ denote the input image Image and $t_{u,l}$ denote the two threshold images (see prepare_variation_model or prepare_direct_variation_model). Then the output region Region contains all points that differ substantially from the model, i.e., the points that fulfill the following condition:

$$c(x, y) > t_u(x, y) \vee c(x, y) < t_l(x, y) \ .$$

If only too bright or too dark errors should be segmented the operator compare_ext_variation_model can be used.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2
  Image of the object to be compared.
- ▷ **Region** (output_object) ..... region(-array) ⇝ *object*
  Region containing the points that differ substantially from the model.
- ▷ **ModelID** (input_control) ..... variation_model ⇝ *handle*
  ID of the variation model.

Example

```
read_shape_model ('model.shm', TemplateID)
read_variation_model ('model.var', ModelID)
for K := 1 to 10 by 1
    read_image (Image, 'pen-' + K$'02')
    find_generic_shape_model (Image, TemplateID, MatchResultID, \
                              NumMatchResult)
    get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                    HomMat2D)
    dev_display (Image)
    if (NumMatchResult == 1)
        affine_trans_image (Image, ImageTrans, HomMat2D, 'constant', \
                            'false')
        compare_variation_model (ImageTrans, RegionDiff, ModelID)
        dev_display (RegionDiff)
    endif
endfor
```

Result

compare_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct and if the internal threshold images have been generated with prepare_variation_model or prepare_direct_variation_model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

prepare_variation_model, prepare_direct_variation_model

Possible Successors

connection

Alternatives

compare_ext_variation_model, dyn_threshold

See also

get_thresh_images_variation_model

Module

Matching

create_variation_model (: : Width, Height, Type, Mode : ModelID)

Create a variation model for image comparison.

create_variation_model creates a variation model that can be used for image comparison. The handle for the variation model is returned in ModelID.

Typically, the variation model is used to discriminate correctly manufactured objects ("good objects") from incorrectly manufactured objects ("bad objects"). It is assumed that the discrimination can be done solely based on the gray values of the object.

The variation model consists of an ideal image of the object to which the images of the objects to be tested are compared later on with compare_variation_model or compare_ext_variation_model, and an image that represents the amount of gray value variation at every point of the object. The size of the images with which the object model is trained and with which the model is compared later on is passed in Width and Height, respectively. The image type of the images used for training and comparison is passed in Type.

The variation model is trained using multiple images of good objects. Therefore, it is essential that the training images show the objects in the same position and rotation. If this cannot be guaranteed by external means, the pose of the object can, for example, be determined by using matching (see find_generic_shape_model). The image can then be transformed to a reference pose with affine_trans_image.

The parameter Mode is used to determine how the image of the ideal object and the corresponding variation image are computed. For Mode='*standard*', the ideal image of the object is computed as the mean of all training images at the respective image positions. The corresponding variation image is computed as the standard deviation of the training images at the respective image positions. This mode has the advantage that the variation model can be trained iteratively, i.e., as soon as an image of a good object becomes available, it can be trained with train_variation_model. The disadvantage of this mode is that great care must be taken to ensure that only images of good objects are trained, because the mean and standard deviation are not robust against outliers, i.e., if an image of a bad object is trained inadvertently, the accuracy of the ideal object image and that of the variation image might be degraded.

If it cannot be avoided that the variation model is trained with some images of objects that can contain errors, Mode can be set to *'robust'*. In this mode, the image of the ideal object is computed as the median of all training images at the respective image positions. The corresponding variation image is computed as a suitably scaled median absolute deviation of the training images and the median image at the respective image positions. This mode has the advantage that it is robust against outliers. It has the disadvantage that it cannot be trained iteratively, i.e., all training images must be accumulated using concat_obj and be trained with train_variation_model in a single call.

In some cases, it is impossible to acquire multiple training images. In this case, a useful variation image cannot be trained from the single training image. To solve this problem, variations of the training image can be created synthetically, e.g., by shifting the training image by ± 1 pixel in the row and column directions or by using gray value morphology (e.g., gray_erosion_shape and gray_dilation_shape), and then training the synthetically modified images. A different possibility to create the variation model from a single image is to create the model with Mode='*direct*'. In this case, the variation model can only be trained by specifying the ideal image and the variation image directly with prepare_direct_variation_model. Since the variation typically is large at the edges of the object, edge operators like sobel_amp, edges_image, or gray_range_rect should be used to create the variation image.

Parameters

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the images to be compared.
Default: 640
Suggested values: Width ∈ {160, 192, 320, 384, 640, 768}

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the images to be compared.
Default: 480
Suggested values: Height ∈ {120, 144, 240, 288, 480, 576}

▷ **Type** (input_control) . . . . . string ⇝ *string*
Type of the images to be compared.
Default: 'byte'
Suggested values: Type ∈ {'byte', 'int2', 'uint2'}

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Method used for computing the variation model.
Default: 'standard'
Suggested values: Mode ∈ {'standard', 'robust', 'direct'}

▷ **ModelID** (output_control) . . . . . variation_model ⇝ *handle*
ID of the variation model.

Complexity

A variation model created with create_variation_model requires $12 * \texttt{Width} * \texttt{Height}$ bytes of memory for Mode = *'standard'* and Mode = *'robust'* for Type = *'byte'*. For Type = *'uint2'* and Type = *'int2'*, $14 * \texttt{Width} * \texttt{Height}$ are required. For Mode = *'direct'* and after the training data has been cleared with clear_train_data_variation_model, $2 * \texttt{Width} * \texttt{Height}$ bytes are required for Type = *'byte'* and $4 * \texttt{Width} * \texttt{Height}$ for the other image types.

Result

create_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

train_variation_model, prepare_direct_variation_model

See also

prepare_variation_model, clear_variation_model, clear_train_data_variation_model, find_generic_shape_model, affine_trans_image

Module

Matching

deserialize_variation_model (: : SerializedItemHandle : ModelID)

Deserialize a variation model.

deserialize_variation_model deserializes a variation model, that was serialized by serialize_variation_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized variation model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created variation model with the handle ModelID.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **ModelID** (output_control) ..... variation_model ⇝ *handle*
ID of the variation model.

Result

If the parameters are valid, the operator `deserialize_variation_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_variation_model`

Possible Successors

`compare_variation_model`, `compare_ext_variation_model`

See also

`read_variation_model`, `write_variation_model`

Module

Matching

get_thresh_images_variation_model (: MinImage, MaxImage : ModelID :)

Return the threshold images used for image comparison by a variation model.

`get_thresh_images_variation_model` returns the threshold images of the variation model `ModelID` in `MaxImage` and `MinImage`. The threshold images must be computed with `prepare_variation_model` or `prepare_direct_variation_model` before they can be read out. The formula used for calculating the threshold images is described with `prepare_variation_model` or `prepare_direct_variation_model`. The threshold images are used in `compare_variation_model` and `compare_ext_variation_model` to detect too large deviations of an image with respect to the model. As described with `compare_variation_model` and `compare_ext_variation_model`, gray values outside the interval given by `MinImage` and `MaxImage` are regarded as errors.

Parameters

▷ **MinImage** (output_object) ..... image ⇝ *object :* byte / int2 / uint2
Threshold image for the lower threshold.

▷ **MaxImage** (output_object) ..... image ⇝ *object :* real
Threshold image for the upper threshold.

▷ **ModelID** (input_control) ..... variation_model ⇝ *handle*
ID of the variation model.

Result

`get_thresh_images_variation_model` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

prepare_variation_model, prepare_direct_variation_model

See also

compare_variation_model, compare_ext_variation_model

Module

Matching

get_variation_model (: Image, VarImage : ModelID :)

Return the images used for image comparison by a variation model.

get_variation_model returns the image of the ideal object and the corresponding variation image of the variation model ModelID in Image and VarImage, respectively. The returned images can be used to check whether an image of a bad object has been trained with train_variation_model. This can be seen from the variation image. If an image of a bad object has been trained, the variation image typically has large variations in areas that should exhibit no variations.

Parameters

- ▷ **Image** (output_object) ..... image ⇝ *object :* byte / int2 / uint2
  Image of the trained object.
- ▷ **VarImage** (output_object) ..... image ⇝ *object :* real
  Variation image of the trained object.
- ▷ **ModelID** (input_control) ..... variation_model ⇝ *handle*
  ID of the variation model.

Result

get_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_variation_model

See also

prepare_variation_model, compare_variation_model, compare_ext_variation_model

Module

Matching

prepare_direct_variation_model (RefImage, VarImage : : ModelID, AbsThreshold, VarThreshold :)

Prepare a variation model for comparison with an image.

prepare_direct_variation_model prepares a variation model for the image comparison with compare_variation_model or compare_ext_variation_model. The variation model must have been created with Mode='direct' with create_variation_model. In contrast to prepare_variation_model, the ideal image of the object and the corresponding variation image are not computed with train_variation_model, but are specified directly in RefImage and VarImage. This is useful if the variation model should be created from a single image, as described with create_variation_model. The variation image should typically be created with edge operators like sobel_amp, edges_image, or gray_range_rect.

prepare_direct_variation_model converts the ideal image RefImage and the variation image VarImage into two threshold images and stores them in the variation model. These threshold images are used

in compare_variation_model or compare_ext_variation_model to perform the comparison of the current image to the variation model.

Two thresholds are used to compute the threshold images. The parameter AbsThreshold determines the minimum amount of gray levels by which the image of the current object must differ from the image of the ideal object. The parameter VarThreshold determines a factor relative to the variation image for the minimum difference of the current image and the ideal image. VarThreshold utilizes VarImage to define regions, in which differences in gray values may occur which should not be counted as errors. This allows variations for specific parts of the object to inspect. AbsThreshold and VarThreshold each can contain one or two values. If two values are specified, different thresholds can be determined for too bright and too dark pixels. In this mode, the first value refers to too bright pixels, while the second value refers to too dark pixels. If one value is specified, this value refers to both the too bright and too dark pixels. Let $i(x,y)$ be the ideal image RefImage, $v(x,y)$ the variation image VarImage, a_u = AbsThreshold[0], a_l = AbsThreshold[1], b_u = VarThreshold[0], and b_l = VarThreshold[1] (or a_u = AbsThreshold, a_l = AbsThreshold, b_u = VarThreshold, and b_l = VarThreshold, respectively). Then the two threshold images $t_{u,l}$ are computed as follows:

$$t_u(x,y) = i(x,y) + \max\{a_u, b_u v(x,y)\} \qquad t_l(x,y) = i(x,y) - \max\{a_l, b_l v(x,y)\} \ .$$

If the current image $c(x,y)$ is compared to the variation model using compare_variation_model, the output region contains all points that differ substantially from the model, i.e., that fulfill the following condition:

$$c(x,y) > t_u(x,y) \vee c(x,y) < t_l(x,y) \ .$$

In compare_ext_variation_model, extended comparison modes are available, which return only too bright errors, only too dark errors, or bright and dark errors as separate regions.

After the threshold images have been created they can be read out with get_thresh_images_variation_model.

It should be noted that RefImage and VarImage are not stored as the ideal and variation images in the model to save memory in the model.

Parameters

- ▷ **RefImage** (input_object) ..... singlechannelimage ⇝ *object :* byte / int2 / uint2
  Reference image of the object.
- ▷ **VarImage** (input_object) ..... singlechannelimage ⇝ *object :* byte / int2 / uint2
  Variation image of the object.
- ▷ **ModelID** (input_control) ..... variation_model ⇝ *handle*
  ID of the variation model.
- ▷ **AbsThreshold** (input_control) ..... number(-array) ⇝ *real* / integer
  Absolute minimum threshold for the differences between the image and the variation model.
  Default: 10
  Suggested values: AbsThreshold ∈ {0, 5, 10, 15, 20, 30, 40, 50}
  Restriction: AbsThreshold >= 0
- ▷ **VarThreshold** (input_control) ..... number(-array) ⇝ *real* / integer
  Threshold for the differences based on the variation of the variation model.
  Default: 2
  Suggested values: VarThreshold ∈ {1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5}
  Restriction: VarThreshold >= 0

Example

```
read_image (Image, 'model')
sobel_amp (Image, VarImage, 'sum_abs', 3)
get_image_pointer1 (Image, Pointer, Type, Width, Height)
create_variation_model (Width, Height, Type, 'direct', ModelID)
prepare_direct_variation_model (Image, VarImage, ModelID, 20, 1)
write_variation_model (ModelID, 'model.var')
```

Result

prepare_direct_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

sobel_amp, edges_image, gray_range_rect

Possible Successors

compare_variation_model, compare_ext_variation_model, get_thresh_images_variation_model, write_variation_model

Alternatives

prepare_variation_model

See also

create_variation_model

Module

Matching

prepare_variation_model (: : ModelID, AbsThreshold, VarThreshold :)

Prepare a variation model for comparison with an image.

prepare_variation_model prepares a variation model for the image comparison with compare_variation_model or compare_ext_variation_model. This is done by converting the ideal image and the variation image that have been trained with train_variation_model into two threshold images and storing them in the variation model. These threshold images are used in compare_variation_model or compare_ext_variation_model to speed up the comparison of the current image to the variation model.

Two thresholds are used to compute the threshold images. The parameter AbsThreshold determines the minimum amount of gray levels by which the image of the current object must differ from the image of the ideal object. The parameter VarThreshold determines a factor relative to the variation image for the minimum difference of the current image and the ideal image. AbsThreshold and VarThreshold each can contain one or two values. If two values are specified, different thresholds can be determined for too bright and too dark pixels. In this mode, the first value refers to too bright pixels, while the second value refers to too dark pixels. If one value is specified, this value refers to both the too bright and too dark pixels. Let $i(x,y)$ be the ideal image, $v(x,y)$ the variation image, a_u = AbsThreshold[0], a_l = AbsThreshold[1], b_u = VarThreshold[0], and b_l = VarThreshold[1] (or a_u = AbsThreshold, a_l = AbsThreshold, b_u = VarThreshold, and b_l = VarThreshold, respectively). Then the two threshold images $t_{u,l}$ are computed as follows:

$$t_u(x,y) = i(x,y) + \max\{a_u, b_u v(x,y)\} \qquad t_l(x,y) = i(x,y) - \max\{a_l, b_l v(x,y)\} \ .$$

If the current image $c(x,y)$ is compared to the variation model using compare_variation_model, the output region contains all points that differ substantially from the model, i.e., that fulfill the following condition:

$$c(x,y) > t_u(x,y) \vee c(x,y) < t_l(x,y) \ .$$

In compare_ext_variation_model, extended comparison modes are available, which return only too bright errors, only too dark errors, or bright and dark errors as separate regions.

After the threshold images have been created they can be read out with get_thresh_images_variation_model. Furthermore, the training data can be deleted with clear_train_data_variation_model to save memory.

Parameters

▷ **ModelID** (input_control) . . . . . variation_model ⇝ *handle*
ID of the variation model.

▷ **AbsThreshold** (input_control) . . . . . number(-array) ⇝ *real* / integer
Absolute minimum threshold for the differences between the image and the variation model.
Default: 10
Suggested values: AbsThreshold ∈ {0, 5, 10, 15, 20, 30, 40, 50}
Restriction: AbsThreshold >= 0

▷ **VarThreshold** (input_control) . . . . . number(-array) ⇝ *real* / integer
Threshold for the differences based on the variation of the variation model.
Default: 2
Suggested values: VarThreshold ∈ {1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5}
Restriction: VarThreshold >= 0

Result

prepare_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

train_variation_model

Possible Successors

compare_variation_model, compare_ext_variation_model, get_thresh_images_variation_model, clear_train_data_variation_model, write_variation_model

Alternatives

prepare_direct_variation_model

See also

create_variation_model

Module

Matching

read_variation_model (: : FileName : ModelID)

Read a variation model from a file.

The operator read_variation_model reads a variation model, which has been written with write_variation_model, from the file FileName. The default HALCON file extension for a variation model is 'vam'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .vam

▷ **ModelID** (output_control) . . . . . variation_model ⇝ *handle*
ID of the variation model.

Result

If the file name is valid, the operator `read_variation_model` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`compare_variation_model`, `compare_ext_variation_model`

See also

`write_variation_model`

Module

Matching

serialize_variation_model (: : ModelID : SerializedItemHandle)

Serialize a variation model.

`serialize_variation_model` serializes the data of a variation model (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The same data that is written in a file by `write_variation_model` is converted to a serialized item. The variation model is defined by the handle `ModelID`. The serialized variation model is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_variation_model`.

Parameters

▷ **ModelID** (input_control) . . . . . variation_model ⇝ *handle*
ID of the variation model.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_variation_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`train_variation_model`

Possible Successors

`fwrite_serialized_item`, `send_serialized_item`, `deserialize_variation_model`

See also

`read_variation_model`, `write_variation_model`

Module

Matching

train_variation_model (Images : : ModelID :)

Train a variation model.

train_variation_model trains the variation model that is passed in ModelID with one or more images, which are passed in Images.

As described for create_variation_model, a variation model that has been created using the mode *'standard'* can be trained iteratively, i.e., as soon as images of good objects become available, they can be trained with train_variation_model. The ideal image of the object is computed as the mean of all previous training images and the images that are passed in Images. The corresponding variation image is computed as the standard deviation of the training images and the images that are passed in Images.

If the variation model has been created using the mode *'robust'*, the model cannot be trained iteratively, i.e., all training images must be accumulated using concat_obj and be trained with train_variation_model in a single call. If any images have been trained previously, the training information of the previous call is discarded. The image of the ideal object is computed as the median of all training images passed in Images. The corresponding variation image is computed as a suitably scaled median absolute deviation of the training images and the median image.

Attention

At most 65535 training images can be trained.

Parameters

- ▷ **Images** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2
  Images of the object to be trained.
- ▷ **ModelID** (input_control) ..... variation_model ⇝ *handle*
  ID of the variation model.

Example

```
create_variation_model (Width, Height, Type, 'standard', ModelID)
for K := 1 to 10 by 1
    read_image (Image, 'pen-' + K$'02')
    find_generic_shape_model (Image, TemplateID, MatchResultID, \
                              NumMatchResult)
    get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                    HomMat2D)
    if (NumMatchResult == 1)
        affine_trans_image (Image, ImageTrans, HomMat2D, 'constant', \
                            'false')
        train_variation_model (ImageTrans, ModelID)
    endif
endfor
prepare_variation_model (ModelID, 10, 4)
```

Result

train_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_variation_model, find_generic_shape_model, affine_trans_image, concat_obj

Possible Successors

prepare_variation_model

See also

prepare_variation_model, compare_variation_model, compare_ext_variation_model, clear_variation_model

Module

Matching

write_variation_model (: : ModelID, FileName :)

Write a variation model to a file.

write_variation_model writes a variation model to the file FileName. The model can be read with read_variation_model. The default HALCON file extension for a variation model is 'vam'.

Parameters

▷ **ModelID** (input_control) . . . . . variation_model ⇝ *handle*
ID of the variation model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: .vam

Result

If the file name is valid (write permission), the operator write_variation_model returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_variation_model

See also

read_variation_model

Module

Matching

Chapter 17

Legacy

17.1 2D Metrology

```
copy_metrology_object ( : : MetrologyHandle,
    Index : CopiedIndices )
```

Copy metrology metrology objects of a metrology model.

copy_metrology_object is obsolete and is only provided for reasons of backward compatibility.

copy_metrology_object copies metrology metrology objects within this metrology model.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The metrology model is defined by the handle MetrologyHandle. The parameter Index determines the metrology objects that are copied. With Index set to *'all'*, all metrology objects are copied. The operator copy_metrology_object returns the index of the new created metrology objects in the parameter CopiedIndices. The order of the new metrology objects corresponds to the order of the original metrology objects. Access to the parameters of the metrology objects of the metrology model is possible, e.g., with the operator get_metrology_object_param.

Parameters

- ▷ **MetrologyHandle** (input_control) . . . . . metrology_model ⇝ *handle*
  Handle of the metrology model.
- ▷ **Index** (input_control) . . . . . integer(-array) ⇝ *string* / integer
  Index of the metrology objects.
  Default: 'all'
  Suggested values: Index ∈ {'all', 0, 1, 2}
- ▷ **CopiedIndices** (output_control) . . . . . integer(-array) ⇝ *integer*
  Indices of the copied metrology objects.

Result

If the parameters are valid, the operator copy_metrology_object returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

2D Metrology

transform_metrology_object (: : MetrologyHandle, Index, Row, Column, Phi, Mode :)

Transform metrology objects of a metrology model, e.g., for alignment.

transform_metrology_object is obsolete and is only provided for reasons of backward compatibility. New applications should use the operator align_metrology_model instead.

transform_metrology_object translates and rotates the metrology objects contained in MetrologyHandle specified by the index Index according to the parameters Row, Column, and Phi.

For an explanation of the concept of 2D metrology see the introduction of chapter 2D Metrology.

The index Index refer to the index returned by the operators add_metrology_object_circle_measure, add_metrology_object_ellipse_measure, add_metrology_object_line_measure, or add_metrology_object_rectangle2_measure. Index may contain a single value or a tuple of values. If all metrology objects shall be transformed, Index can be set to *'all'*.

The parameter Mode can be set to *'absolute'* or *'relative'* and specifies the effect of the transformation:

Mode = *'absolute'*: The metrology objects are translated to the image coordinates Row and Column and rotated by Phi with respect to the image coordinate system.
For metrology objects of the type ellipse, or rectangle, the origin of the rotation is defined in the center of the metrology object as specified in the operators add_metrology_object_ellipse_measure or add_metrology_object_rectangle2_measure. For lines and circles, Phi is ignored.

Mode = *'relative'*: The values of the transformation are considered as relative values, i.e., they are specified relative to the previous position and rotation of the metrology object.
For metrology objects of the type ellipse, or rectangle, the origin of the rotation is defined in the center of the metrology object as specified in the operators add_metrology_object_ellipse_measure, or add_metrology_object_rectangle2_measure. For lines, the starting point of the line is used as origin of the rotation. For circles, Phi is ignored.

transform_metrology_object is fastest if no rotation is defined, the original and translated measure regions both lie completely within the image. In all other cases, the measure regions have to be newly generated, which is slower.

Attention

Note that any results (fitted geometric shapes) that have been generated by the operator apply_metrology_model before calling transform_metrology_object are discarded during the transformation.

Parameters

▷ **MetrologyHandle** (input_control) ..... metrology_model ⇝ *handle*
Handle of the metrology model.

▷ **Index** (input_control) ..... integer(-array) ⇝ *string* / integer
Index of the metrology objects.
Default: 'all'
Suggested values: Index ∈ {'all', 0, 1, 2}

▷ **Row** (input_control) ..... real(-array) ⇝ *real* / integer
Translation in row direction.

▷ **Column** (input_control) ..... real(-array) ⇝ *real* / integer
Translation in column direction.

▷ **Phi** (input_control) ..... real(-array) ⇝ *real* / integer
Rotation angle.

▷ **Mode** (input_control) ..... string(-array) ⇝ *string*
Mode of the transformation.
Default: 'absolute'
Suggested values: Mode ∈ {'absolute', 'relative'}

Result

If the parameters are valid, the operator transform_metrology_object returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MetrologyHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

2D Metrology

17.2 Classification

clear_sampset (: : SampKey :)

Free memory of a data set.

clear_sampset is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

clear_sampset frees the memory which was used for training data set having read by read_sampset. This memory is only reusable in combination with read_sampset.

Parameters

▷ **SampKey** (input_control) ..... feature_set ⇝ *handle*
Number of the data set.

Result

clear_sampset returns 2 (H_MSG_TRUE). An exception is raised if the key SampKey does not exist.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SampKey

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_box, enquire_class_box, learn_class_box, write_class_box

See also

test_sampset_box, learn_sampset_box, read_sampset

Module

Foundation

close_class_box (: : ClassifHandle :)

Destroy the classifier.

close_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

close_class_box deletes the classifier and frees the used memory space. All the trained data will be lost. For saving this trained data you should use write_class_box before.

Parameters

▷ **ClassifHandle** (input_control) ..... class_box ⇝ *handle*
Handle of the classifier.

Result

close_class_box returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ClassifHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_box, enquire_class_box, learn_class_box, write_class_box

See also

create_class_box, enquire_class_box, learn_class_box

Module

Foundation

create_class_box (: : : ClassifHandle)

Create a new classifier.

create_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

create_class_box creates a new adaptive classifier. All operators which are explained in chapter classification refer to such a initialized classifier (of type 2). See learn_class_box for more details about the functionality of the classifier.

Parameters

▷ **ClassifHandle** (output_control) ..... class_box ⇝ *handle*
Handle of the classifier.

Result

create_class_box returns 2 (H_MSG_TRUE) if the parameter is correct. An exception is raised if a classifier with this name already exists or there is not enough memory.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

learn_class_box, enquire_class_box, write_class_box, close_class_box, clear_sampset

See also

learn_class_box, enquire_class_box, close_class_box

Module

Foundation

descript_class_box (: : ClassifHandle, Dimensions : ClassIdx, BoxIdx, BoxLowerBound, BoxHigherBound, BoxNumSamplesTrain, BoxNumSamplesWrong)

Describe the classes of a box classifier.

descript_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

descript_class_box describes the classes of a box classifier. A box classifier uses a set of hyper cuboids (boxes) for every class. These boxes describe the distribution of the samples.

descript_class_box returns for every class (ClassIdx) the boundaries of every contained box (BoxIdx) from dimension 1 up to Dimensions (BoxLowerBound, BoxHigherBound) as well as the number of samples that were used during the learning phase to determine these boundaries at each dimension (BoxNumSamplesTrain). Furthermore, the number of samples that were assigned to the wrong class during the learning phase is returned in BoxNumSamplesWrong.

The boundary information of the boxes can be used to inspect the box classifier.

Parameters

- ▷ **ClassifHandle** (input_control) . . . class_box ⇝ *handle*
  Handle of the classifier.
- ▷ **Dimensions** (input_control) . . . integer ⇝ *integer*
  Highest dimension for output.
  Default: 3
- ▷ **ClassIdx** (output_control) . . . integer(-array) ⇝ *integer*
  Indices of the classes.
- ▷ **BoxIdx** (output_control) . . . integer(-array) ⇝ *integer*
  Indices of the boxes.
- ▷ **BoxLowerBound** (output_control) . . . integer(-array) ⇝ *integer*
  Lower bounds of the boxes (for each dimension).
- ▷ **BoxHigherBound** (output_control) . . . integer(-array) ⇝ *integer*
  Higher bounds of the boxes (for each dimension).
- ▷ **BoxNumSamplesTrain** (output_control) . . . integer(-array) ⇝ *integer*
  Number of training samples that were used to define this box (for each dimension).
- ▷ **BoxNumSamplesWrong** (output_control) . . . integer(-array) ⇝ *integer*
  Number of training samples that were assigned incorrectly to the box.

Result

descript_class_box returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_class_box, learn_class_box, set_class_box_param

Possible Successors

enquire_class_box, learn_class_box, write_class_box, close_class_box

See also

create_class_box, enquire_class_box, learn_class_box, read_class_box, write_class_box

Module

Foundation

deserialize_class_box (: : ClassifHandle, SerializedItemHandle :)

Deserialize a serialized classifier.

deserialize_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

deserialize_class_box deserializes a classifier, that was serialized by serialize_class_box (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized classifier is defined by the handle SerializedItemHandle. The values of the current classifier, defined by the handle ClassifHandle, are overwritten.

Attention

All values of the classifier are going to be overwritten.

Parameters

▷ **ClassifHandle** (input_control) ..... class_box ⇝ *handle*
Handle of the classifier.

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator deserialize_class_box returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_class_box, create_class_box

Possible Successors

test_sampset_box, enquire_class_box, write_class_box, close_class_box, clear_sampset

See also

create_class_box, serialize_class_box

Module

Foundation

enquire_class_box (: : ClassifHandle, FeatureList : Class)

Classify a tuple of attributes.

enquire_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

FeatureList is a tuple of any floating point- or integer numbers (attributes) which has to be assigned to a class with assistance of a previous trained (learn_class_box) classifier. It is possible to specify attributes as unknown by indicating the symbol '∗' instead of a number. If you specify n values, then all following values, i.e. the attributes n+1 until 'max', are automatically supposed to be undefined.

See learn_class_box for more details about the functionality of the classifier.

You may call the operators learn_class_box and enquire_class_box alternately, so that it is possible to classify already in the phase of learning. This means you could see when a satisfying behavior had been reached.

Parameters

- ▷ **ClassifHandle** (input_control) . . . . . class_box ⇝ *handle*
  Handle of the classifier.
- ▷ **FeatureList** (input_control) . . . . . real-array ⇝ *real* / integer / string
  Array of attributes which has to be classified.
  Default: 1.0
- ▷ **Class** (output_control) . . . . . integer ⇝ *integer*
  Number of the class to which the array of attributes had been assigned.

Result

enquire_class_box returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_class_box, learn_class_box, set_class_box_param

Possible Successors

learn_class_box, write_class_box, close_class_box

Alternatives

enquire_reject_class_box

See also

test_sampset_box, learn_class_box, learn_sampset_box

Module

Foundation

```
enquire_reject_class_box ( : : ClassifHandle,
    FeatureList : Class )
```

Classify a tuple of attributes with rejection class.

enquire_reject_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

FeatureList is a tuple of any floating point- or integer numbers (attributes) which has to be assigned to a class with assistance of a previous trained (learn_class_box) classifier. It is possible to specify attributes as unknown by indicating the symbol '∗' instead of a number. If you specify n values, then all following values, i.e. the attributes n+1 until 'max', are automatically supposed to be undefined.

If the array of attributes cannot be assigned to a class, i.e. the array does not reside inside of one of the hyper boxes, then in contrary to enquire_class_box not the next class is going to be returned, but the rejection class *-1* as a result is going to be passed.

See learn_class_box for more details about the functionality of the classifier.

You may call the operators learn_class_box and enquire_class_box alternately, so that it is possible to classify already in the phase of learning. By this means you could see when a satisfying behavior had been reached.

Parameters

▷ **ClassifHandle** (input_control) . . . . . class_box ⇝ *handle*
Handle of the classifier.

▷ **FeatureList** (input_control) . . . . . real-array ⇝ *real* / integer / string
Array of attributes which has to be classified.
Default: 1.0

▷ **Class** (output_control) . . . . . integer ⇝ *integer*
Number of the class, to which the array of attributes had been assigned or -1 for the rejection class.

Result

enquire_reject_class_box returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_class_box, learn_class_box, set_class_box_param

Possible Successors

learn_class_box, write_class_box, close_class_box

Alternatives

enquire_class_box

See also

test_sampset_box, learn_class_box, learn_sampset_box

Module

Foundation

get_class_box_param (: : ClassifHandle, Flag : Value)

Get information about the current parameter.

get_class_box_param is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

get_class_box_param gets the parameter of the classifier. The meaning of the parameter is explained in set_class_box_param.

Default:

'min_samples_for_split' = 80, 'split_error' = 0.1, 'prop_constant' = 0.25.

Parameters

▷ **ClassifHandle** (input_control) . . . . . class_box ⇝ *handle*
Handle of the classifier.

▷ **Flag** (input_control) . . . . . string ⇝ *string*
Name of the system parameter.
Default: 'split_error'
List of values: Flag ∈ {'split_error', 'prop_constant', 'used_memory', 'min_samples_for_split'}

▷ **Value** (output_control) . . . . . number ⇝ *real* / integer
Value of the system parameter.

Result

`get_class_box_param` returns `2` (`H_MSG_TRUE`). An exception is raised if `Flag` has been set with wrong values.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_class_box`, `enquire_class_box`, `learn_class_box`, `write_class_box`

Possible Successors

`set_class_box_param`, `learn_class_box`, `enquire_class_box`, `write_class_box`, `close_class_box`, `clear_sampset`

See also

`create_class_box`, `set_class_box_param`

Module

Foundation

`learn_class_box` (: : `ClassifHandle`, `Features`, `Class` :)

Train the classifier.

`learn_class_box` is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

`Features` is a tuple of any floating point numbers or integers (attributes) which has to be assigned to the class `Class`. This class is specified by an integer. You may use the operator `enquire_class_box` later to find the most probable class for any array (=tuple). The algorithm tries to describe the set of arrays of one class by hyper cuboids in the feature space. On demand you may even create several cuboids per class. Hence it is possible to learn disjunct concepts, too. I.e such concepts which split in several "cluster" of points in the feature space. The data structure is hidden to the user and only accessible with such operators which are described in this chapter.

Note that if a class consists of disjunct sub-classes that would lead to a splitting of the respective hyper cuboid, the training samples should be in random order with respect to the sub-classes. Otherwise, the splitting of the hyper cuboid will be sub-optimal.

It is possible to specify attributes as unknown by indicating the symbol '$*$' instead of a number. If you specify n values, then all following values, i.e. the attributes n+1 until 'max', are automatically supposed to be undefined.

You may call the operators `learn_class_box` and `enquire_class_box` alternately, so that it is possible to classify already in the phase of learning. By this means you could see when a satisfying behavior had been reached.

The classifier is going to be bigger using further training. This means, that it is not advisable to continue training after reaching a satisfactory behavior.

Parameters

▷ **`ClassifHandle`** (input_control) . . . . . class_box ⇝ *handle*
Handle of the classifier.

▷ **`Features`** (input_control) . . . . . number-array ⇝ *real* / integer / string
Array of attributes to learn.
Default: [1.0,1.5,2.0]

▷ **`Class`** (input_control) . . . . . integer ⇝ *integer*
Class to which the array has to be assigned.
Default: 1

Result

learn_class_box returns 2 (H_MSG_TRUE) for a normal case. An exception is raised if there are memory allocation problems. The number of classes is constrained. If this limit is passed, an exception is raised, too.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ClassifHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_box, enquire_class_box

Possible Successors

test_sampset_box, enquire_class_box, write_class_box, close_class_box, clear_sampset

See also

test_sampset_box, close_class_box, create_class_box, enquire_class_box, learn_sampset_box

Module

Foundation

learn_sampset_box (: : ClassifHandle, SampKey, Outfile, NSamples, StopError, ErrorN :)

Train the classifier with one data set.

learn_sampset_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

learn_sampset_box trains the classifier with data for the key SampKey (see read_sampset). The training sequence is terminated at least after NSamples examples. If NSamples is bigger than the number of examples in SampKey, then a cyclic start at the beginning occurs. If the error underpasses the value StopError, then the training sequence is prematurely terminated. StopError is calculated with N / ErrorN. Whereby N means the number of examples which were wrong classified during the last ErrorN training examples. Typically ErrorN is the number of examples in SampKey and NSamples is a multiple of it. If you want a data set with 100 examples to run 5 times at most and if you want it to terminate with an error lower than 5%, then the corresponding values are NSamples = 500, ErrorN = 100 and StopError = 0.05. A protocol of the training activity is going to be written in file Outfile.

Parameters

▷ **ClassifHandle** (input_control) ..... class_box ⇝ *handle*
Handle of the classifier.

▷ **SampKey** (input_control) ..... feature_set ⇝ *handle*
Number of the data set to train.

▷ **Outfile** (input_control) ..... filename.write ⇝ *string*
Name of the protocol file.
Default: 'training_prot'

▷ **NSamples** (input_control) ..... integer ⇝ *integer*
Number of arrays of attributes to learn.
Default: 500

▷ **StopError** (input_control) ..... real ⇝ *real*
Classification error for termination.
Default: 0.05

▷ **ErrorN** (input_control) ..... integer ⇝ *integer*
Error during the assignment.
Default: 100

Result

`learn_sampset_box` returns `2` (`H_MSG_TRUE`). An exception is raised if key `SampKey` does not exist or there are problems while opening the file.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ClassifHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_class_box`

Possible Successors

`test_sampset_box`, `enquire_class_box`, `write_class_box`, `close_class_box`, `clear_sampset`

See also

`test_sampset_box`, `enquire_class_box`, `learn_class_box`, `read_sampset`

Module

Foundation

read_class_box (: : ClassifHandle, FileName :)

Read a classifier from a file.

read_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

`read_class_box` reads the saved classifier from the file `FileName` (see `write_class_box`). The values of the current classifier are overwritten. The default HALCON file extension for the box classifier is 'gbc'.

Attention

All values of the classifier are going to be overwritten.

Parameters

▷ **ClassifHandle** (input_control) ..... class_box ⇝ *handle*
Handle of the classifier.

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Filename of the classifier.
File extension: `.gbc`

Result

`read_class_box` returns `2` (`H_MSG_TRUE`). An exception is raised if it was not possible to open file `FileName` or the file has the wrong format.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ClassifHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_box

Possible Successors

test_sampset_box, enquire_class_box, write_class_box, close_class_box, clear_sampset

See also

create_class_box, write_class_box

Module

Foundation

read_sampset (: : FileName : SampKey)

Read a training data set from a file.

read_sampset is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead.

The training examples are accessible with the key SampKey by calling the operators clear_sampset and learn_sampset_box. You may edit the file using an editor. Every row contains an array of attributes with corresponding class. An example for a format might be:

```
(1.0, 25.3, *, 17 | 3)
```

This row specifies an array of attributes which belong to class 3. In this array the third attribute is unknown. Attributes upwards 5 are supposed to be unknown, too. You may insert comments like /* .. */ in any place.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Filename of the data set to train.
Default: 'sampset1'

▷ **SampKey** (output_control) . . . . . feature_set ⇝ *handle*
Identification of the data set to train.

Result

read_sampset returns 2 (H_MSG_TRUE). An exception is raised if it is not possible to open the file or it contains syntax errors or there is not enough memory.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

create_class_box

Possible Successors

test_sampset_box, enquire_class_box, write_class_box, close_class_box, clear_sampset

See also

test_sampset_box, clear_sampset, learn_sampset_box

Module

Foundation

serialize_class_box (: : ClassifHandle : SerializedItemHandle)

Serialize a classifier.

serialize_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

serialize_class_box serializes a classifier (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_class_box is converted to a serialized item. The classifier is defined by the handle ClassifHandle. The serialized classifier is returned by the handle SerializedItemHandle and can be deserialized by deserialize_class_box.

Parameters

▷ **ClassifHandle** (input_control) ..... class_box ⇝ *handle*
Handle of the classifier.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_class_box returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_class_box, enquire_class_box, learn_class_box, test_sampset_box

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_class_box

See also

create_class_box, deserialize_class_box

Module

Foundation

set_class_box_param (: : ClassifHandle, Flag, Value :)

Set system parameters for classification.

set_class_box_param is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

set_class_box_param modifies parameter which manipulate the training sequence while calling learn_class_box. Only parameters of the classifier are modified, all other classifiers remain unmodified. *'min_samples_for_split'* is the number of examples at least which have to train in one cuboid of this classifier, before the cuboid is allowed to divide itself. *'split_error'* indicates the critical error. By its exceeding the cuboid divides itself, if there are more than *'min_samples_for_split'* examples to train. *'prop_constant'* manipulates the

extension of the cuboids. It is proportional to the average distance of the training examples in this cuboid to the center of the cuboid. More detailed:

extension × prop = average distance of the expectation value.

This relation is valid in every dimension. Hence inside a cuboid the dimensions of the feature space are supposed to be independent.

The parameters are set with problem independent default values, which must not modified without any reason. Parameters are only important during a learning sequence. They do not influence on the behavior of `enquire_class_box`.

Default:

'min_samples_for_split' = 80, *'split_error' = 0.1*, *'prop_constant' = 0.25*.

Parameters

▷ **`ClassifHandle`** (input_control) . . . . . class_box ⇝ *handle*
Handle of the classifier.

▷ **`Flag`** (input_control) . . . . . string ⇝ *string*
Name of the wanted parameter.
Default: 'split_error'
Suggested values: `Flag` ∈ {'min_samples_for_split', 'split_error', 'prop_constant'}

▷ **`Value`** (input_control) . . . . . number ⇝ *real* / integer
Value of the parameter.
Default: 0.1

Result

`read_sampset` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ClassifHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_class_box`, `enquire_class_box`

Possible Successors

`learn_class_box`, `test_sampset_box`, `write_class_box`, `close_class_box`, `clear_sampset`

See also

`enquire_class_box`, `get_class_box_param`, `learn_class_box`

Module

Foundation

`test_sampset_box` (: : ClassifHandle, SampKey : Error)

Classify a set of arrays.

`test_sampset_box` is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

In contrast to learn_sampset_box there is not a learning here. Typically you use test_sampset_box to classify independent test data. Error gives you information about the applicability of the learned training set on new examples.

Parameters

▷ **ClassifHandle** (input_control) . . . . . class_box ⇝ *handle*
Handle of the classifier.

▷ **SampKey** (input_control) . . . . . feature_set ⇝ *handle*
Key of the test data.

▷ **Error** (output_control) . . . . . real ⇝ *real*
Error during the assignment.

Result

test_sampset_box returns 2 (H_MSG_TRUE). An exception is raised, if if key SampKey does not exist or problems occur while opening the file.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_class_box, learn_class_box, set_class_box_param

Possible Successors

enquire_class_box, learn_class_box, write_class_box, close_class_box, clear_sampset

See also

enquire_class_box, learn_class_box, learn_sampset_box, read_sampset

Module

Foundation

write_class_box (: : ClassifHandle, FileName :)

Save a classifier in a file.

write_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM, KNN or GMM operators instead. The operator will be removed with HALCON 25.11.

write_class_box writes the classifier ClassifHandle to the file given by FileName. The classifier can be read with read_class_box. The default HALCON file extension for the box classifier is 'gbc'.

Attention

If a file with this name exists, it is overwritten without a warning. The file can not be edited.

Parameters

▷ **ClassifHandle** (input_control) . . . . . class_box ⇝ *handle*
Handle of the classifier.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the file which contains the written data.
File extension: .gbc

Result

write_class_box returns 2 (H_MSG_TRUE). An exception is raised if it was not possible to open file FileName.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_class_box, enquire_class_box, learn_class_box, test_sampset_box

Possible Successors

close_class_box, clear_sampset

See also

create_class_box, read_class_box

Module

Foundation

17.3 Control

ifelse (: : Condition :)

Conditional statement with alternative.

ifelse is obsolete and is only provided for reasons of backward compatibility. if should be used instead.

ifelse is a conditional statement with an alternative. If the condition is true (i.e., not *0*), all expressions and calls between the head and operator else are performed. If the condition is false (i.e., *0*) the part between else and endif is executed. Note that the operator is called ifelse and it is displayed as if in the program text area.

Parameters

▷ **Condition** (input_control) ..... integer ⇝ *integer*
Condition for the if statement.
Default: 1

Result

If the condition is correct ifelse (as operator) returns 2 (H_MSG_TRUE). Otherwise, an exception is raised and an error code returned.

Alternatives

if

See also

else, elseif, for, while, repeat, until

Module

Foundation

17.4 DL Classification

The workflow and operators described in this chapter are obsolete and only provided for reasons backward compatibility. The operators will be removed with HALCON 25.05. New applications should use the workflow and operators described in Deep Learning / Classification.

This chapter explains how to use classification based on deep learning, both for the training and inference phases.

General Workflow

This section describes the main steps and the workflow for classification with Deep Learning using the obsolete workflow.

Prepare the Network and the Data This part is about how to prepare and preprocess your data.

1. First, a pretrained network has to be read using the operator

- read_dl_classifier.

This operator is used as well when you want to read your own trained networks, after you saved them with write_dl_classifier.

2. To read the data for your deep learning classification training the procedure

   - read_dl_classifier_data_set

   is available. Using this procedure you can get a list of image file paths and their respective labels (the ground truth labels) as well as a list of the unique classes, to which at least one of the listed images belongs.

3. The network will impose several requirements on the images, as the image dimensions and the gray value range. The default values are listed in read_dl_classifier. These are the values with which the networks have been pretrained. The network architectures allow different image dimensions, which can be set with set_dl_classifier_param, but depending on the network a change may make a retraining necessary. The actually set values can be retrieved with

   - get_dl_classifier_param.

   The procedure preprocess_dl_classifier_images provides great guidance on how to implement such a preprocessing stage. We recommend to preprocess and store all images used for the training before starting the classifier training, since this speeds up the training significantly.

4. Next, we recommend to split the dataset into three distinct datasets which are used for training, validation, and testing, see the section "Data" in the chapter Deep Learning. This can be achieved using the procedure

   - split_dl_classifier_data_set.

5. You need to specify the *'classes'* (determined before by use of read_dl_classifier_data_set) you want to differentiate with your classifier. For this, the operator

   - set_dl_classifier_param

   is available.

   This operator can also be used to set hyperparameters, which are important for training, e.g., *'batch_size'* and *'learning_rate'*. For a more detailed explanation, see the chapter Deep Learning and the documentation of set_dl_classifier_param.

Train the Network and Evaluate the Training Progress Once your network is set up and your data prepared it is time to train the classifier for your specific task.

1. Set the hyperparameters used for training with the operator

   - set_dl_classifier_param.

   For an overview of possible hyperparameters, see the documentation of set_dl_classifier_param. Additional explanations can be found in the chapter Deep Learning.

2. To train the classifier the operator

   - train_dl_classifier_batch

   is available. The intermediate training results are stored in the output handle.

   As the name of train_dl_classifier_batch indicates, this operator processes a batch of data (images and ground truth labels) at once. We iterate through our training data in order to train the classifier successively with train_dl_classifier_batch. You can repeat this process multiple times and iterate over so many training epochs until you are satisfied with the training result.

3. To know how well the classifier learns the new task, the procedure

   - plot_dl_classifier_training_progress

   is provided. With it you can plot the classification errors during training. To compute the input necessary for the visualization, the procedures

   - select_percentage_dl_classifier_data,
   - apply_dl_classifier_batchwise, and

- evaluate_dl_classifier

are available. With them you can reduce the number of images used for this classification validation, apply the classifier on the selected data and compute, for example, the top-1 error.

Apply and Evaluate the Final Classifier Your classifier is trained for your task and ready to be applied. But before deploying in the real world you should evaluate how well the classifier performs on basis of your test data.

1. To apply the classifier on a set containing an arbitrary number of images, use the operator

   - apply_dl_classifier.

   The runtime of this operator depends on the number of batches needed for the given image set.

   The results are returned in a handle.

   To retrieve the predicted classes and confidences, use the operator

   - get_dl_classifier_result.

2. Now it is time to evaluate these results. The performance of the classifier can be evaluated as during the training with evaluate_dl_classifier.

   To visualize and analyze the classifier quality, the confusion matrix is a useful tool (see below for an explanation). For this, you can use the procedures

   - gen_confusion_matrix
   - gen_interactive_confusion_matrix.

   The interactive procedure gives you the possibility to select examples of a specific category, but it does not work with exported code.

   Additionally, after applying the classifier on a set of data, you can use the procedure

   - get_dl_classifier_image_results

   to display and return images according to certain criteria, e.g., wrongly classified ones. Then, you might want to use this input for the procedure

   - dev_display_dl_classifier_heatmap,

   to display a heatmap of the input image, with which you can analyze which regions of the image are relevant for the classification result.

Inference Phase When your classifier is trained and you are satisfied with its performance, you can use it to classify new images. For this, you simply preprocess your images according to the network requirements (i.e., the same way as you did for your dataset used for training the classifier) and apply the classifier using

- apply_dl_classifier.

Data for Classification

We distinguish between data used for training and data for inference. Latter one consists of bare images. But for the former one you already know to which class the images belong and provide this information over the corresponding labels.

The training data is used to train a classifier for your specific task. With the aid of this data the classifier can learn which classes are to be distinguished and how their representatives look like. In classification, the image is classified as a whole. Therefore, the training data consists of images and their ground truth labels, thus the class you say this image belongs to. Note that the images should be as representative as possible for your task. There are different possible ways, how to store and retrieve the ground truth labels. The procedure read_dl_classifier_data_set supports the following sources of the ground truth label for an image:

- The last directory name containing the image
- The file name.

For training a classifier, we use a technique called transfer learning (see the chapter Deep Learning). For this, you need less resources, but still a suitable set of data which is generally in the order of hundreds to thousands per class. While in general the network should be more reliable when trained on a larger dataset, the amount of data needed for training also depends on the complexity of the task. You also want enough training data to split it into three subsets, which are preferably independent and identically distributed, see the section "Data" in the chapter Deep Learning.

Regardless of the application, the network poses requirements on the images regarding the image dimensions, the gray value range, and the type. The specific values depend on the network itself and can be queried with `get_dl_classifier_param`. You can find guidance on how to implement such a preprocessing stage by the procedure `preprocess_dl_classifier_images`.

apply_dl_classifier (
Images : : DLClassifierHandle : DLClassifierResultHandle)

Infer the class affiliations for a set of images using a deep-learning-based classifier.

`apply_dl_classifier` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use the general CNN-based operator `apply_dl_model` instead.

`apply_dl_classifier` applies the deep-learning-based classifier given by `DLClassifierHandle` on the set of input images stored in the input object tuple `Images`. It returns the results in the result handle `DLClassifierResultHandle`. For information how to retrieve the corresponding results stored in `DLClassifierResultHandle`, please refer to the documentation of the operator `get_dl_classifier_result`.

The tuple `Images` can contain an arbitrary number of images to be processed in one operator call and is generally independent of the classifier parameter *'batch_size'*. Please notice that this only holds for `apply_dl_classifier` and not for `train_dl_classifier_batch`. This is because `apply_dl_classifier` always classifies a batch with up to *'batch_size'* images simultaneously, whether filled up or not. So in case the number of images in the set `Images` is larger than *'batch_size'*, `apply_dl_classifier` iterates over the necessary number of batches internally. For a `Images` tuple with less than *'batch_size'* images, it is padded to a full batch which means that the runtime of processing for a given batch is independent of whether it is filled up or just consists of a single image. Additionally, if fewer images than *'batch_size'* are classified in one operator call, the network still requires the same amount of memory as for a full batch. Therefore, it is recommended to adapt the *'batch_size'* according to the number of images to be processed in one operator call for greater efficiency. The current value of *'batch_size'* can be retrieved using `get_dl_classifier_param`.

Note that the images must be processed before feeding them into the operator `apply_dl_classifier` in order to have the correct size, gray value range, number of channels and type. We would like to stress the image type: the images must be of type 'real'. For a possibly necessary conversion the operator `convert_image_type` is available. The procedure `preprocess_dl_classifier_images` provides great guidance on how to implement such a preprocessing stage.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Attention

To run this operator, cuDNN and cuBLAS are required when *'runtime'* is set to *'gpu'*, see `set_dl_classifier_param`. For further details, please refer to the "`Installation Guide`", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods".

Parameters

- ▷ **`Images`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* real
  Tuple of input images.
- ▷ **`DLClassifierHandle`** (input_control) ..... dl_classifier ⇝ *handle*
  Handle of the deep-learning-based classifier.
- ▷ **`DLClassifierResultHandle`** (output_control) ..... dl_classifier_result ⇝ *handle*
  Handle of the deep learning classification results.

Result

If the parameters are valid, the operator `apply_dl_classifier` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`read_dl_classifier`, `train_dl_classifier_batch`, `set_dl_classifier_param`

Possible Successors

`get_dl_classifier_result`, `clear_dl_classifier`

Alternatives

`apply_dl_model`, `classify_class_mlp`, `classify_class_svm`

Module

Deep Learning Enhanced

clear_dl_classifier (: : DLClassifierHandle :)

Clear a deep-learning-based classifier.

`clear_dl_classifier` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use the common used CNN-based operator `clear_dl_model` instead.

`clear_dl_classifier` clears the handle of the deep-learning-based classifier given by `DLClassifierHandle`, that was created with `read_dl_classifier`, and frees all memory required for the classifier. After calling `clear_dl_classifier`, the classifier can no longer be used. The handle `DLClassifierHandle` becomes invalid.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

▷ **`DLClassifierHandle`** (input_control) ..... dl_classifier(-array) ⇝ *handle*
Handle of the deep-learning-based classifier.

Result

If the parameters are valid, the operator `clear_dl_classifier` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DLClassifierHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

read_dl_classifier, apply_dl_classifier, train_dl_classifier_batch

Module

Deep Learning Enhanced

clear_dl_classifier_result (: : DLClassifierResultHandle :)

Clear a handle containing the results of the deep-learning-based classification.

clear_dl_classifier_result is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05.

clear_dl_classifier_result releases the memory used to store the deep-learning-based classification results given in DLClassifierResultHandle. After calling clear_dl_classifier_result, the results can no longer be queried. The handle DLClassifierResultHandle becomes invalid.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

▷ **DLClassifierResultHandle** (input_control) ..... dl_classifier_result(-array) ⇝ *handle*
Handle of the deep learning classification results.

Result

If the parameters are valid, the operator clear_dl_classifier_result returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_dl_classifier_result, apply_dl_classifier

Module

Deep Learning Enhanced

clear_dl_classifier_train_result (: : DLClassifierTrainResultHandle :)

Clear the handle of a deep-learning-based classifier training result.

clear_dl_classifier_train_result is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05.

clear_dl_classifier_train_result clears an arbitrary number of training result handles DLClassifierTrainResultHandle of a deep-learning-based classifier. Each of these training result handles has previously been generated using train_dl_classifier_batch. All memory allocated in the handles is freed.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

▷ **DLClassifierTrainResultHandle** (input_control) ..... dl_classifier_train_result(-array) ⇝ *handle*
Handle of the training results from the deep-learning-based classifier.

Result

If the parameters are valid, the operator `clear_dl_classifier_train_result` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`train_dl_classifier_batch`, `get_dl_classifier_train_result`

Module

Deep Learning Professional

deserialize_dl_classifier (: : SerializedItemHandle : DLClassifierHandle)

Deserialize a deep-learning-based classifier.

`deserialize_dl_classifier` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use the common CNN-based operator `deserialize_dl_model`.

`deserialize_dl_classifier` deserializes a deep-learning-based classifier, that was serialized by `serialize_dl_classifier`.

The operator acts the same as `read_dl_classifier` except that the input is a serialized item instead of a file. For a detailed description please refer to the documentation of `read_dl_classifier`.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **DLClassifierHandle** (output_control) ..... dl_classifier ⇝ *handle*
Handle of the deep-learning-based classifier.

Result

If the parameters are valid, the operator `deserialize_dl_classifier` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_dl_classifier`

Possible Successors

`apply_dl_classifier`, `train_dl_classifier_batch`, `set_dl_classifier_param`, `get_dl_classifier_param`

Alternatives

deserialize_dl_model

See also

serialize_dl_classifier

Module

Deep Learning Enhanced

get_dl_classifier_param (: : DLClassifierHandle, GenParamName : GenParamValue)

Return the parameters of a deep-learning-based classifier.

get_dl_classifier_param is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use common CNN-based operator get_dl_model_param instead.

get_dl_classifier_param returns the parameter values GenParamValue of GenParamName of the neural network DLClassifierHandle.

The hyperparameters and network parameters can be set with the operator set_dl_classifier_param, in whose reference entry they are described in detail. With get_dl_classifier_param you can query all these values.

Additionally, there are parameters defined by the network which are read-only. These parameters are:

'image_range_min': Minimum gray value.

'image_range_max': Maximum gray value.

The precise values for these parameters and the default parameters for the image dimension depend on the concrete network, see read_dl_classifier.

Every image that is fed into the network must be present according to the parameters defining the image properties. To preprocess images accordingly, the procedure preprocess_dl_classifier_images is available.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

▷ **DLClassifierHandle** (input_control) ..... dl_classifier ⇝ *handle*
Handle of the deep-learning-based classifier.

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: 'gpu'
List of values: GenParamName ∈ {'batch_size', 'batch_size_multiplier', 'classes', 'gpu', 'image_dimensions', 'image_num_channels', 'image_height', 'image_range_max', 'image_range_min', 'image_width', 'learning_rate', 'momentum', 'runtime', 'runtime_init', 'weight_prior'}

▷ **GenParamValue** (output_control) ..... attribute.name(-array) ⇝ *integer* / string / real
Value of the generic parameter.

Result

If the parameters are valid, the operator get_dl_classifier_param returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_dl_classifier, set_dl_classifier_param

Possible Successors

train_dl_classifier_batch, apply_dl_classifier

Alternatives

get_dl_model_param

See also

set_dl_classifier_param

Module

Deep Learning Enhanced

get_dl_classifier_result (: : DLClassifierResultHandle, Index, GenResultName : GenResultValue)

Retrieve classification results inferred by a deep-learning-based classifier.

get_dl_classifier_result is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05.

get_dl_classifier_result allows the user to return classification results. These results are stored in the handle DLClassifierResultHandle and have been obtained from the operator apply_dl_classifier.

The input parameter GenResultName names the sort of result asked and Index identifies for which image the result is to be returned. The respective value is given in GenResultValue.

Concretely, GenResultName can attain the following values:

'confidences': return the confidences values for the selected image Index.

'predicted_classes': return the names of the inferred classes for the selected image Index.

'predicted_class_indices': return the indices of the inferred classes for the selected image Index. The class indices correspond to the positions within the array of classes (counting from *0*).

The input parameter Index can attain the value *'all'*, in which case get_dl_classifier_result returns the confidence value, respectively class, for the best candidate (prediction with the highest confidence value) for each image of the input batch.

Otherwise, Index must be an integer selecting one of the input images in the batch, in which case get_dl_classifier_result returns all output classes, respectively, confidences sorted by decreasing confidence value. This is useful in order to retrieve the best K candidates for a given input. As we start indexing at *0*, Index must have a value between *0* and the size of the batch of input images minus one.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

- ▷ **DLClassifierResultHandle** (input_control) ..... dl_classifier_result ⇝ *handle*
  Handle of the deep learning classification results.
- ▷ **Index** (input_control) ..... number ⇝ *string* / integer
  Index of the image in the batch.
  Default: 'all'
- ▷ **GenResultName** (input_control) ..... attribute.name(-array) ⇝ *string*
  Name of the generic parameter.
  Default: 'predicted_classes'
  List of values: GenResultName ∈ {'predicted_classes', 'predicted_class_indices', 'confidences'}
- ▷ **GenResultValue** (output_control) ..... attribute.value(-array) ⇝ *real* / string / integer
  Value of the generic parameter, either the confidence values, the class names or class indices.

Result

If the parameters are valid, the operator `get_dl_classifier_result` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`apply_dl_classifier`

Possible Successors

`clear_dl_classifier_result`

Module

Deep Learning Enhanced

get_dl_classifier_train_result (: : DLClassifierTrainResultHandle, GenParamName : GenParamValue)

Return the results for the single training step of a deep-learning-based classifier.

`get_dl_classifier_train_result` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05.

`get_dl_classifier_train_result` returns the value `GenParamValue` of the result `GenParamName` of a deep learning training handle `DLClassifierTrainResultHandle`.

The options for `GenParamValue` are the following:

'loss' **or** *'total_loss'***:** Current value of the loss function. See `train_dl_classifier_batch` for a detailed explanation of the loss function.

*'mll_loss'***:** Current value of the loss function, i.e. *'loss'*, without *'regularization_loss'*.

*'regularization_loss'***:** Current value of the regularization loss function, i.e. *'loss'* without *'mll_loss'*.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

▷ **`DLClassifierTrainResultHandle`** (input_control) . . . . . dl_classifier_train_result ⇝ *handle*
Handle of the training results from the deep-learning-based classifier.

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: 'loss'
List of values: `GenParamName` ∈ {'loss', 'mll_loss', 'regularization_loss', 'total_loss'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.name(-array) ⇝ *real* / integer / string
Value of the generic parameter.

Result

If the parameters are valid, the operator `get_dl_classifier_train_result` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_dl_classifier_batch

Possible Successors

clear_dl_classifier_train_result

Module

Deep Learning Professional

read_dl_classifier (: : FileName : DLClassifierHandle)

Read a deep-learning-based classifier from a file.

read_dl_classifier is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use the common CNN-based operator read_dl_model.

The operator read_dl_classifier reads a neural network written by write_dl_classifier. As a result, the handle DLClassifierHandle is returned.

HALCON provides pretrained neural networks. These neural networks are good starting points to train a custom classifier for image classification. They have been pretrained on a large image dataset. The provided pretrained neural networks are:

'pretrained_dl_classifier_compact.hdl': This neural network is designed to be memory and runtime efficient.

This classifier expects the images to be of the type real. Additionally, the network is designed for certain image properties. The corresponding values can be retrieved with get_dl_classifier_param. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

This network does not contain any fully connected layer. The network architecture allows changes concerning the image dimensions, but requires a minimum *'image_width'* and *'image_height'* of 15 pixels.

'pretrained_dl_classifier_enhanced.hdl': This neural network has more hidden layers than *'pretrained_dl_classifier_compact.hdl'* and is therefore assumed to be better suited for more complex classification tasks. But this comes at the cost of being more time and memory demanding. As a result, e.g., in comparison to the above compact network, the batch size has to be decreased network during the training, see set_dl_classifier_param.

This classifier expects the images to be of the type real. Additionally, the network is designed for certain image properties. The corresponding values can be retrieved with get_dl_classifier_param. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The network architecture allows changes concerning the image dimensions, but requires a minimum *'image_width'* and *'image_height'* of 47 pixels. There is no maximum image size, but large image sizes will increase the memory demand and the runtime significantly. Changing the image size will reinitialize the weights of the fully connected layers and therefore makes a retraining necessary.

'pretrained_dl_classifier_resnet18.hdl': As the neural network *'pretrained_dl_classifier_enhanced.hdl'*, this classifier is suited for more complex tasks. But its structure differs, bringing the advantage of making the training more stable and being internally more robust. Compared to the neural network *'pretrained_dl_classifier_resnet50.hdl'* it is less complex and has faster inference times.

This classifier expects the images to be of the type `real`. Additionally, the network is designed for certain image properties. The corresponding values can be retrieved with `get_dl_classifier_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The network architecture allows changes concerning the image dimensions, but a minimum *'image_width'* and *'image_height'* of 32 pixels is recommended. There is no maximum image size, but large image sizes will increase the memory demand and the runtime significantly. Despite the fully connected layer a change of the image size does not lead to a reinitialization of the weights.

'pretrained_dl_classifier_resnet50.hdl': As the neural network *'pretrained_dl_classifier_enhanced.hdl'*, this classifier is suited for more complex tasks. But its structure differs, bringing the advantage of making the training more stable and being internally more robust.

This classifier expects the images to be of the type `real`. Additionally, the network is designed for certain image properties. The corresponding values can be retrieved with `get_dl_classifier_param`. Here we list the default values with which the classifier has been trained:

'image_width': 224
'image_height': 224
'image_num_channels': 3
'image_range_min': -127.0
'image_range_max': 128.0

The network architecture allows changes concerning the image dimensions, but a minimum *'image_width'* and *'image_height'* of 32 pixels is recommended. There is no maximum image size, but large image sizes will increase the memory demand and the runtime significantly. Despite the fully connected layer a change of the image size does not lead to a reinitialization of the weights.

The values listed above are the default image dimensions and gray value range for the networks and these are the values with which the classifiers have been trained. The network architectures allow different image sizes which can be set with `set_dl_classifier_param`. For networks with at least one fully connected layer such a change makes a retraining necessary. Networks without fully connected layers are directly applicable to different image sizes. However, images with a size differing from the size with which the classifier has been trained are likely to show a reduced classification accuracy.

The actually configured dimensions can be queried by `get_dl_classifier_param`. Every image that is fed into a network must be present according to the required dimensions. To adjust images accordingly, the procedure `preprocess_dl_classifier_images` is available.

Typically it is easier, faster and better to retrain a pretrained classifier for a given classification problem. A pretrained classifier has already learned good general purpose features. To retrain the network for a custom problem, the new *'classes'* of the classifier have to be set with `set_dl_classifier_param`.

The neural network is loaded from the file `FileName`. This file is hereby searched in the directory ($HALCONROOT/dl/) as well as in the currently used directory.

Please note that the runtime specific parameter *'gpu'* of the classifier is not read from file. Instead it is initialized with its default value (see `set_dl_classifier_param`).

The default HALCON file extension for deep learning classifiers is `'.hdl'`.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

▷ **`FileName`** (input_control) . . . . . filename.read ⇝ *string*
File name.
Default: 'pretrained_dl_classifier_compact.hdl'
List of values: `FileName` ∈ {'pretrained_dl_classifier_compact.hdl', 'pretrained_dl_classifier_enhanced.hdl', 'pretrained_dl_classifier_resnet18.hdl', 'pretrained_dl_classifier_resnet50.hdl'}
File extension: `.hdl`

▷ **DLClassifierHandle** (output_control) ..... dl_classifier ⇝ *handle*
Handle of the deep learning classifier.

Result

If the indicated file is available and the format is correct, the operator `read_dl_classifier` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`set_dl_classifier_param`, `get_dl_classifier_param`, `apply_dl_classifier`, `train_dl_classifier_batch`

Alternatives

`read_dl_model`, `read_class_mlp`, `read_class_svm`

Module

Deep Learning Enhanced

serialize_dl_classifier

```
serialize_dl_classifier ( : : DLClassifierHandle : SerializedItemHandle )
```

Serialize a deep-learning-based classifier.

`serialize_dl_classifier` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use the common CNN-based operator `serialize_dl_model`.

`serialize_dl_classifier` serializes a deep-learning-based classifier (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The classifier is defined by the handle `DLClassifierHandle`. The serialized classifier is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_dl_classifier`.

The operator acts the same as `write_dl_classifier` except that the output is a serialized item instead of a file. For a detailed description please refer to the documentation of `write_dl_classifier`.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

▷ **DLClassifierHandle** (input_control) ..... dl_classifier ⇝ *handle*
Handle of the deep-learning-based classifier.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_dl_classifier` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

train_dl_classifier_batch

Possible Successors

apply_dl_classifier, train_dl_classifier_batch, fwrite_serialized_item, send_serialized_item, set_dl_classifier_param, get_dl_classifier_param

Alternatives

serialize_dl_model

See also

deserialize_dl_classifier, apply_dl_classifier, train_dl_classifier_batch

Module

Deep Learning Enhanced

set_dl_classifier_param (: : DLClassifierHandle, GenParamName, GenParamValue :)

Set the parameters of a deep-learning-based classifier.

set_dl_classifier_param is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use the common CNN-based operator set_dl_model_param instead.

set_dl_classifier_param sets the parameters and hyperparameters GenParamName of the neural network DLClassifierHandle to the values GenParamValue.

The pretrained classifiers are trained for their default image dimensions, see read_dl_classifier.

The network architectures allow different image dimensions. But for networks with at least one fully connected layer such a change makes a retraining necessary. Networks without fully connected layers are directly applicable to different image sizes. However, images with a size differing from the size with which the classifier has been trained, are likely to show a reduced classification accuracy.

GenParamName can attain the following values:

'batch_size': Number of images (and corresponding labels) in a batch that is transferred to device memory. The batch of images which are processed simultaneously in a single training iteration contains a number of images which is equal to *'batch_size'* times *'batch_size_multiplier'*. Please refer to train_dl_classifier_batch for further details. The parameter *'batch_size'* is stored in the pretrained classifier. Per default, the *'batch_size'* is set such that a training of the pretrained classifier with up to 100 classes can be easily performed on a device with 8 gigabyte of memory. For the pretrained classifiers, the default values are hence given as follows:

pretrained classifier	default value of *'batch_size'*
'pretrained_dl_classifier_compact.hdl'	*160*
'pretrained_dl_classifier_enhanced.hdl'	*96*
'pretrained_dl_classifier_resnet18.hdl'	*24*
'pretrained_dl_classifier_resnet50.hdl'	*23*

For inference, the *'batch_size'* can be generally set independently from the number of input images. See apply_dl_classifier for details on how to set this parameter for greater efficiency.

'batch_size_multiplier': Multiplier for *'batch_size'* to enable training with larger numbers of images in one step which would otherwise not be possible due to GPU memory limitations. For detailed information see train_dl_classifier_batch. This model parameter does not have any impact during evaluation and inference. For the pretrained classifiers, the default value of *'batch_size_multiplier'* is set to 1.

'classes': Tuple of labels corresponding to the classes of objects which are to be recognized. The order of the class names remains unchanged after the setting.

'gpu': Identifier of the GPU where the training and inference operators (train_dl_classifier_batch and apply_dl_classifier) are executed. Per default, the first available GPU is used. get_system with *'cuda_devices'* can be used to retrieve a list of available GPUs. Pass the index in this list to *'gpu'*.

'image_width': Width of the images the network will process. The default value is given by the network, see read_dl_classifier.

'image_height': Height of the images the network will process. The default value is given by the network, see read_dl_classifier.

'image_num_channels': Number of channels of the images the network will process. Possible are one channel (gray value image), or three channels (three-channel image). The default value is given by the network, see read_dl_classifier. Changing to a single channel image modifies the network configuration. This process removes the color information contained in certain layers and is not invertible.

'image_dimensions': Tuple containing the image dimensions *'image_width'*, *'image_height'*, and number of channels *'image_num_channels'*. The default values are given by the network, see read_dl_classifier. Concerning the number of channels, the values one (gray value image), or three (three-channel image) are possible. Changing to a single channel image modifies the network configuration. This process removes the color information contained in certain layers and is not invertible.

'learning_rate': Initial value of the factor determining the gradient influence during training. Please refer to train_dl_classifier_batch for further details. The default value depends on the classifier.

'momentum': When updating the arguments of the loss function, the hyperparameter *'momentum'* specifies to which extent previous updating vectors will be added to the current updating vector. Please refer to train_dl_classifier_batch for further details. Per default, the *'momentum'* is set to *0.9*.

'runtime': Defines the device on which the operators will be executed. Per default, the *'runtime'* is set to *'gpu'*.

'cpu': The operator apply_dl_classifier will be executed on the CPU, whereas the operator train_dl_classifier_batch is not executable.
In case the GPU has been used before, CPU memory is initialized, and if necessary values stored on the GPU memory are moved to the CPU memory.
On Intel or AMD architectures the *'cpu'* runtime uses OpenMP for the parallelization of apply_dl_classifier, where per default, all threads available to the OpenMP runtime are used. You may use the set_system parameter *'thread_num'* to specify the number of threads.
On Arm architectures the *'cpu'* runtime uses a global thread pool. You may specify the number of threads with the set_system parameter *'thread_num'*.
For both architectures mentioned above, it is not possible to specify a thread-specific number of threads (via the parameter *'tsp_thread_num'* of the operator set_system).

'gpu': The GPU memory is initialized and the corresponding handle created. The operators apply_dl_classifier and train_dl_classifier_batch will be executed on the GPU. For the specific requirements please refer to the HALCON "Installation Guide".

'runtime_init': If called with *'immediately'*, the GPU memory is initialized and the corresponding handle created. Otherwise this is done on demand, which may result in significantly larger execution times for the first call of apply_dl_classifier or train_dl_classifier_batch. If *'gpu'* or *'batch_size'* is changed with subsequent calls of set_dl_classifier_param, the GPU memory is reinitialized.

Note, this parameter has no effect if running on CPUs, thus if *'runtime'* is set to *'cpu'*.

'weight_prior': Regularization parameter $\alpha \geq 0.0$ used for l_2 regularization of the loss function. Regularization is helpful in the presence of overfitting during the classifier training. If the hyperparameter *'weight_prior'* is non-zero, the regularization term given below is added to the loss function (see also train_dl_classifier_batch)

$$E_\alpha(w) = \frac{\alpha}{2} \sum_{k=0}^{K-1} |w_k|^2$$

Here the index k runs over all weights of the network, except for the biases which are not regularized. The regularization term $E_\alpha(w)$ generally penalizes large weights, thus pushing the weights towards zero, which effectively reduces the complexity of the model. Simply put: Regularization favors simpler models that are less likely to learn noise in the data and generalize better. In case the classifier overfits the data, it is strongly recommended to try different values for the parameter *'weight_prior'* to improve the generalization properties of the neural network. Choosing its value is a trade-off between the models ability to generalize, overfitting, and underfitting. If α is too small, the model might overfit, if its too large the model might loose its ability to fit the data, because all weights are effectively zero. For finding an ideal value for α, we recommend a cross-validation, i.e. to perform the training for a range of values and choose the value that results in the best

validation error. For typical applications, we recommend testing the values for *'weight_prior'* on a logarithmic scale between $1.0, ..., 0.0001$. If the training takes a very long time, one might consider performing the hyperparameter optimization on a reduced amount of data. The default value depends on the classifier.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Attention

To successfully set *'gpu'* parameters, cuDNN and cuBLAS are required, i.e., to set the parameter `GenParamName` *'runtime'* to *'gpu'* or to set the `GenParamName` *'gpu'*. For further details, please refer to the "`Installation Guide`", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods".

Parameters

▷ **`DLClassifierHandle`** (input_control) ..... dl_classifier ⇝ *handle*
Handle of the deep-learning-based classifier.

▷ **`GenParamName`** (input_control) ..... attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: 'classes'
List of values: `GenParamName` ∈ {'batch_size', 'batch_size_multiplier', 'classes', 'gpu', 'image_dimensions', 'image_num_channels', 'image_height', 'image_width', 'learning_rate', 'momentum', 'runtime', 'runtime_init', 'weight_prior'}

▷ **`GenParamValue`** (input_control) ..... attribute.value(-array) ⇝ *string* / real / integer
Value of the generic parameter.
Default: ['class_1','class_2','class_3']
Suggested values: `GenParamValue` ∈ {1, 2, 3, 50, 0.001, 'cpu', 'gpu', 'immediately'}

Result

If the parameters are valid, the operator `set_dl_classifier_param` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_dl_classifier`

Possible Successors

`get_dl_classifier_param`, `apply_dl_classifier`, `train_dl_classifier_batch`

Alternatives

`set_dl_model_param`

See also

`get_dl_classifier_param`

Module

Deep Learning Enhanced

`train_dl_classifier_batch` (`BatchImages : : DLClassifierHandle, BatchLabels : DLClassifierTrainResultHandle`)

Perform a training step of a deep-learning-based classifier on a batch of images.

`train_dl_classifier_batch` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use the common CNN-based operator `train_dl_model_batch`.

`train_dl_classifier_batch` performs a training step of the deep-learning-based classifier contained in `DLClassifierHandle`. The classifier handle `DLClassifierHandle` has to be read previously

using read_dl_classifier. In order to apply training steps, classes have to be specified using set_dl_classifier_param. Other hyperparameters such as the learning rate and the momentum are also important for a successful training. They are set using set_dl_classifier_param.

The training step is done on basis of a single batch of images from the training dataset, thus the images BatchImages with labels BatchLabels. The number of images within the batch needs to be a multiple of the *'batch_size'* where the parameter *'batch_size'* is limited by the amount of available GPU memory. In order to process more images in one training step, the classifier parameter *'batch_size_multiplier'* can be set to a value greater than 1. The number of images being passed to the training operator needs to be equal to *'batch_size'* times *'batch_size_multiplier'*. Note that a training step calculated for a batch and a *'batch_size_multiplier'* greater 1 is an approximation of a training step calculated for the same batch but with a *'batch_size_multiplier'* equal to 1 and an accordingly greater *'batch_size'*. As an example, the loss calculated with a *'batch_size'* of 4 and a *'batch_size_multiplier'* of 2 is usually not equal to the loss calculated with a *'batch_size'* of 8 and a *'batch_size_multiplier'* of 1, although the same number of images is used for training in both cases. However, the approximation generally delivers comparably good results, so it can be utilized if you wish to train with a larger number of images than your GPU allows. In some rare cases the approximation with a *'batch_size'* of 1 and an accordingly large *'batch_size_multiplier'* does not show the expected performance which for example can happen when the pretrained network *'pretrained_dl_classifier_resnet50.hdl'* is used. Setting the *'batch_size'* to a value greater than 1 can help to solve this issue.

Note that the images in BatchImages must fulfill certain conditions regarding, for example, the image size and gray value range, depending on the chosen network. Please have a look at read_dl_classifier and set_dl_classifier_param for more information. The labels in BatchLabels can be handed over as an array of strings, or as an array of indices corresponding to the position of the label within the array of classes (counting from *0*) set before via *'classes'* with set_dl_classifier_param. Information about the results of the training step as the value of the loss are stored in DLClassifierTrainResultHandle and can be accessed using get_dl_classifier_train_result.

Note that an epoch generally consists of a large number of batches and that a successful training involves many epochs. Therefore train_dl_classifier_batch has to be applied several times with different batches. For a more detailed explanation, we refer to Legacy / DL Classification.

During training, a nonlinear optimization algorithm minimizes the value of the loss function. The later one is determined based on the prediction of the neural network on the current batch of images. The algorithm used for optimization is stochastic gradient descent (SGD). It updates the layers' weights of the previous iteration w_t to the new values w_{t+1} at iteration $t+1$ as follows:

$$\begin{aligned} v^{t+1} &= \mu v^t - \lambda \nabla_w L\big(f(x, w^t)\big) \\ w^{t+1} &= w^t + v^{t+1}. \end{aligned}$$

Here, λ is the learning rate, μ for the momentum, and $f(x, w)$ for the classification result of the deep learning-based classifier which depends on the network weights w and the input batch x. The variable v_t is used to involve the influence of the momentum μ. The loss function used here is the Multinomial Logistic Loss in combination with a quadratic regularization term $E_\alpha(w)$,

$$L\big(f(x,w)\big) = -\frac{1}{N}\sum_{n=0}^{N-1}\langle y_n, \log\big(f(x_n,w)\big)\rangle + E_\alpha(w), \quad E_\alpha(w) = \frac{\alpha}{2}\sum_{k=0}^{K-1}|w_k|^2.$$

Here, y_n is a one-hot encoded target vector that encodes the label of the n-th image x_n of the batch x containing N-many images, and $\log(f(x_n, w))$ shall be understood to be a vector such that $\log$ is applied on each component of $f(x_n, w)$. The regularization term $E_\alpha(w)$ is a weighted l^2-norm involving all K weights except for biases. Its influence can be controlled through α. In the above formula, α denotes the hyperparameter *'weight_prior'* that can be set with set_dl_classifier_param. In order to gain more insight, you can retrieve the current value of the total loss function as well as individual contributions using get_dl_classifier_train_result.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Attention

The operator train_dl_classifier_batch internally calls functions that might not be deterministic.

Therefore, results from multiple calls of `train_dl_classifier_batch` can slightly differ, although the same input values have been used.

To run this operator, cuDNN and cuBLAS are required. For further details, please refer to the "`Installation Guide`", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods".

Parameters

- ▷ **`BatchImages`** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* real
  Images comprising the batch.
- ▷ **`DLClassifierHandle`** (input_control) ..... dl_classifier ⇝ *handle*
  Handle of the deep-learning-based classifier.
- ▷ **`BatchLabels`** (input_control) ..... string-array ⇝ *string* / integer
  Corresponding labels for each of the images.
  Default: []
  List of values: `BatchLabels` ∈ {[]}
- ▷ **`DLClassifierTrainResultHandle`** (output_control) ..... dl_classifier_train_result ⇝ *handle*
  Handle of the training results from the deep-learning-based classifier.

Result

If the parameters are valid, the operator `train_dl_classifier_batch` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`read_dl_classifier`, `set_dl_classifier_param`, `get_dl_classifier_param`

Possible Successors

`get_dl_classifier_train_result`, `apply_dl_classifier`, `clear_dl_classifier_train_result`, `clear_dl_classifier`

Alternatives

`train_dl_model_batch`, `train_class_mlp`, `train_class_svm`

See also

`apply_dl_classifier`

Module

Deep Learning Professional

`write_dl_classifier` (: : `DLClassifierHandle`, `FileName` :)

Write a deep-learning-based classifier in a file.

`write_dl_classifier` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use the common CNN-based operator `write_dl_model` instead.

`write_dl_classifier` writes the deep-learning-based classifier `DLClassifierHandle` to the file given by `FileName`.

`write_dl_classifier` is typically called after the classifier has been trained with `train_dl_classifier_batch`. The classifier can be read with `read_dl_classifier`.

Please note that the runtime specific parameter *'gpu'* of the classifier is not written.

The default HALCON file extension for deep learning classifiers is `'.hdl'`.

For an explanation of the concept of deep-learning-based classification see the introduction of chapter Deep Learning / Classification. The workflow involving this legacy operator is described in the chapter Legacy / DL Classification.

Parameters

▷ **DLClassifierHandle** (input_control) . . . . . dl_classifier ⇝ *handle*
Handle of the deep-learning-based classifier.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: `.hdl`

Result

If the parameters are valid, the operator `write_dl_classifier` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_dl_classifier`, `train_dl_classifier_batch`, `set_dl_classifier_param`

Alternatives

`write_dl_model`

Module

Deep Learning Enhanced

17.5 Develop

dev_map_par (: : :)

Open the dialog to specify the visualization parameters.

dev_map_par is obsolete and is only provided for reasons of backward compatibility. New applications should use dev_open_tool with ToolName set to *'visualization_parameters_dialog'* instead.

`dev_map_par` opens the `Visualization Parameter` dialog. This can also be achieved interactively via the menu `Visualization` ▷ `Set Parameters...` or the appropriate tool bar button. The dialog is used to configure the visualization parameters that are used to display iconic objects.

Attention

This operator is not supported for code export.

Example

```
read_image (Image, 'fabrik')
threshold (Image, Region, 128, 255)
dev_map_par ()
```

Result

`dev_map_par` returns always `2` (`H_MSG_TRUE`).

Possible Successors

`dev_unmap_par`

Module

Foundation

dev_map_prog (: : :)

Make the main window of HDevelop visible.

dev_map_prog is obsolete and is only provided for reasons of backward compatibility. New applications should use dev_open_tool with ToolName set to *'program_window'* instead.

dev_map_prog is used to re-show (map) the main window of HDevelop after it has been hidden (unmapped) by dev_unmap_prog.

Attention

This operator is not supported for code export.

Depending on the operating system or the window manager the execution of dev_map_prog will result only in a visible icon of the window. In this case it has to be opened by the user with mouse interaction.

Result

dev_map_prog always returns 2 (H_MSG_TRUE).

Possible Predecessors

dev_unmap_prog

Possible Successors

dev_unmap_prog

See also

dev_map_par, dev_map_var

Module

Foundation

dev_map_var (: : :)

Open the variable window of HDevelop.

dev_map_var is obsolete and is only provided for reasons of backward compatibility. New applications should use dev_open_tool with ToolName set to *'variable_window'* instead.

dev_map_var is used to re-show (map) the variable window on the screen after it has been hidden (unmapped) by dev_unmap_var.

Attention

This operator is not supported for code export.

Result

dev_map_var always returns 2 (H_MSG_TRUE).

Possible Predecessors

dev_unmap_var

Possible Successors

dev_unmap_var

See also

dev_map_par, dev_map_prog

Module

Foundation

dev_unmap_par (: : :)

Closes the dialog to specify the visualization parameters.

dev_unmap_par is obsolete and is only provided for reasons of backward compatibility. New applications should use dev_close_tool with ToolId set to *'visualization_parameters_dialog'* instead.

dev_unmap_par closes the Visualization Parameter dialog. This can also be achieved interactively by pressing the Close button on the dialog's title bar. The dialog can be opened again by dev_map_par.

Attention

This operator is not supported for code export.

Result

dev_unmap_par always returns 2 (H_MSG_TRUE).

Possible Successors

dev_map_par

See also

dev_map_par, dev_map_prog, dev_map_var

Module

Foundation

dev_unmap_prog (: : :)

Hide the main window.

dev_unmap_prog is obsolete and is only provided for reasons of backward compatibility. New applications should use dev_close_tool with ToolId set to *'program_window'* instead.

dev_unmap_prog hides in MDI mode the main window and in SDI mode the program window so that it is no longer visible. It can be shown again by dev_map_prog.

Attention: after hiding the main window there is no opportunity to enter and execute the dev_map_prog operator. So take care that dev_unmap_prog is only executed in continuous mode with a reachable dev_map_prog operator.

Attention

This operator is not supported for code export.

Result

dev_unmap_prog always returns 2 (H_MSG_TRUE).

Possible Successors

dev_map_prog, stop

See also

dev_map_par, dev_map_prog, dev_map_var

Module

Foundation

dev_unmap_var (: : :)

Hides the variable window.

dev_unmap_var is obsolete and is only provided for reasons of backward compatibility. New applications should use dev_close_tool with ToolId set to *'variable_window'* instead.

dev_unmap_var hides the variable window. This can also be achieved interactively by pressing the Close button on the window's title bar. The window can be opened again by dev_map_var or interactively via the Window menu.

Attention

This operator is not supported for code export.

Result

dev_unmap_var always returns 2 (H_MSG_TRUE).

Possible Successors

dev_map_var

See also

dev_map_par, dev_map_prog

Module

Foundation

17.6 Filters

gauss_image (Image : ImageGauss : Size :)

Smooth an image using discrete Gaussian functions.

gauss_image is obsolete and is only provided for reasons of backward compatibility. New applications should use the operator gauss_filter instead.

The operator gauss_image smoothes images using the discrete Gaussian. The smoothing effect increases with increasing filter size. The following filter sizes (Size) are supported (the sigma value of the Gauss function is indicated in brackets):

- *3* (0.65)
- *5* (0.87)
- *7* (1.43)
- *9* (1.88)
- *11* (2.31)

For border treatment the gray values of the images are reflected at the image borders.

The operator binomial_filter can be used as an alternative to gauss_image. binomial_filter is significantly faster than gauss_image. It should be noted that the mask size in binomial_filter does not lead to the same amount of smoothing as the mask size in gauss_image. Corresponding mask sizes can be determined based on the respective values of the Gaussian smoothing parameter sigma.

gauss_image can be executed on OpenCL devices for all supported image types. However, the OpenCL implementation can produce slightly different results from the scalar implementation.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

In order to be able to process gauss_image on an OpenCL device, Image must be at least 64 pixels in both width and height.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Image to be smoothed.

▷ **ImageGauss** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
Filtered image.

▷ **Size** (input_control) . . . . . integer ⇝ *integer*
Required filter size.
Default: 5
List of values: Size ∈ {3, 5, 7, 9, 11}

Example

```
gauss_image(Input,Gauss,7)
regiongrowing(Gauss,Segments,7,7,5,100)
```

Complexity

For each pixel: $O(Size * 2)$.

Result

If the parameter values are correct the operator gauss_image returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Module

Foundation

polar_trans_image (ImageXY : ImagePolar : Row, Column, Width, Height :)

Transform an image to polar coordinates

polar_trans_image is obsolete and is only provided for reasons of backward compatibility. New applications should use the operator polar_trans_image_ext instead.

polar_trans_image transforms an image in Cartesian coordinates to an image in polar coordinates. The size of the resulting image is selected with Width and Height. Width determines the angular resolution, while Height determines the resolution of the radius. Row and Column determine the center of the polar coordinate system in the original image ImageXY. This point is mapped to the upper row of ImagePolar.

A point (x',y') in the result image corresponds to the point (x,y) in the original image in the following manner:

$$\begin{aligned} x &= y' * \cos(2\pi(x'/result\_width)) + \texttt{Column} \\ y &= y' * \sin(2\pi(x'/result\_width)) + \texttt{Row}. \end{aligned}$$

polar_trans_image can be executed on an OpenCL device if the input image does not exceed the maximum size of image objects of the selected device. Due to numerical reasons, there can be slight differences in the output compared to the execution on the CPU.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **ImageXY** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Input image in Cartesian coordinates.

▷ **ImagePolar** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / real
Result image in polar coordinates.

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row coordinate of the center of the coordinate system.
Default: 100
Suggested values: Row ∈ {0, 10, 100, 200}
Value range: 0 ≤ Row ≤ 512
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column coordinate of the center of the coordinate system.
Default: 100
Suggested values: Column ∈ {0, 10, 100, 200}
Value range: 0 ≤ Column ≤ 512
Minimum increment: 1
Recommended increment: 1

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the result image.
Default: 314
Suggested values: Width ∈ {100, 200, 157, 314, 512}
Value range: 2 ≤ Width ≤ 512
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the result image.
Default: 200
Suggested values: Height ∈ {100, 128, 256, 512}
Value range: 2 ≤ Height ≤ 512
Minimum increment: 1
Recommended increment: 10

Example

```
read_image(Image,'monkey')
dev_display (Image)
polar_trans_image(Image,PolarImage,100,100,314,200)
dev_display (PolarImage)
```

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.

Module

Foundation

17.7 Graphics

clear_rectangle (: : WindowHandle, Row1, Column1, Row2, Column2 :)

Delete a rectangle on the output window.

clear_rectangle is obsolete and is only provided for reasons of backward compatibility.

clear_rectangle deletes all entries in the rectangle which is defined through the upper left corner (Row1,Column1) and the lower right corner (Row2,Column2). Deletion means that the specified rectangle is set to the background color (see open_window).

If you want to delete more than one rectangle, you may pass several rectangles, i.e., the parameters Row1, Column1, Row2 and Column2 are tuples.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row1** (input_control) . . . . . rectangle.origin.y(-array) ⇝ *integer*
Line index of upper left corner.
Default: 10
Value range: 0 ≤ Row1 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column1** (input_control) ..... rectangle.origin.x(-array) ⇝ *integer*
Column index of upper left corner.
Default: 10
Value range: 0 ≤ Column1 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Row2** (input_control) ..... rectangle.corner.y(-array) ⇝ *integer*
Row index of lower right corner.
Default: 118
Value range: 0 ≤ Row2 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1
Restriction: Row2 > Row1

▷ **Column2** (input_control) ..... rectangle.corner.x(-array) ⇝ *integer*
Column index of lower right corner.
Default: 118
Value range: 0 ≤ Column2 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1
Restriction: Column2 >= Column1

Example

```
* Erase a rectangle in the output window interactively:
draw_rectangle1(WindowHandle,L1,C1,L2,C2)
```

Result

If an output window exists and the specified parameters are correct clear_rectangle returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_draw, set_color, set_colored, set_line_width, set_rgb, set_hsi, draw_rectangle1

Alternatives

clear_window, disp_rectangle1

See also

open_window

Module

Foundation

disp_distribution (: : WindowHandle, Distribution, Row, Column, Scale :)

Displays a noise distribution.

disp_distribution is obsolete and is only provided for reasons of backward compatibility.

disp_distribution displays a distribution in the window. Row and Column define the position of the center of the graphic. Scale allows scaling of the graphic, whereby *1* means displaying all 256 values, *2* means displaying 128 values, *3* means displaying only 64 values, etc. Noise distributions can be generated with operations like gauss_distribution or noise_distribution_mean.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Distribution** (input_control) . . . . . real-array ⇝ *real*
Gray value distribution (513 values).

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row index of center.
Default: 256
Suggested values: Row ∈ {0, 64, 128, 256}
Value range: 0 ≤ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column index of center.
Default: 256
Suggested values: Column ∈ {0, 64, 128, 256}
Value range: 0 ≤ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Scale** (input_control) . . . . . integer ⇝ *integer*
Size of display.
Default: 1
Suggested values: Scale ∈ {1, 2, 3, 4, 5, 6}

Example

```
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
set_draw(WindowHandle,'fill')
set_color(WindowHandle,'white')
read_image(Image,'monkey')
draw_region(Region,WindowHandle)
noise_distribution_mean(Region,Image,21,Distribution)
disp_distribution (WindowHandle,Distribution,100,100,3)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_draw, set_color, set_colored, set_line_width, set_rgb, set_hsi, noise_distribution_mean, gauss_distribution

See also

gen_region_histo, gauss_distribution, noise_distribution_mean

Module

Foundation

disp_lut (: : WindowHandle, Row, Column, Scale :)

Graphical view of the look-up-table (lut).

disp_lut is obsolete and is only provided for reasons of backward compatibility.

disp_lut displays a graphical view of the look-up-table (lut) in the valid window. A look-up-table defines the transformation of image gray values to colors/gray levels on the screen. On most systems this can be modified.

disp_lut creates a graphical view of the table assigned to the output window with the logical window number WindowHandle and displays it for every basic color (red, green, blue). Row and Column define the position of the center of the graphic. Scale allows scaling of the graphic, whereby *1* means displaying all 256 values, *2* means displaying 128 values, *3* means displaying only 64 values, etc. Tables for monochrome-representations are displayed in the currently set color (see set_color, set_rgb, etc.). Tables for displaying "false colors" are viewed with red, green and blue for each color component.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Row of center of the graphic.
Default: 128
Value range: $0 \leq$ Row ≤ 511

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column of center of the graphic.
Default: 128
Value range: $0 \leq$ Column ≤ 511

▷ **Scale** (input_control) ..... integer ⇝ *integer*
Scaling of the graphic.
Default: 1
List of values: Scale $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
Value range: $0 \leq$ Scale ≤ 20

Result

disp_lut returns 2 (H_MSG_TRUE) if the hardware supports a look-up-table, the window is valid and the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_lut

See also

open_window, set_lut, set_fix, set_pixel, write_lut, get_lut, set_color

Module

Foundation

get_comprise (: : WindowHandle : Mode)

Get the output treatment of an image matrix.

get_comprise is obsolete and is only provided for reasons of backward compatibility.

get_comprise returns the output mode of gray values in the window WindowHandle that is used by disp_image and disp_color. The output mode defines whether only the gray values of objects are displayed or the whole image is displayed. The query is used for temporary mode settings, i.e., the current mode is queried, then overwritten with (set_comprise) and finally reset.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Mode** (output_control) ..... string ⇝ *string*
Display mode for images.

Result

`get_comprise` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`set_comprise`, `disp_image`, `disp_image`

See also

`set_comprise`, `disp_image`, `disp_color`

Module

Foundation

get_fix (: : WindowHandle : Mode)

Get mode of fixing of current look-up-table (lut).

`get_fix` is obsolete and is only provided for reasons of backward compatibility.

Use `get_fix` to get mode of fixing of current look-up-table (look-up-table of valid window) set before by `set_fix`.

Parameters

▷ **`WindowHandle`** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **`Mode`** (output_control) ..... string ⇝ *string*
Current Mode of fixing.

Result

`get_fix` returns `2` (`H_MSG_TRUE`) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`set_fix`, `set_pixel`, `set_rgb`

See also

`set_fix`

Module

Foundation

get_fixed_lut (: : WindowHandle : Mode)

Get fixing of "look-up-table" (lut) for "real color images"

`get_fixed_lut` is obsolete and is only provided for reasons of backward compatibility.

Get fixing of "look-up-table" (lut) for "real color images"

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Mode** (output_control) . . . . . string ⇝ *string*
Mode of fixing.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_fixed_lut

Module

Foundation

get_insert (: : WindowHandle : Mode)

Get the current display mode.

get_insert is obsolete and is only provided for reasons of backward compatibility.

get_insert returns the display mode of the output window. It is used by operators like disp_region, disp_line, disp_rectangle1, etc. The mode is set with set_insert. Possible values for Mode can be queried with the operator query_insert.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Mode** (output_control) . . . . . string ⇝ *string*
Display mode.

Result

get_insert returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_insert

Possible Successors

set_insert, disp_image

See also

set_insert, query_insert, disp_region, disp_line

Module

Foundation

get_line_approx (: : WindowHandle : Approximation)

Get the current approximation error for contour display.

get_line_approx is obsolete and is only provided for reasons of backward compatibility.

get_line_approx returns a parameter that controls the approximation error for region contour display in the window. It is used by the operator disp_region. Approximation controls the polygon approximation for contour display (0 ⇔ no approximation). Approximation is only important for displaying the contour of objects, especially if a line style was set with set_line_style.

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Approximation** (output_control) ..... string ⇝ *integer*
  Current approximation error for contour display.

Result

get_line_approx returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_line_approx, set_line_style, disp_region

See also

get_region_polygon, set_line_approx, set_line_style, disp_region

Module

Foundation

get_lut_style (: : WindowHandle : Hue, Saturation, Intensity)

Get modification parameters of look-up-table (lut).

get_lut_style is obsolete and is only provided for reasons of backward compatibility.

get_lut_style returns the values that were set with set_lut_style. Default is:

Hue: 0.0

Saturation 1.0

Intensity 1.0

Parameters

- ▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
  Window handle.
- ▷ **Hue** (output_control) ..... real ⇝ *real*
  Modification of color value.
- ▷ **Saturation** (output_control) ..... real ⇝ *real*
  Modification of saturation.
- ▷ **Intensity** (output_control) ..... real ⇝ *real*
  Modification of intensity.

Result

get_lut_style returns 2 (H_MSG_TRUE) if the window is valid and the parameter is correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_lut_style, set_lut

See also

set_lut_style

Module

Foundation

get_pixel (: : WindowHandle : Pixel)

Get the current color lookup table index.

get_pixel is obsolete and is only provided for reasons of backward compatibility.

get_pixel returns the internal coding of the output gray value or color, respectively, for the window. If the output mode is set to color(s) or gray value(s) (see set_color or set_gray), then the color- or gray values are transformed for internal use. The internal code is then used for (physical) screen display. The transformation depends on the mapping characteristics and the condition of the output device and can be different in different program runs. Don't confuse the term "pixel" with the term "pixel" in image processing (the other operator is get_grayval). Here a pixel is meant to be the color lookup table index.

With get_pixel it is possible to save the output mode without knowing whether colors or gray values are used. Pixel is set with the operator set_pixel.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Pixel** (output_control) ..... integer-array ⇝ *integer*
Index of the current color look-up table.

Result

get_part_style returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_pixel, disp_region, disp_image

See also

set_pixel

Module

Foundation

get_tshape (: : WindowHandle : TextCursor)

Get the shape of the text cursor.

get_tshape is obsolete and is only provided for reasons of backward compatibility.

get_tshape queries the shape of the text cursor for the output window. A new cursor shape is set by the operator set_tshape.

A text cursor marks the current position for text output (which can also be invisible). It is different from the mouse cursor (although both will be called ”cursor” if the context makes misconceptions impossible). The available shapes for the text cursor can be queried with query_tshape.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **TextCursor** (output_control) . . . . . string ⇝ *string*
Name of the current text cursor.

Result

get_tshape returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, set_font

Possible Successors

set_tshape, set_tposition, write_string, read_string, read_char

See also

set_tshape, query_tshape, write_string, read_string

Module

Foundation

move_rectangle (: : WindowHandle, Row1, Column1, Row2, Column2, DestRow, DestColumn :)

Copy inside an output window.

move_rectangle is obsolete and is only provided for reasons of backward compatibility.

move_rectangle copies all entries in the rectangle (Row1,Column1), (Row2,Column2) of the output window to a new position inside the same window. This position is determined by the upper left corner (DestRow, DestColumn). Regions of the window, which are “uncovered” through moving the rectangle, are set to the color of the background.

If you want to move several rectangles at once, you may pass parameters in form of tuples.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Row1** (input_control) . . . . . rectangle.origin.y(-array) ⇝ *integer*
Row index of upper left corner of the source rectangle.
Default: 0
Value range: $0 \leq$ Row1 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column1** (input_control) . . . . . rectangle.origin.x(-array) ⇝ *integer*
Column index of upper left corner of the source rectangle.
Default: 0
Value range: $0 \leq$ Column1 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Row2** (input_control) . . . . . rectangle.corner.y(-array) ⇝ *integer*
Row index of lower right corner of the source rectangle.
Default: 64
Value range: 0 ≤ `Row2` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column2** (input_control) . . . . . rectangle.corner.x(-array) ⇝ *integer*
Column index of lower right corner of the source rectangle.
Default: 64
Value range: 0 ≤ `Column2` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **DestRow** (input_control) . . . . . point.y(-array) ⇝ *integer*
Row index of upper left corner of the target position.
Default: 64
Value range: 0 ≤ `DestRow` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **DestColumn** (input_control) . . . . . point.x(-array) ⇝ *integer*
Column index of upper left corner of the target position.
Default: 64
Value range: 0 ≤ `DestColumn` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Example

```
* "Interactive" copy of a rectangle in the same window
draw_rectangle1(WindowHandle,L1,C1,L2,C2)
get_mbutton(WindowHandle,LN,CN,Button)
move_rectangle(WindowHandle,L1,C1,L2,C2,LN,CN)
```

Result

If the window is valid and the specified parameters are correct `move_rectangle` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_window`

Alternatives

`copy_rectangle`

See also

`open_window`

Module

Foundation

open_textwindow (: : Row, Column, Width, Height, BorderWidth, BorderColor, BackgroundColor, FatherWindow, Mode, Machine : WindowHandle)

Open a textual window.

open_textwindow is obsolete and is only provided for reasons of backward compatibility. New applications should use the operator open_window instead.

open_textwindow opens a new textual window, which can be used to perform textual input and output, as well as to perform output of images. All output (write_string, read_string, disp_region, etc.) is redirected to this window, if the same logical window number WindowHandle is used.

Besides the mouse cursor textual windows possess also a textual cursor which indicates the current writing position (more exactly: the lower left corner of the output string without consideration of descenders). Its position is indicated through an underscore or another type (the indication of this position may also be disabled (= default setting); cf. set_tshape). You may set or query the position by calling the operators set_tposition or get_tposition.

After you opened a textual window the position of the cursor is set to (H,0). Whereby H signifies the height of the default font less the descenders. But the cursor is not shown. Hence the output starts for writing in the upper left corner of the window.

You may query the colors of the background and the image edges by calling query_color. In the same way you may use query_color in a window of type 'invisible'. During output (write_string) you may set the clipping of text out of the window edges by calling set_check(::'~text':). This disables the creation of error messages, if text passes over the edge of the window.

The origin of the coordinate system of the window resides in the upper left corner (coordinates: (0,0)). The row index grows downward (maximal: Height-1), the column index grows to the right (maximal: Width-1).

The parameter Machine indicates the name of the computer, which has to open the window. In case of a X-window, TCP-IP only sets the name, DEC-Net sets in addition a colon behind the name. The "server" or the "screen", respectively, are not specified. If the empty string is passed the environment variable DISPLAY is used. It indicates the target computer. At this the name is indicated in common syntax

```
<Host>:0.0
```

.

For windows of type *'WIN32-Window'* and *'X-Window'* the parameter FatherWindow can be used to determine the father window for the window to be opened. In case the control 'father' is set via set_check, FatherWindow must be the ID of a HALCON window, otherwise (set_check(::'~father':)) it can also be the ID of an operating system window. If FatherWindow is passed the value 0 or 'root', then under Windows and Unix-like systems the desktop and the root window become the father window, respectively. In this case, the value of the control 'father' (set via set_check) is irrelevant. The caller must ensure that FatherWindow is a valid handle and not destroyed as long as the embedded HALCON window is used.

Position and size of a window may change during runtime of a program. This may be achieved by calling set_window_extents, but also through external interferences (window manager). In the latter case the operator set_window_extents is provided.

Opening a window causes the assignment of a called default font. It is used in connection with operators like write_string and you may overwrite it by performing set_font after calling open_textwindow. On the other hand you have the possibility to specify a default font by calling set_system(::'default_font',<Fontname>:) before opening a window (and all following windows; see also query_font).

You may set the color of the font (write_string, read_string) by calling set_color, set_rgb, set_hsi or set_gray. Calling set_insert specifies how the text or the graphics, respectively, is combined with the content of the image repeat memory. So you may achieve by calling, e.g., set_insert(::'not':) to eliminate the font after writing text twice at the same position.

Normally every output (e.g., write_string, disp_region, disp_circle, etc.) in a window is terminated by a "flush". This causes the data to be fully visible on the display after termination of the output operator. But this is not necessary in all cases, in particular if there are permanently output tasks or there is a mouse procedure active. Therefore it is more favorable (i.e., more rapid) to store the data until sufficient data is available. You may stop this behavior by calling set_system(::'flush_graphic','false':).

The content of windows is saved (in case it is supported by special driver software); i.e., it is preserved, also if the window is hidden by other windows. But this is not necessary in all cases: If you use a textual window, e.g., as a parent window for other windows, you may suppress the security mechanism for it and save the necessary memory at the same moment. You achieve this before opening the window by calling set_system(::'backing_store','false':).

Difference: graphical window - textual window

- In contrast to graphical windows (open_window) you may specify more parameters (color, edge) for a textual window while opening it.
- You may use textual windows only for input of user data (read_string).
- Using textual windows, the output of images, regions and graphics is "clipped" at the edges. Whereas during the use of graphical windows the edges are "zoomed".
- The coordinate system (e.g., with get_mbutton or get_mposition) consists of display coordinates independently of image size. The maximum coordinates are equal to the size of the window minus 1. In contrast to this, graphical windows (open_window) use always a coordinate system, which corresponds to the image format.

The parameter Mode specifies the mode of the window. It can have following values:

'visible': Normal mode for textual windows: The window is created according to the parameters and all inputs and outputs are possible.

'invisible': Invisible windows are not displayed in the display. Parameters like Row, Column, BorderWidth, BorderColor, BackgroundColor and FatherWindow do not have any meaning. Output to these windows has no effect. Input (read_string, mouse, etc.) is not possible. You may use these windows to query representation parameter for an output device without opening a (visible) window. General queries are, e.g., query_color and get_string_extents.

'transparent': These windows are transparent: the window itself is not visible (edge and background), but all the other operations are possible and all output is displayed. Parameters like BorderColor and BackgroundColor do not have any meaning. A common use for this mode is the creation of mouse sensitive regions.

'buffer': These are also not visible windows. The output of images, regions and graphics is not visible on the display, but is stored in memory. Parameters like Row, Column, BorderWidth, BorderColor, BackgroundColor and FatherWindow do not have any meaning. You may use buffer windows, if you prepare output (in the background) and copy it finally with copy_rectangle in a visible window. Another usage might be the rapid processing of image regions during interactive manipulations. Textual input and mouse interaction are not possible in this mode.

Attention

You have to keep in mind that parameters like Row, Column, Width and Height are restricted by the output device. Is a father window (FatherWindow <> 'root') specified, then the coordinates are relative to this window.

Parameters

▷ **Row** (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Row index of upper left corner.
Default: 0
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column index of upper left corner.
Default: 0
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Width** (input_control) . . . . . rectangle.extent.x ⇝ *integer*
Window's width.
Default: 256
Value range: $0 \leq$ Width (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Height** (input_control) ..... rectangle.extent.y ⇝ *integer*
Window's height.
Default: 256
Value range: 0 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 1

▷ **BorderWidth** (input_control) ..... integer ⇝ *integer*
Window border's width.
Default: 2
Value range: 0 ≤ BorderWidth (lin)
Minimum increment: 1
Recommended increment: 1

▷ **BorderColor** (input_control) ..... string ⇝ *string*
Window border's color.
Default: 'white'

▷ **BackgroundColor** (input_control) ..... string ⇝ *string*
Background color.
Default: 'black'

▷ **FatherWindow** (input_control) ..... pointer ⇝ *integer* / string
Logical number of the father window. For the display as father you may specify 'root' or 0.
Default: 0
Restriction: FatherWindow >= 0

▷ **Mode** (input_control) ..... string ⇝ *string*
Window mode.
Default: 'visible'
List of values: `Mode` ∈ {'visible', 'invisible', 'transparent', 'buffer'}

▷ **Machine** (input_control) ..... string ⇝ *string*
Computer name, where the window has to be opened or empty string.
Default: ''

▷ **WindowHandle** (output_control) ..... window ⇝ *handle*
Window handle.

Example

```
open_textwindow(0,0,900,600,1,'black','slate blue','root','visible', \
                '',WindowHandle1)
open_textwindow(10,10,300,580,3,'red','blue',WindowHandle1,'visible', \
                '',WindowHandle2)
open_window(10,320,570,580,WindowHandle1,'visible','',WindowHandle)
set_color(WindowHandle,'red')
read_image(Image,'monkey')
disp_image(Image,WindowHandle)
Button := 0
repeat
  try
      get_mposition(WindowHandle,Row,Column,Button)
      get_grayval(Image,Row,Column,Gray)
      write_string(WindowHandle2,['  Position (',Row,',',Column,')  '])
      write_string(WindowHandle2,['Gray value (',Gray,')  '])
      new_line(WindowHandle2)
  catch (Exception)
  endtry
until(Button == 4)
close_window(WindowHandle1)
```

Result

If the values of the specified parameters are correct `open_textwindow` returns `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

reset_obj_db

Possible Successors

set_color, query_window_type, get_window_type, set_window_type, get_mposition, set_tposition, set_tshape, set_window_extents, get_window_extents, query_color, set_check, set_system

Alternatives

open_window

See also

write_string, read_string, new_line, get_string_extents, get_tposition, set_color, query_window_type, get_window_type, set_window_type, get_mposition, set_tposition, set_tshape, set_window_extents, get_window_extents, query_color, set_check, set_system

Module

Foundation

query_insert (: : WindowHandle : Mode)

Query the possible graphic modes.

query_insert is obsolete and is only provided for reasons of backward compatibility.

query_insert returns the possible modes pixels can be displayed in the output window. New pixels may e.g., overwrite old ones. In most of the cases there is a functional relationship between old and new values.

Possible display functions:

'copy': overwrite displayed pixels

'xor': display old "xor" new pixels

'complement': complement displayed pixels

"copy" is always available.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Mode** (output_control) ..... string-array ⇝ *string*
Display function name.

Result

query_insert returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_insert, disp_region

See also

set_insert, get_insert

Module

Foundation

query_tshape (: : WindowHandle : TextCursor)

Query all shapes available for text cursors.

query_tshape is obsolete and is only provided for reasons of backward compatibility.

query_tshape queries the available shapes of text cursors for the output window. The retrieved shapes can be used by the operator set_tshape.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **TextCursor** (output_control) ..... string-array ⇝ *string*
Names of the available text cursors.

Result

query_tshape returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Possible Successors

set_tshape, write_string, read_string

See also

set_tshape, get_shape, set_tposition, write_string, read_string

Module

Foundation

set_comprise (: : WindowHandle, Mode :)

Define the image matrix output clipping.

set_comprise is obsolete and is only provided for reasons of backward compatibility.

set_comprise defines the image matrix output clipping. If Mode is set to 'object', only gray values belonging to the output object are displayed. If set to 'image', the whole image matrix is displayed. Default is 'object'.

Attention

If Mode was set to 'image', undefined gray values may be displayed. Depending on the context they are black or can have random content. See the examples.

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Clipping mode for gray value output.
Default: 'object'
List of values: Mode ∈ {'image', 'object'}

Example

```
read_image(Image,'fabrik')
gen_circle (Circle, 200, 200, 100.5)
reduce_domain (Image, Circle, ImageReduced)
set_system('init_new_image','false')
sobel_amp(ImageReduced,SobelReduced,'sum_abs',3)
dev_display (SobelReduced)
get_comprise(WindowHandle,Mode)
set_comprise(WindowHandle,'image')
stop ()
dev_display (SobelReduced)
```

Result

set_comprise returns 2 (H_MSG_TRUE) if Mode is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_comprise

Possible Successors

disp_image

See also

get_comprise, disp_image, disp_color

Module

Foundation

set_fix (: : WindowHandle, Mode :)

Set fixing of "look-up-table" (lut)

set_fix is obsolete and is only provided for reasons of backward compatibility.

Behavior for Mode = *'true'*: set_fix fixes that pixel lastly ascertained by one of the operators set_gray, set_color, set_hsi or set_rgb (Remark: Here a pixel is the index within the current look-up-table). To assign a new color to a fixed pixel set a color or gray value by using set_color, set_rgb, set_hsi or set_gray. This makes it possible to define any color (set_color), any gray value (set_gray) and any color combination (set_rgb, set_hsi) at any position of the look-up-table.

Mode set to 'false' reset the fixing. To modify or create a look-up-table process set_pixel, set_fix(::WindowHandle,'true':), set_rgb and set_fix(::WindowHandle,'false':) one after another.

Attention

As a side effect set_fix can change colors of "non-HALCON windows".

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Mode of fixing.
Default: 'true'
List of values: Mode ∈ {'true', 'false'}

Result

set_fix returns 2 (H_MSG_TRUE) if the window is valid, the hardware supports a look-up-table and all parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_fix

Possible Successors

set_pixel, set_rgb

See also

get_fix, set_pixel, set_rgb, set_color, set_hsi, set_gray

Module

Foundation

set_fixed_lut (: : WindowHandle, Mode :)

Fix "look-up-table" (lut) for "real color images".

set_fixed_lut is obsolete and is only provided for reasons of backward compatibility.

Fix "look-up-table" (lut) for "real color images".

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Mode of fixing.
Default: 'true'
List of values: Mode ∈ {'true', 'false'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_fixed_lut

Module

Foundation

set_insert (: : WindowHandle, Mode :)

Define the pixel output function.

set_insert is obsolete and is only provided for reasons of backward compatibility.

set_insert defines the function, with which pixels are displayed in the output window. It is e.g., possible for a pixel to overwrite the old value. In most of the cases there is a functional relationship between old and new values.

The definition value is only valid for the valid window. Output operators that honor Mode are e.g., disp_region, disp_polygon,disp_circle.

Possible display functions are:

'copy': overwrite displayed pixels

'xor': display old "xor" new pixels

'complement': complement displayed pixels

There may not be all functions available, depending on the physical display. However, "copy" is always available.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Mode** (input_control) ..... string ⇝ *string*
Name of the display function.
Default: 'copy'
List of values: Mode ∈ {'copy', 'xor', 'complement'}

Result

set_insert returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_insert, get_insert

Possible Successors

disp_region

See also

get_insert, query_insert

Module

Foundation

set_line_approx (: : WindowHandle, Approximation :)

Define the approximation error for contour display.

set_line_approx is obsolete and is only provided for reasons of backward compatibility.

set_line_approx defines the approximation error for region and xld contour display in the window WindowHandle. Approximation values greater than zero cause an approximation of line strokes using less points. This may enable faster and in some cases smoother visualization. The parameter describes the maximal deviation in pixels of the approximated contour from the original contour (Ramer-Douglas-Peucker algorithm).

Parameters

▷ **WindowHandle** (input_control) . . . . . window ⇝ *handle*
Window handle.

▷ **Approximation** (input_control) . . . . . integer ⇝ *integer*
Maximum deviation from the original contour.
Default: 0
Value range: 0 ≤ Approximation

Example

```
* Calling...
set_line_approx(WindowHandle,Approximation)
set_draw(WindowHandle,'margin')
disp_region(Obj,WindowHandle)

* ...corresponds with
get_region_polygon(Obj,Approximation,Row,Col)
disp_polygon(WindowHandle,Row,Col)
```

Result

`set_line_approx` returns `2` (`H_MSG_TRUE`) if the parameter is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_line_approx`

Possible Successors

`disp_region`

Alternatives

`get_region_polygon`, `disp_polygon`

See also

`get_line_approx`, `set_line_style`, `set_draw`, `disp_region`, `disp_polygon`

Module

Foundation

set_lut_style (: : WindowHandle, Hue, Saturation, Intensity :)

Changing the look-up-table (lut).

`set_lut_style` is obsolete and is only provided for reasons of backward compatibility.

`set_lut_style` changes the look-up-table (lut) of the device displaying the valid output window. It has got three parameters:

Hue: Rotation of color space, Hue = 1.9 conforms to a one-time rotation of the color space. No changes: Hue = 0.0 Complement colors: Hue = 0.5

Saturation: Changes of saturation, No changes: Saturation = 1.0 Gray value image: Saturation = 0.0

Intensity: Changes of intensity, No changes: Intensity = 1.0 Black image: Intensity = 0.0

Changes affect only the part of an look-up-table that is used for displaying images. The parameter of modification remain until the next call of `set_lut_style`. Calling `set_lut` has got no effect on these parameters.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Hue** (input_control) ..... real ⇝ *real*
Modification of color value.
Default: 0.0
Value range: $0.0 \leq$ Hue ≤ 1.0

▷ **Saturation** (input_control) ..... real ⇝ *real*
Modification of saturation.
Default: 1.5
Value range: $0.0 \leq$ Saturation

▷ **Intensity** (input_control) ..... real ⇝ *real*
Modification of intensity.
Default: 1.5
Value range: $0.0 \leq$ Intensity

Example

```
read_image(Image,'monkey')
dev_set_lut('color1')
repeat
  get_mbutton(WindowHandle,Row,Column,Button)
  Saturation := Row/300.0
  Hue := Column/512.0
  set_lut_style(WindowHandle,Hue,Saturation,1.0)
  dev_display(Image)
until(Button == 4)
```

Result

set_lut_style returns 2 (H_MSG_TRUE) if the window is valid and the parameter is correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_lut_style

Possible Successors

set_lut

Alternatives

set_lut, scale_image

See also

get_lut_style

Module

Foundation

set_pixel (: : WindowHandle, Pixel :)

Define a color lookup table index.

set_pixel is obsolete and is only provided for reasons of backward compatibility.

set_pixel sets pixel values: colors (set_color, set_rgb, etc.) and gray values (set_gray) are coded together into a number, called pixel. This 'pixel' is an index in the color lookup table. It ranges from 0 to 1 in b/w

images and 0 to 255 color images with 8 bit planes. It is different from the 'pixel' ("picture element") in image processing. Therefore HALCON distinguishes between pixel and image element (or gray value).

The current value can be queried with get_pixel.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **Pixel** (input_control) ..... integer(-array) ⇝ *integer*
Color lookup table index.
Default: 128
Value range: $0 \leq$ Pixel ≤ 255

Result

set_pixel returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_pixel

Possible Successors

disp_image, disp_region

Alternatives

set_rgb, set_color, set_hsi

See also

get_pixel, set_lut, disp_region, disp_image, disp_color

Module

Foundation

set_tshape (: : WindowHandle, TextCursor :)

Set the shape of the text cursor.

set_tshape is obsolete and is only provided for reasons of backward compatibility.

set_tshape sets the shape and the display mode of the text cursor of the output window.

A text cursor marks the current position for text output. It is different from the mouse cursor (although both will be called 'cursor', if the context makes misconceptions impossible). The available shapes for the text cursor can be queried with query_tshape.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **TextCursor** (input_control) ..... string ⇝ *string*
Name of cursor shape.
Default: 'invisible'

Result

set_tshape returns 2 (H_MSG_TRUE) if the window is valid and the given cursor shape is defined for this window. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window, query_tshape, get_tshape

Possible Successors

write_string, read_string

See also

get_tshape, query_tshape, write_string, read_string

Module

Foundation

slide_image (: : WindowHandleSource1, WindowHandleSource2, WindowHandle :)

Interactive output from two window buffers.

The operator slide_image does not work with HDevelop graphics windows. It is only provided for reasons of backward compatibility.

slide_image divides the window horizontal in two logical areas dependent of the mouse position. The content of the first indicated window is copied in the upper area, the content of the second window is copied in the lower area. If you press the left mouse button you may scroll the delimitation between the two areas (you may move the mouse outside the window, too. In doing so the position of the mouse relative to the window determines the borderline).

Pressing the right mouse button in the window terminates the operator slide_image.

A useful application of the operator slide_image might be the visualization of the effect of a filtering operation for an image. The output is directed to the currently set window (WindowHandle).

Attention

The three windows must have the same size and have to reside on the same computer.

Parameters

▷ **WindowHandleSource1** (input_control) ..... window ⇝ *handle*
Source window handle of the “upper window”.

▷ **WindowHandleSource2** (input_control) ..... window ⇝ *handle*
Source window handle of the “lower window”.

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Output window handle.

Example

```
read_image (Image, 'fabrik')
sobel_dir (Image, EdgeAmplitude, EdgeDirection, 'sum_abs', 3)
dev_open_window (0, 0, 512, 512, 'black', WindowHandle1)
dev_display (EdgeAmplitude)
dev_open_window (0, 0, 512, 512, 'black', WindowHandle2)
dev_display (EdgeDirection)
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
slide_image (WindowHandle1, WindowHandle2, WindowHandle)
```

Result

If the both windows exist and one of these windows is valid slide_image returns 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_window

Alternatives

copy_rectangle, get_mposition

See also

open_window, move_rectangle

Module

Foundation

write_lut (: : WindowHandle, FileName :)

Write look-up-table (lut) as file.

write_lut is obsolete and is only provided for reasons of backward compatibility.

write_lut saves the look-up-table (resp. the part of it that is relevant for displaying image gray values) of the valid output window into a file named 'FileName.lut'. It can be read again later with set_lut.

Attention

write_lut is only suitable for systems using 256 colors.

Parameters

▷ **WindowHandle** (input_control) ..... window ⇝ *handle*
Window handle.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
File name (of file containing the look-up-table).
Default: '/tmp/lut'
File extension: .lut

Result

write_lut returns 2 (H_MSG_TRUE) if the window with the required properties (256 colors) is valid and the parameter (file name) is correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_lut

See also

set_lut, set_pixel, get_pixel

Module

Foundation

17.8 Identification

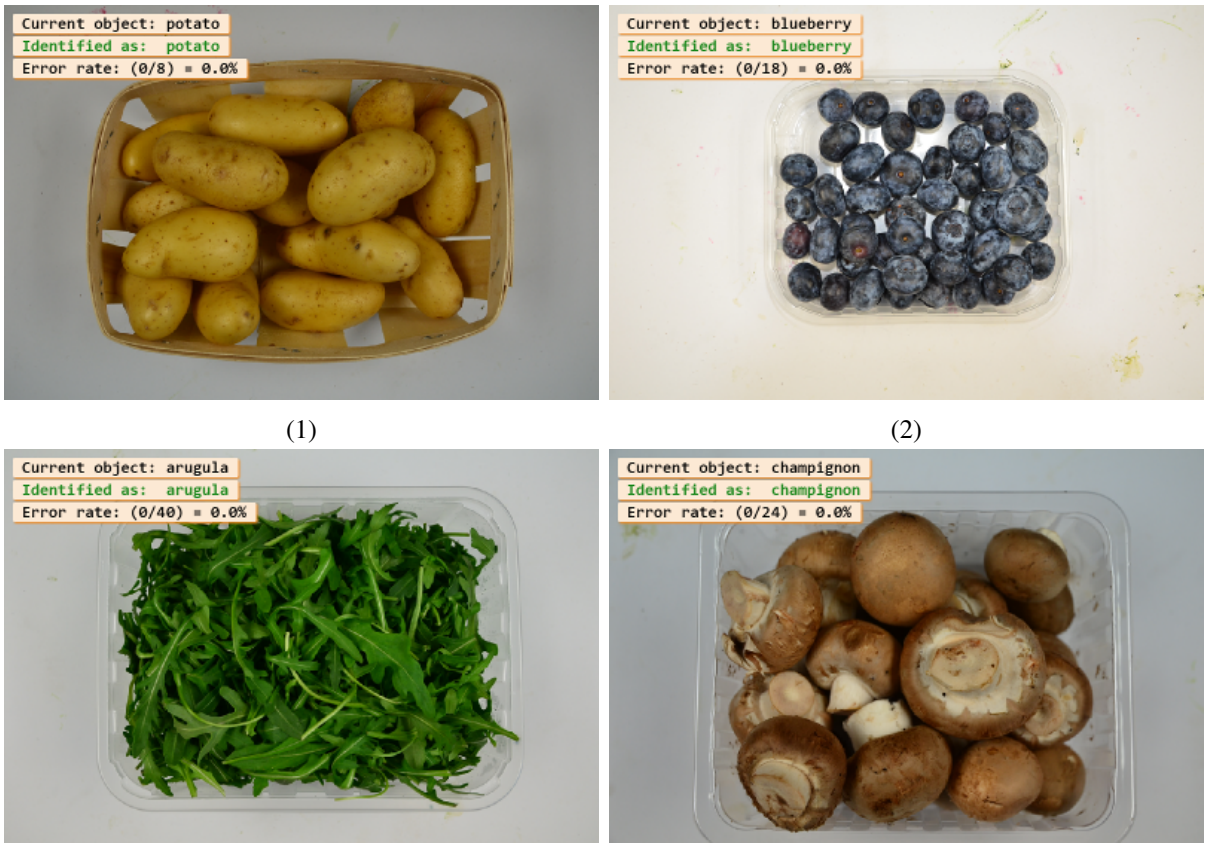
This chapter contains operators for sample-based identification.

Concept of sample-based identification

With sample-based identification, trained objects can be identified based on characteristic features like texture or color. This allows you to identify objects that do not carry bar codes or data codes. Compared to the classification

approaches described in chapter Classification, the preparation and training for sample-based identification is very convenient as no complex parameter tuning is required. Sample-based identification is capable of differentiating a large number of objects. The identification is robust against rotation, scale, and illumination changes as well as against occlusions, clutter, and moderate perspective distortions. Furthermore, the identification is robust against moderate deformations of the object and, within certain limits, it even allows to identify products stored in bulk. On the other hand, this means that objects cannot be distinguished if they appear identical except for one of the characteristics the identification is robust against. Note that sample-based identification works only with textured objects.

Sample-based identification can identify only one object at the same time. This means that each query image, or more precisely the domain of each query image, must not contain multiple objects.



Sample-based identification of the objects 'potato', 'blueberry', 'arugula', and 'champignon'. These images are from the example program `identify_vegetables.hdev`.

In the following, the steps that are required to use sample-based identification are described briefly.

Create and prepare a sample identifier: First, a sample identifier must be provided by creating and preparing a new sample identifier with

- `create_sample_identifier`,
- `add_sample_identifier_preparation_data`, and
- `prepare_sample_identifier`.

The preparation is essential to adapt the internal data structure of the sample identifier to the kind of objects to be identified. Alternatively, an already prepared sample identifier, which has been written to file with `write_sample_identifier`, can be read from file with `read_sample_identifier`.

A prepared sample identifier can be thought of as a warehouse, optimized to handle a specific group of objects.

Train sample identifier: Then, the prepared sample identifier has to be trained with samples of the individual objects to be identified. For this, the operators

- add_sample_identifier_training_data and
- train_sample_identifier

are used. Note that it is possible to retrain the sample identifier at any time. For this, samples can be removed from the sample identifier with the operator remove_sample_identifier_training_data and new samples can be added to the sample identifier with the operator add_sample_identifier_training_data. If the kind of the objects to be identified does not change too much, it is not necessary to repeat the preparation of the sample identifier. To use the picture from above, the training corresponds to filling the warehouse.

Apply sample identifier to identify objects: Finally, the trained sample identifier can be applied to identify objects with

- apply_sample_identifier.

Further operators to administrate and control sample identifiers

In addition to the operators mentioned above, the following operators can be used to administrate the sample identifier. With remove_sample_identifier_preparation_data the data that were added to the sample identifier can be removed in order to exclude them from being used for the preparation. The operators set_sample_identifier_object_info and get_sample_identifier_object_info can be used to assign and query labels, i.e, names, to the individual objects. The latter operator can further be used to determine some additional information about the number of objects available for preparation and training. Finally, the operators set_sample_identifier_param and get_sample_identifier_param can be used to set and retrieve control parameters of the sample identifier.

Glossary

In the following, the most important terms that are used in the context of sample-based identification are described:

object An object to be identified by sample-based identification.

object index The index of an object. This index can be seen as a label of the object, which is set while adding preparation data or training data to the SampleIdentifier. The object index is the most important result of the operator apply_sample_identifier. With the operator set_sample_identifier_object_info, a descriptive name can be set for each object, which eases the interpretation of the identification results.

object sample One sample or view of an object. Sometimes, also the term "sample image" is used to refer to object samples.

sample image This term is used as a synonym for "object sample", if the emphasis lies on the image.

object sample index The index of an object sample. Note that for each object, this index is set individually starting with *0*. Therefore, the object sample index is unambiguous only together with the respective object index.

preparation The adaptation of the internal data structure of the sample identifier to the features of a typical set of object samples that may appear during the identification process.

preparation object An object that has been added to the sample identifier with add_sample_identifier_preparation_data.

preparation sample An object sample of a preparation object. The preparation is typically done based on multiple preparation samples per object.

preparation data The collection of all preparation samples.

training The training of the sample identifier. In this step, the sample identifier learns to differentiate all given objects.

training object An object that has been added to the sample identifier with add_sample_identifier_training_data or which is the result of reusing preparation data as training data. In contrast to preparation objects, all training objects are labeled with a unique object index.

training sample An object sample of a training object. The training is typically done based on multiple training samples per object.

training data The collection of all training samples.

query image An image, in which an object is visible that should be identified with sample-based identification.

```
add_sample_identifier_preparation_data ( SampleImage : : SampleIdentifier, ObjectIdx, GenParamName, GenParamValue : ObjectSampleIdx )
```

Add preparation data to an existing sample identifier.

add_sample_identifier_preparation_data is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

add_sample_identifier_preparation_data adds preparation data to an existing sample identifier. This is a prerequisite for the preparation of the sample identifier with the operator prepare_sample_identifier.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

To achieve the maximum identification rate, the internal data structure of the sample identifier must be adapted to the objects to be identified. For this, it is useful to provide sample images of all kinds of objects to be identified.

With each call of add_sample_identifier_preparation_data, features of one SampleImage are added to the SampleIdentifier. The complete preparation data, which consists of all features of all sample images that have been added to the sample identifier with multiple calls of add_sample_identifier_preparation_data, should cover

- all kinds of objects to be identified and
- all different views of the objects that may appear during the identification process, sampled to at least 45 degrees.

The domain of the SampleImage should be reduced to the object, which is visible in the image. This is necessary to avoid that the SampleIdentifier is adapted to the background of the sample images.

If you cannot provide preparation data that fulfills all of the above mentioned requirements, the sample identifier may be prepared as well, but the identification rate may be slightly worse.

Note that you must provide RGB color images, if the generic parameter *'add_color_info'* has been set to *'true'* in create_sample_identifier.

Ideally, the images used for preparation and training (see add_sample_identifier_training_data and train_sample_identifier) are identical. This can easily be realized by reusing the preparation data for the training of the SampleIdentifier. To be able to reuse the preparation data, the index of the object must be given in ObjectIdx. Otherwise, the ObjectIdx can be set to *'unknown'*. See prepare_sample_identifier for an explanation on how to reuse preparation data for the training of the sample identifier.

The following generic parameters can be used to influence the behavior of the operator add_sample_identifier_preparation_data. These parameters and their corresponding values can be specified by using GenParamName and GenParamValue, respectively. The following values for GenParamName are possible:

'image_resize_method': See create_sample_identifier for a description of this parameter.

List of values: *'none'*, *'scale_factor'*, *'subsampling_step'*, *'image_area'*

Default: If the *'image_resize_method'* is not set explicitly by this operator, the value that has been set with create_sample_identifier or set_sample_identifier_param will be used.

'image_resize_value': See create_sample_identifier for a description of this parameter.

Suggested values: *0.25*, *0.5*, *1.0*, *2*, *3*, *4*

Default: If the *'image_resize_value'* is not set explicitly by this operator, the value that has been set with create_sample_identifier or set_sample_identifier_param will be used.

add_sample_identifier_preparation_data returns the object sample index of the preparation sample given in SampleImage. This index can, e.g., be used to remove this sample from the preparation data, if the sample identifier should be prepared based on a different set of preparation data.

Parameters

- ▷ **SampleImage** (input_object) . . . . . (multichannel-)image ⇝ *object* : byte
  Image that shows an object.
- ▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
  Handle of the sample identifier.
- ▷ **ObjectIdx** (input_control) . . . . . integer ⇝ *integer* / string
  Index of the object visible in the SampleImage.
  Default: 'unknown'
  Suggested values: ObjectIdx ∈ {'unknown', 0, 1, 2, 3, 4, 5}
- ▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
  Generic parameter name.
  Default: []
  List of values: GenParamName ∈ {'image_resize_method', 'image_resize_value'}
- ▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / string / integer
  Generic parameter value.
  Default: []
  Suggested values: GenParamValue ∈ {'none', 'scale_factor', 'subsampling_step', 'image_area', 0.25, 0.5, 0.75, 1.0, 2, 3, 4}
- ▷ **ObjectSampleIdx** (output_control) . . . . . integer ⇝ *integer*
  Index of the object sample.

Result

If the parameters are valid, the operator add_sample_identifier_preparation_data returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

create_sample_identifier, read_sample_identifier

Possible Successors

prepare_sample_identifier

Alternatives

read_sample_identifier

See also

add_sample_identifier_training_data, train_sample_identifier, apply_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, write_sample_identifier, serialize_sample_identifier, deserialize_sample_identifier, clear_sample_identifier, set_sample_identifier_object_info

Module

Matching

```
add_sample_identifier_training_data ( SampleImage : : SampleIdentifier, ObjectIdx, GenParamName,
    GenParamValue : ObjectSampleIdx )
```

Add training data to an existing sample identifier.

add_sample_identifier_training_data is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

add_sample_identifier_training_data adds training data to an existing sample identifier. This is a prerequisite for the training of the sample identifier with the operator train_sample_identifier.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

In order to be able to identify objects, the SampleIdentifier must be trained with a representative set of sample images of the objects to be identified. These sample images must be added to the SampleIdentifier with add_sample_identifier_training_data.

With each call of add_sample_identifier_training_data, one SampleImage is added to the SampleIdentifier. The complete training data, which consists of all the sample images that have been added to the sample identifier with multiple calls of add_sample_identifier_training_data, must cover

- all objects to be identified and
- all different views of the objects that may appear during the identification process, sampled to at least 45°.

The domain of the SampleImage should be reduced to the object, which is visible in the image. This is necessary to avoid that the SampleIdentifier is trained to the background of the sample images.

Note that in contrast to the preparation data, it is essential that the requirements mentioned above are fulfilled. Otherwise, the sample identifier will not be able to identify the objects for which relevant sample images are missing.

Also note that you must provide RGB color images, if the generic parameter *'add_color_info'* has been set to *'true'* in create_sample_identifier.

Ideally, the images used for preparation and training (see add_sample_identifier_preparation_data and prepare_sample_identifier) are identical. See prepare_sample_identifier for an explanation on how to reuse preparation data for the training of the sample identifier.

The following generic parameters can be used to influence the behavior of the operator add_sample_identifier_training_data. These parameters and their corresponding values can be specified by using GenParamName and GenParamValue, respectively. The following values for GenParamName are possible:

'image_resize_method': See create_sample_identifier for a description of this parameter.

List of values: *'none'*, *'scale_factor'*, *'subsampling_step'*, *'image_area'*

Default: If the *'image_resize_method'* is not set explicitly by this operator, the value that has been set with create_sample_identifier or set_sample_identifier_param will be used.

'image_resize_value': See create_sample_identifier for a description of this parameter.

Suggested values: *0.25*, *0.5*, *1.0*, *2*, *3*, *4*

Default: If the *'image_resize_value'* is not set explicitly by this operator, the value that has been set with create_sample_identifier or set_sample_identifier_param will be used.

add_sample_identifier_training_data returns the object sample index of the training sample given in SampleImage. This index can, e.g., be used to remove this sample from the training data, if the sample identifier should be retrained based on a different set of training data.

Parameters

▷ **SampleImage** (input_object) ..... (multichannel-)image ⇝ *object : byte*
Image that shows an object.

▷ **SampleIdentifier** (input_control) ..... sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **ObjectIdx** (input_control) . . . . . integer ⇝ *integer* / string
Index of the object visible in the SampleImage.
Suggested values: ObjectIdx ∈ {0, 1, 2, 3, 4, 5}

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Generic parameter name.
Default: []
List of values: GenParamName ∈ {'image_resize_method', 'image_resize_value'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / string / integer
Generic parameter value.
Default: []
Suggested values: GenParamValue ∈ {'none', 'scale_factor', 'subsampling_step', 'image_area', 0.25, 0.5, 0.75, 1.0, 2, 3, 4}

▷ **ObjectSampleIdx** (output_control) . . . . . integer ⇝ *integer*
Index of the object sample.

Result

If the parameters are valid, the operator add_sample_identifier_training_data returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

prepare_sample_identifier, read_sample_identifier

Possible Successors

train_sample_identifier

Alternatives

read_sample_identifier

See also

create_sample_identifier, add_sample_identifier_preparation_data, apply_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, write_sample_identifier, serialize_sample_identifier, deserialize_sample_identifier, clear_sample_identifier, set_sample_identifier_object_info

Module

Matching

apply_sample_identifier (Image : : SampleIdentifier, NumResults, RatingThreshold, GenParamName, GenParamValue : ObjectIdx, Rating)

Identify objects with a sample identifier.

apply_sample_identifier is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

apply_sample_identifier identifies the object in the Image using the given SampleIdentifier and returns the corresponding ObjectIdx.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

The operator apply_sample_identifier can identify one object per query Image. This means that the Image, or more precisely the domain of the Image, must not contain multiple objects. The parameter

NumResults defines the number of hypotheses that are returned, ordered by their rating value. If the generic parameter *'apply_rating_threshold'* is set to *'true'* (see below), all returned hypotheses have a rating value better than the specified RatingThreshold.

The index of the identified object is returned in ObjectIdx and its rating value is returned in Rating. If NumResults is set to a value larger than *1*, the indices and ratings of the best rated NumResults hypotheses are returned in ObjectIdx and Rating.

The following generic parameters can be used to influence the behavior of the SampleIdentifier. These parameters and their corresponding values can be specified by using GenParamName and GenParamValue, respectively. The following values for GenParamName are possible:

'rating_method': This parameter determines, which rating method is used for the identification of the objects. There are three different methods available:

'distance' The rating is based on the distance between the features of the image of the object to be identified and the trained sample images. The rating values lie between *0.0* and *2.0*. If the images are identical, a rating value of *0.0* will be returned in Rating. The more different the images are, the higher the rating value will be. With this rating method, the raw internal rating is returned. This rating has the advantage that the rating value is independent of training samples other than that of the identified object.

'score' The rating is based on an elaborate combination of the scores for texture and color. The rating values lie between *0.0* and *1.0*. Higher rating values indicate more similar images. If both texture and color are used, this rating method yields the best identification results and should therefore be used. One drawback of this rating method is that the rating value of a training image is typically far below *1.0*. Furthermore, a suitable threshold must be determined for each application individually.

'score_single' The rating is based on a simple combination of the scores for texture and color. The rating values lie between *0.0* and *1.0*. Higher rating values indicate more similar images. If both texture and color are used, the rating method *'score'* yields better results. The advantage of the rating method *'score_single'* is, that it yields a rating value of *1.0* for training images. Therefore, this rating method should be used if only texture or only color is used for the identification. For this rating method, it is much simpler to select a suitable threshold than for the rating method *'score'*.

Note that the selection of the rating method influences both the identification of the objects, i.e., their discrimination, as well as the rating value that is returned in the parameter Rating.

List of values: *'distance'*, *'score'*, *'score_single'*

Default: *'score'*

'apply_rating_threshold': This parameter determines if the RatingThreshold will be applied or not. If *'apply_rating_threshold'* is set to *'true'*, the rating threshold given in RatingThreshold will be applied.

List of values: *'true'*, *'false'*

Default: *'true'*

'use_color_info': See set_sample_identifier_param for a description of this parameter.

List of values: *'true'*, *'false'*

Default: As long as *'use_color_info'* has not been set, the value set with *'add_color_info'* in create_sample_identifier is used.

'use_texture_info': See set_sample_identifier_param for a description of this parameter.

List of values: *'true'*, *'false'*

Default: As long as *'use_texture_info'* has not been set, the value set with *'add_texture_info'* in create_sample_identifier is used.

'image_resize_method': See create_sample_identifier for a description of this parameter.

List of values: *'none'*, *'scale_factor'*, *'subsampling_step'*, *'image_area'*

Default: If the *'image_resize_method'* is not set explicitly by this operator, the value that has been set with create_sample_identifier or set_sample_identifier_param will be used.

'image_resize_value': See create_sample_identifier for a description of this parameter. Note the objects to be identified should appear approximately in the same scale in the sample images as well as in the query images.

Suggested values: *0.25*, *0.5*, *1.0*, *2*, *3*, *4*

Default: If the *'image_resize_value'* is not set explicitly by this operator, the value that has been set with create_sample_identifier or set_sample_identifier_param will be used.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object : byte*
  Image showing the object to be identified.
- ▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
  Handle of the sample identifier.
- ▷ **NumResults** (input_control) . . . . . integer ⇝ *integer*
  Number of suggested object indices.
  Default: 1
  Suggested values: NumResults ∈ {1, 2, 3, 4, 5, 10}
- ▷ **RatingThreshold** (input_control) . . . . . real ⇝ *real*
  Rating threshold.
  Default: 0.0
  Suggested values: RatingThreshold ∈ {0.05, 0.1, 0.15, 0.2}
- ▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
  Generic parameter name.
  Default: []
  List of values: GenParamName ∈ {'use_texture_info', 'use_color_info', 'image_resize_method', 'image_resize_value', 'rating_method', 'apply_rating_threshold'}
- ▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / string / integer
  Generic parameter value.
  Default: []
  List of values: GenParamValue ∈ {'true', 'false', 'distance', 'score', 'score_single'}
- ▷ **ObjectIdx** (output_control) . . . . . integer(-array) ⇝ *integer*
  Index of the identified object.
- ▷ **Rating** (output_control) . . . . . real(-array) ⇝ *real*
  Rating value of the identified object.

Result

If the parameters are valid, the operator set_sample_identifier_param returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

train_sample_identifier, read_sample_identifier

Possible Successors

add_sample_identifier_training_data

See also

create_sample_identifier, add_sample_identifier_preparation_data, prepare_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, write_sample_identifier, serialize_sample_identifier, deserialize_sample_identifier, clear_sample_identifier, set_sample_identifier_object_info

Module

Matching

clear_sample_identifier (: : SampleIdentifier :)

Free the memory of a sample identifier.

clear_sample_identifier is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

The operator clear_sample_identifier frees the memory of a sample identifier that was created by create_sample_identifier, read_sample_identifier, or deserialize_sample_identifier. After calling clear_sample_identifier, the sample identifier can no longer be used. The handle SampleIdentifier becomes invalid.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

Parameters

▷ **SampleIdentifier** (input_control) ..... sample_identifier ⇝ *handle*
Handle of the sample identifier.

Result

If the handle of the sample identifier is valid, the operator clear_sample_identifier returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SampleIdentifier

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

write_sample_identifier, serialize_sample_identifier

See also

create_sample_identifier, add_sample_identifier_preparation_data, prepare_sample_identifier, add_sample_identifier_training_data, train_sample_identifier, apply_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, read_sample_identifier, deserialize_sample_identifier, set_sample_identifier_object_info

Module

Matching

create_sample_identifier (: : GenParamName, GenParamValue : SampleIdentifier)

Create a new sample identifier.

create_sample_identifier is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

The operator create_sample_identifier creates a new sample identifier. Alternatively, an already available sample identifier can be read from file with the operator read_sample_identifier.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

Note that if you want to use color information, you must specify this explicitly by setting the generic parameter *'add_color_info'* to *'true'*.

The generic parameters can be used to influence the behavior of the sample identifier. Typically, only *'add_color_info'* must be considered. The parameters and their corresponding values can be specified by using GenParamName and GenParamValue, respectively. The following values for GenParamName are possible:

'add_color_info': This parameter determines if color information is used for the identification. If *'add_color_info'* is set to *'true'*, color information is used. Note that in this case, all images used for the preparation and the training of the sample identifier must be RGB color images. If *'add_color_info'* is set to *'false'*, no color information is used. Use color information, if the objects to be identified have different colors and if the illumination can be controlled to some degree. In this case, the use of color information makes the identification more robust.

List of values: *'true'*, *'false'*

Default: *'false'*

'add_texture_info': This parameter determines if texture information is used for the identification. If *'add_texture_info'* is set to *'true'*, texture information is used. If *'add_texture_info'* is set to *'false'*, no texture information is used. Typically, *'add_texture_info'* should be set to *'true'*, because sample-based identification requires textured objects. Note that at least one of the two generic parameters *'add_color_info'* and *'add_texture_info'* must be set to *'true'*.

List of values: *'true'*, *'false'*

Default: *'true'*

'image_resize_method': To speed up the identification process, the images are resized internally to a given size or by a given factor. If *'none'* is selected, no resizing will be done. For *'scale_factor'*, a constant scale factor can be set and for *'subsampling_step'*, a constant subsampling step, i.e., the inverse of the scale factor can be set. Finally, for *'image_area'*, a constant size of the resized image can be specified. The corresponding values can be set with the *'image_resize_value'* (see below).

To make the identification process faster, choose parameters that lead to a smaller internal image, i.e., use smaller scale factors or image sizes or a larger subsampling step. If the objects to be identified show high-frequency texture, the identification result may become better, if a larger internal image size is used, because otherwise the texture may be eliminated in the resized images.

List of values: *'none'*, *'scale_factor'*, *'subsampling_step'*, *'image_area'*

Default: *'image_area'*

'image_resize_value': With this parameter, the selected *'image_resize_method'* can be parameterized. If *'image_resize_method'* is set to *'scale_factor'*, the value of *'image_resize_value'* defines the scale factor to be used. If *'image_resize_method'* is set to *'subsampling_step'*, the value of *'image_resize_value'* defines the subsampling step, i.e., the inverse of the scale factor. If *'image_resize_method'* is set to *'image_area'*, the value of *'image_resize_value'* defines the area of the resized image given in megapixels, i.e., in million pixels.

Suggested values: *0.25*, *0.5*, *1.0*, *2*, *3*, *4*

Default: The default value depends on the selected *'image_resize_method'*. It is

- *0.5* for *'scale_factor'*,
- *2.0* for *'subsampling_step'*, and
- *0.5* for *'image_area'*.

Parameters

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Parameter name.
Default: []
List of values: GenParamName ∈ {'image_resize_method', 'image_resize_value', 'add_texture_info', 'add_color_info'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / real / integer
Parameter value.
Default: []
Suggested values: GenParamValue ∈ {'none', 'scale_factor', 'subsampling_step', 'image_area', 0.25, 0.5, 0.75, 1.0, 2, 3, 4, 'true', 'false'}

▷ **SampleIdentifier** (output_control) . . . . . sample_identifier ⇝ *handle*
Handle of the sample identifier.

Result

If the parameters are valid, the operator `create_sample_identifier` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`add_sample_identifier_preparation_data`, `set_sample_identifier_param`

Alternatives

`read_sample_identifier`

See also

`prepare_sample_identifier`, `add_sample_identifier_training_data`, `train_sample_identifier`, `apply_sample_identifier`, `get_sample_identifier_param`, `get_sample_identifier_object_info`, `remove_sample_identifier_preparation_data`, `remove_sample_identifier_training_data`, `write_sample_identifier`, `serialize_sample_identifier`, `deserialize_sample_identifier`, `clear_sample_identifier`, `set_sample_identifier_object_info`

Module

Matching

deserialize_sample_identifier (: : SerializedItemHandle : SampleIdentifier)

Deserialize a serialized sample identifier.

`deserialize_sample_identifier` is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

`deserialize_sample_identifier` deserializes a sample identifier that was serialized by `serialize_sample_identifier` (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The serialized sample identifier is defined by the handle `SerializedItemHandle`. The deserialized values are stored in an automatically created sample identifier with the handle `SampleIdentifier`.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

Parameters

▷ **`SerializedItemHandle`** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **`SampleIdentifier`** (output_control) ..... sample_identifier ⇝ *handle*
Handle of the sample identifier.

Result

If the parameters are valid, the operator `deserialize_sample_identifier` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Successors

apply_sample_identifier, add_sample_identifier_training_data

Alternatives

read_sample_identifier

See also

create_sample_identifier, add_sample_identifier_preparation_data, prepare_sample_identifier, train_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, write_sample_identifier, serialize_sample_identifier, clear_sample_identifier, set_sample_identifier_object_info

Module

Matching

get_sample_identifier_object_info (: : SampleIdentifier, ObjectIdx, InfoName : InfoValue)

Retrieve information about an object of a sample identifier.

get_sample_identifier_object_info is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

get_sample_identifier_object_info retrieves information from the SampleIdentifier. This information comprises the number of objects and object samples as well as their indices. Furthermore, the information set with set_sample_identifier_object_info can be retrieved.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

The parameter ObjectIdx defines the index of the object for which information is retrieved. Note that this parameter is not evaluated if InfoName is set to *'num_preparation_objects'*, *'preparation_object_idx'*, *'num_training_objects'*, or *'training_object_idx'*.

InfoName defines the kind of information to be returned in InfoValue. The following values for InfoName are possible:

'num_preparation_objects': The number of preparation objects. Note that all preparation objects for which the object index has been set to *'unknown'*, are counted as one single preparation object.

'preparation_object_idx': The list of indices of all available preparation objects. Note that this list will contain the string *'unknown'*, if the object index of at least one preparation object hast been set to *'unknown'*.

'num_preparation_samples': The number of preparation samples for the preparation object indicated by ObjectIdx.

'preparation_sample_idx': The list of indices of all available preparation samples for the preparation object indicated by ObjectIdx.

'num_training_objects': The number of training objects.

'training_object_idx': The list of indices of all available training objects.

'num_training_samples': The number of training samples for the training object indicated by ObjectIdx.

'training_sample_idx': The list of indices of all available training samples for the training object indicated by ObjectIdx.

'preparation_object_name': The information of the preparation object that has been set with set_sample_identifier_object_info for the preparation object indicated by ObjectIdx.

'training_object_name': The information of the training object that has been set with set_sample_identifier_object_info for the training object indicated by ObjectIdx.

Parameters

▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **ObjectIdx** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Index of the object for which information is retrieved.
Suggested values: ObjectIdx ∈ {0, 1, 2, 3, 4, 5, 'unknown'}

▷ **InfoName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Define, for which kind of object information is retrieved.
Default: 'num_training_objects'
List of values: InfoName ∈ {'num_preparation_objects', 'preparation_object_idx', 'num_preparation_samples', 'preparation_sample_idx', 'num_training_objects', 'training_object_idx', 'num_training_samples', 'training_sample_idx', 'preparation_object_name', 'training_object_name'}

▷ **InfoValue** (output_control) . . . . . attribute.value(-array) ⇝ *integer* / string
Information about the object.

Result

If the parameters are valid, the operator get_sample_identifier_object_info returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_identifier_preparation_data, add_sample_identifier_training_data

See also

create_sample_identifier, prepare_sample_identifier, train_sample_identifier, apply_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, write_sample_identifier, read_sample_identifier, serialize_sample_identifier, deserialize_sample_identifier, clear_sample_identifier

Module

Matching

get_sample_identifier_param (: : SampleIdentifier, GenParamName : GenParamValue)

Get selected parameters of a sample identifier.

get_sample_identifier_param is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

The operator get_sample_identifier_param allows to query the values of the different parameters of the given SampleIdentifier.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

The values of the following parameters can be queried:

'add_color_info': Do the operators add_sample_identifier_preparation_data and add_sample_identifier_training_data add color information to the SampleIdentifier?

'add_texture_info': Do the operators add_sample_identifier_preparation_data and add_sample_identifier_training_data add texture information to the SampleIdentifier?

'use_color_info': Is the SampleIdentifier configured to use color information for the identification with apply_sample_identifier?

'use_texture_info': Is the SampleIdentifier configured to use texture information for the identification apply_sample_identifier?

'image_resize_method': The selected image resize method.

'image_resize_value': The current image resize value.

'rating_method': The selected rating method which is used by apply_sample_identifier.

Parameters

▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Parameter name.
Default: 'rating_method'
List of values: GenParamName ∈ {'add_texture_info', 'add_color_info', 'use_texture_info', 'use_color_info', 'image_resize_method', 'image_resize_value', 'rating_method'}

▷ **GenParamValue** (output_control) . . . . . attribute.value ⇝ *real* / string / integer
Parameter value.

Result

If the parameters are valid, the operator get_sample_identifier_param returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_sample_identifier, set_sample_identifier_param

See also

add_sample_identifier_preparation_data, prepare_sample_identifier, add_sample_identifier_training_data, train_sample_identifier, apply_sample_identifier, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, write_sample_identifier, read_sample_identifier, serialize_sample_identifier, deserialize_sample_identifier, clear_sample_identifier, set_sample_identifier_object_info

Module

Matching

prepare_sample_identifier (: : SampleIdentifier, RemovePreparationData, GenParamName, GenParamValue :)

Adapt the internal data structure of a sample identifier to the objects to be identified.

prepare_sample_identifier is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

prepare_sample_identifier prepares the given SampleIdentifier using the preparation data, which have been set with add_sample_identifier_preparation_data.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

In order to achieve the maximum identification rate, the internal data structure of the SampleIdentifier must be adapted to the kind of objects to be identified.

To prepare the SampleIdentifier, first, all the relevant sample images of the objects to be identified must be added with add_sample_identifier_preparation_data. Then, with the call of prepare_sample_identifier, the internal data structure of the SampleIdentifier is adapted to the features of the provided sample images.

This process needs a lot of resources. Both, memory consumption and runtime are fairly high. E.g., if 100 sample images are used to prepare the sample identifier, approximately 300 MB of memory are required and the runtime will be in the range of a few minutes. If 1000 sample images are used, expect a memory consumption of about 2 GB and a runtime in the range of an hour.

Especially the preparation data requires a lot of memory. Typically, this data is no longer needed after the preparation of the sample identifier. Therefore, it can be removed by setting the parameter RemovePreparationData to *'true'*.

As an alternative to the preparation, e.g., if you are not able to spend this much memory and/or time, and if you are fine with a slightly higher misidentification rate, you can just read a sample identifier that was previously prepared with samples of similar object kinds from file (see read_sample_identifier).

To reuse the preparation data for the training, you can derive training data from the preparation data. This saves the effort to add training data separately with add_sample_identifier_training_data (see train_sample_identifier for a description of the training of the sample identifier). Note that the training data require far less memory than the preparation data. If the generic parameter *'copy_preparation_data_to_training_data'* is set to *'true'* (which is the default), for all sample images that have been added with add_sample_identifier_preparation_data and for which the object index has been set, training data is automatically derived. The object index that has been set with add_sample_identifier_preparation_data is taken unchanged. But note that the object sample index may be changed, because it is renumbered consecutively, starting with zero.

Parameters

- ▷ **SampleIdentifier** (input_control) ..... sample_identifier ⇝ *handle*
  Handle of the sample identifier.
- ▷ **RemovePreparationData** (input_control) ..... string ⇝ *string*
  Indicates if the preparation data should be removed.
  Default: 'true'
  List of values: RemovePreparationData ∈ {'true', 'false'}
- ▷ **GenParamName** (input_control) ..... attribute.name-array ⇝ *string*
  Generic parameter name.
  Default: []
  List of values: GenParamName ∈ {'copy_preparation_data_to_training_data'}
- ▷ **GenParamValue** (input_control) ..... attribute.value-array ⇝ *string* / integer / real
  Generic parameter value.
  Default: []
  List of values: GenParamValue ∈ {'true', 'false'}

Result

If the parameters are valid, the operator prepare_sample_identifier returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_identifier_preparation_data

Possible Successors

write_sample_identifier, add_sample_identifier_training_data, train_sample_identifier

Alternatives

read_sample_identifier

See also

create_sample_identifier, apply_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, write_sample_identifier, serialize_sample_identifier, deserialize_sample_identifier, clear_sample_identifier, set_sample_identifier_object_info

Module

Matching

read_sample_identifier (: : FileName : SampleIdentifier)

Read a sample identifier from a file.

read_sample_identifier is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

The operator read_sample_identifier reads a sample identifier, which has been written with write_sample_identifier, from the file FileName. The default HALCON file extension for the sample identifier is 'sid'.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
File name.
File extension: .sid

▷ **SampleIdentifier** (output_control) ..... sample_identifier ⇝ *handle*
Handle of the sample identifier.

Result

If the file name is valid, the operator read_sample_identifier returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

apply_sample_identifier, add_sample_identifier_training_data

Alternatives

deserialize_sample_identifier

See also

create_sample_identifier, add_sample_identifier_preparation_data, prepare_sample_identifier, train_sample_identifier, apply_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, write_sample_identifier, serialize_sample_identifier, clear_sample_identifier, set_sample_identifier_object_info

Module

Matching

remove_sample_identifier_preparation_data (: : SampleIdentifier, ObjectIdx, ObjectSampleIdx :)

Remove preparation data from a sample identifier.

remove_sample_identifier_preparation_data is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

remove_sample_identifier_preparation_data removes preparation data from a sample identifier.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

ObjectIdx defines the index of the preparation object, of which samples should be removed. To remove preparation samples of preparation objects for which the object index has been set to *'unknown'* with add_sample_identifier_preparation_data, set ObjectIdx to *'unknown'*. The indices of all currently available preparation objects can be queried using get_sample_identifier_object_info.

ObjectSampleIdx defines the index of the preparation sample that should be removed. Removing the last preparation sample of a preparation object will remove also the corresponding preparation object itself from the SampleIdentifier. To remove all preparation samples of the preparation object defined by ObjectIdx, ObjectSampleIdx can be set to *'all'*.

To remove all preparation samples of **all** preparation objects from the SampleIdentifier, set both ObjectIdx and ObjectSampleIdx to *'all'*.

Parameters

▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **ObjectIdx** (input_control) . . . . . integer ⇝ *integer* / string
Index of the preparation object, of which samples should be removed.
Suggested values: ObjectIdx ∈ {0, 1, 2, 3, 4, 5, 'all', 'unknown'}

▷ **ObjectSampleIdx** (input_control) . . . . . integer ⇝ *integer* / string
Index of the preparation sample that should be removed.
Suggested values: ObjectSampleIdx ∈ {0, 1, 2, 3, 4, 5, 'all'}

Result

If the parameters are valid, the operator remove_sample_identifier_preparation_data returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_identifier_preparation_data

Possible Successors

prepare_sample_identifier

Alternatives

create_sample_identifier

See also

add_sample_identifier_training_data, train_sample_identifier, apply_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_training_data, write_sample_identifier,

read_sample_identifier, serialize_sample_identifier,
deserialize_sample_identifier, clear_sample_identifier,
set_sample_identifier_object_info

Module

Matching

remove_sample_identifier_training_data (: : SampleIdentifier,
ObjectIdx, ObjectSampleIdx :)

Remove training data from a sample identifier.

remove_sample_identifier_training_data is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

remove_sample_identifier_training_data removes training data from a sample identifier.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

ObjectIdx defines the index of the training object, of which samples should be removed.

ObjectSampleIdx defines the index of the training sample that should be removed. To remove all training samples of the training object defined by ObjectIdx, ObjectSampleIdx can be set to *'all'*. This will remove also the corresponding training object itself from the SampleIdentifier.

To remove all training samples of all training objects from the SampleIdentifier, set both ObjectIdx and ObjectSampleIdx to *'all'*.

Parameters

▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **ObjectIdx** (input_control) . . . . . integer ⇝ *integer* / string
Index of the training object, from which samples should be removed.
Suggested values: ObjectIdx ∈ {0, 1, 2, 3, 4, 5, 'all'}

▷ **ObjectSampleIdx** (input_control) . . . . . integer ⇝ *integer* / string
Index of the training sample that should be removed.
Suggested values: ObjectSampleIdx ∈ {0, 1, 2, 3, 4, 5, 'all'}

Result

If the parameters are valid, the operator remove_sample_identifier_training_data returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_identifier_training_data

Possible Successors

train_sample_identifier

Alternatives

read_sample_identifier

See also

create_sample_identifier, add_sample_identifier_preparation_data,
prepare_sample_identifier, apply_sample_identifier,
set_sample_identifier_param, get_sample_identifier_param,
get_sample_identifier_object_info, remove_sample_identifier_preparation_data,

write_sample_identifier, serialize_sample_identifier, deserialize_sample_identifier, clear_sample_identifier, set_sample_identifier_object_info

Module

Matching

serialize_sample_identifier (: : SampleIdentifier : SerializedItemHandle)

Serialize a sample identifier.

serialize_sample_identifier is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

serialize_sample_identifier serializes the data of a sample identifier (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_sample_identifier is converted to a serialized item. The sample identifier is defined by the handle SampleIdentifier. The serialized sample identifier is returned by the handle SerializedItemHandle and can be deserialized by deserialize_sample_identifier.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

Parameters

▷ **SampleIdentifier** (input_control) ..... sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_sample_identifier returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

prepare_sample_identifier, train_sample_identifier, add_sample_identifier_preparation_data, add_sample_identifier_training_data, set_sample_identifier_param

Possible Successors

clear_sample_identifier

Alternatives

write_sample_identifier

Module

Matching

set_sample_identifier_object_info (: : SampleIdentifier, ObjectIdx, InfoName, InfoValue :)

Define a name or a description for an object of a sample identifier.

set_sample_identifier_object_info is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

set_sample_identifier_object_info sets information like a name or a description for an object of the SampleIdentifier. This information can then be retrieved with get_sample_identifier_object_info.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

To set information for preparation objects, set InfoName to *'preparation_object_name'*. To set information for training objects, set InfoName to *'training_object_name'*. The information itself is set with the parameter InfoValue.

Parameters

▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **ObjectIdx** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Index of the object for which information is set.
Suggested values: ObjectIdx ∈ {0, 1, 2, 3, 4, 5}

▷ **InfoName** (input_control) . . . . . attribute.name ⇝ *string*
Define, for which kind of object information is set.
Default: 'training_object_name'
List of values: InfoName ∈ {'preparation_object_name', 'training_object_name'}

▷ **InfoValue** (input_control) . . . . . attribute.value(-array) ⇝ *string*
Information about the object.

Result

If the parameters are valid, the operator set_sample_identifier_object_info returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_identifier_preparation_data, add_sample_identifier_training_data

See also

create_sample_identifier, prepare_sample_identifier, train_sample_identifier, apply_sample_identifier, get_sample_identifier_object_info, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, write_sample_identifier, read_sample_identifier, serialize_sample_identifier, deserialize_sample_identifier, clear_sample_identifier

Module

Matching

set_sample_identifier_param (: : SampleIdentifier, GenParamName, GenParamValue :)

Set selected parameters of a sample identifier.

set_sample_identifier_param is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

The operator set_sample_identifier_param is used to set or change the different parameters of the given SampleIdentifier.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

The following parameters can be used to influence the behavior of the SampleIdentifier. These parameters and their corresponding values can be specified by using GenParamName and GenParamValue, respectively. The following values for GenParamName are possible:

'use_color_info': This parameter determines if color information is used for the identification. If *'use_color_info'* is set to *'true'*, color information is used. Note that in this case, the images used for apply_sample_identifier must be RGB color images. If *'use_color_info'* is set to *'false'*, no color information is used. Use color information if the objects to be identified have different colors and the illumination can be controlled to some degree. In this case, the use of color information makes the identification more robust.

Note that in contrast to the generic parameter *'add_color_info'*, which can only be set in create_sample_identifier, this parameter affects only the operator apply_sample_identifier. With this, it is, e.g., possible to train the sample identifier using texture and color and to identify selected objects only based on the texture information.

List of values: *'true'*, *'false'*

Default: As long as *'use_color_info'* has not been set, the value set with *'add_color_info'* in create_sample_identifier is used.

'use_texture_info': This parameter determines if texture information is used for the identification. If *'use_texture_info'* is set to *'true'*, texture information is used. If *'use_texture_info'* is set to *'false'*, no texture information is used. Typically, *'use_texture_info'* should be set to *'true'*, because sample-based identification requires textured objects. Note that at least one of the two generic parameters *'use_color_info'* and *'use_texture_info'* must be set to *'true'*.

Note that in contrast to the generic parameter *'add_texture_info'*, which can only be set in create_sample_identifier, this parameter affects only the operator apply_sample_identifier.

List of values: *'true'*, *'false'*

Default: As long as *'use_texture_info'* has not been set, the value set with *'add_texture_info'* in create_sample_identifier is used.

'image_resize_method': See create_sample_identifier for a description of this parameter.

List of values: *'none'*, *'scale_factor'*, *'subsampling_step'*, *'image_area'*

Default: If the *'image_resize_method'* is not set explicitly by this operator, the value that has been set with create_sample_identifier will be used.

'image_resize_value': See create_sample_identifier for a description of this parameter.

Suggested values: *0.25*, *0.5*, *1.0*, *2*, *3*, *4*

Default: If the *'image_resize_value'* is not set explicitly by this operator, the value that has been set with create_sample_identifier will be used.

'rating_method': See apply_sample_identifier for a description of this parameter that is used only by apply_sample_identifier.

List of values: *'distance'*, *'score'*, *'score_single'*

Default: *'score'*

Parameters

▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Parameter name.
Default: 'rating_method'
List of values: GenParamName ∈ {'use_texture_info', 'use_color_info', 'image_resize_method', 'image_resize_value', 'rating_method'}

▷ **GenParamValue** (input_control) . . . . . attribute.value ⇝ *real* / integer / string
Parameter value.
Default: 'score_single'
List of values: GenParamValue ∈ {'true', 'false', 'none', 'scale_factor', 'subsampling_step', 'distance', 'score', 'score_single', 'image_area', 0.25, 0.5, 0.75, 1.0, 2, 3, 4}

Result

If the parameters are valid, the operator `set_sample_identifier_param` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_sample_identifier`, `read_sample_identifier`

Alternatives

`create_sample_identifier`

See also

`add_sample_identifier_preparation_data`, `prepare_sample_identifier`, `add_sample_identifier_training_data`, `train_sample_identifier`, `apply_sample_identifier`, `get_sample_identifier_param`, `get_sample_identifier_object_info`, `remove_sample_identifier_preparation_data`, `remove_sample_identifier_training_data`, `write_sample_identifier`, `serialize_sample_identifier`, `deserialize_sample_identifier`, `set_sample_identifier_object_info`

Module

Matching

`train_sample_identifier` `( : : SampleIdentifier, GenParamName, GenParamValue : )`

Train a sample identifier.

`train_sample_identifier` is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

`train_sample_identifier` trains the given `SampleIdentifier` using the training data, which has been set with `add_sample_identifier_training_data`.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

In order to be able to identify objects, the `SampleIdentifier` must be trained with a representative set of sample images of the objects to be identified (see `add_sample_identifier_training_data`).

To train the `SampleIdentifier`, first, all the relevant sample images of the objects to be identified must be added with `add_sample_identifier_training_data`. Alternatively, the preparation data can be reused. See `prepare_sample_identifier` for an explanation on how to reuse preparation data for the training of the sample identifier. Then, with the call of `train_sample_identifier`, the `SampleIdentifier` is trained.

Note that the training itself is very fast and does not consume a lot of memory. Therefore, it is very easy to retrain the `SampleIdentifier`, e.g., after new objects were added or already trained objects were removed from the `SampleIdentifier`.

To retrain an existing sample identifier with new objects, just add the relevant sample images with `add_sample_identifier_training_data` and call `train_sample_identifier`, again. To remove objects, use the operator `remove_sample_identifier_training_data` and retrain the sample identifier with `train_sample_identifier`.

Note that the training of the sample identifier is only possible, if the sample identifier contains training data for at least two different training objects.

The generic parameters `GenParamName` and `GenParamValue` are intended for future use, only.

Parameters

▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Parameter name.
Default: []
List of values: GenParamName ∈ {[]}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / string / integer
Parameter value.
Default: []
List of values: GenParamValue ∈ {[]}

Result

If the parameters are valid, the operator train_sample_identifier returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

add_sample_identifier_training_data

Possible Successors

apply_sample_identifier, write_sample_identifier

Alternatives

read_sample_identifier

See also

create_sample_identifier, add_sample_identifier_preparation_data, prepare_sample_identifier, apply_sample_identifier, set_sample_identifier_param, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, serialize_sample_identifier, deserialize_sample_identifier, clear_sample_identifier, set_sample_identifier_object_info

Module

Matching

write_sample_identifier (: : SampleIdentifier, FileName :)

Write a sample identifier to a file.

write_sample_identifier is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators for Deep-Learning-based classification instead, for details see Deep Learning / Classification.

The operator write_sample_identifier writes a sample identifier to the file FileName. The sample identifier can be read again with read_sample_identifier. The default HALCON file extension for the sample identifier is 'sid'.

For an explanation of the concept of sample-based identification see the introduction of chapter Legacy / Identification.

Parameters

▷ **SampleIdentifier** (input_control) . . . . . sample_identifier ⇝ *handle*
Handle of the sample identifier.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: .sid

Result

If the file name is valid (write permission), the operator write_sample_identifier returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

prepare_sample_identifier, train_sample_identifier, add_sample_identifier_preparation_data, add_sample_identifier_training_data, set_sample_identifier_param

Possible Successors

clear_sample_identifier

Alternatives

serialize_sample_identifier

See also

create_sample_identifier, prepare_sample_identifier, train_sample_identifier, apply_sample_identifier, get_sample_identifier_param, get_sample_identifier_object_info, remove_sample_identifier_preparation_data, remove_sample_identifier_training_data, read_sample_identifier, deserialize_sample_identifier, set_sample_identifier_object_info

Module

Matching

17.9 Matching

adapt_template (Image : : TemplateID :)

Adapting a template to the size of an image.

adapt_template is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator adapt_template serves to adapt a template which has been created by create_template to the size of an image. The operator adapt_template can be called before the template is used with images of another size, or if the image used to create the template had another size. If it is not called explicitly it will be called internally each time another image size is used. The contents of the image is hereby irrelevant; only the width of Image will be considered.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Image which determines the size of the later matching.

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Template number.

Result

If the parameter values are correct, the operator `adapt_template` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TemplateID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_template`, `create_template_rot`, `read_template`

Possible Successors

`set_reference_template`, `best_match`, `fast_match`, `fast_match_mg`, `set_offset_template`, `best_match_mg`, `best_match_pre_mg`, `best_match_rot`, `best_match_rot_mg`

Module

Matching

`best_match` `( Image : : TemplateID, MaxError, SubPixel : Row, Column, Error )`

Searching the best matching of a template and an image.

`best_match` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator `best_match` performs a matching of the template of `TemplateID` and `Image`. Hereby the template will be moved over the points of `Image` so that the template will lie always inside `Image`. `best_match` works similar to `fast_match`, with the exception, that each time a better match is found the value of `MaxError` is internally updated to a lower value to reduce runtime.

With regard to the parameter `SubPixel`, the position will be indicated by subpixel accuracy. The matching criterion ("displaced frame difference") is defined as follows:

$$error[row, col] = \frac{\sum_{u,v} |\texttt{Image}[row-u, col-v] - \texttt{TemplateID}[u,v]|}{area(TemplateID)}$$

The runtime of the operator depends on the size of the domain of `Image`. Therefore it is important to restrict the domain as far as possible, i.e. to apply the operator only in a very confined "region of interest". The parameter `MaxError` determines the maximal error which the searched position is allowed to have at most. The lower this value is, the faster the operator runs.

`Row` and `Column` return the position of the best match, whereby `Error` indicates the average difference of the gray values. If no position with an error below `MaxError` was found the position $(0, 0)$ and a matching result of 255 for `Error` are returned. In this case `MaxError` has to be set larger.

The maximum error of the position (without noise) is 0.1 pixel. The average error is 0.03 pixel.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Input image inside of which the pattern has to be found.

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Template number.

▷ **MaxError** (input_control) . . . . . real ⇝ *real*
Maximum average difference of the gray values.
Default: 20.0
Suggested values: MaxError ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 17.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0}
Value range: 0 ≤ MaxError ≤ 255
Minimum increment: 1
Recommended increment: 3

▷ **SubPixel** (input_control) . . . . . string ⇝ *string*
Subpixel accuracy in case of *'true'*.
Default: 'false'
List of values: SubPixel ∈ {'true', 'false'}

▷ **Row** (output_control) . . . . . point.y(-array) ⇝ *real*
Row position of the best match.

▷ **Column** (output_control) . . . . . point.x(-array) ⇝ *real*
Column position of the best match.

▷ **Error** (output_control) . . . . . real(-array) ⇝ *real*
Average divergence of the gray values of the best match.

Result

If the parameter values are correct, the operator best_match returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

create_template, read_template, set_offset_template, set_reference_template, adapt_template, draw_region, draw_rectangle1, reduce_domain

Alternatives

fast_match, fast_match_mg, best_match_mg, best_match_pre_mg, best_match_rot, best_match_rot_mg, exhaustive_match, exhaustive_match_mg

Module

Matching

best_match_mg (Image : : TemplateID, MaxError, SubPixel, NumLevels, WhichLevels : Row, Column, Error)

Searching the best gray value matches in a pyramid.

best_match_mg is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

best_match_mg applies gray value matching using an image pyramid. best_match_mg works analogously to best_match, but it is faster due to the use of a pyramid. Input is an image with an optionally reduced domain. The template is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image.

The parameter MaxError specifies the maximum error for template matching. Using smaller values results in a reduced runtime but it is possible that the pattern might be missed. The value of MaxError has to be chosen larger compared with best_match, because the error at higher levels of the pyramid is often larger.

SubPixel specifies if the result is calculated with sub pixel accuracy or not. A value of *1* for NumLevels results in an operator similar to best_match, i.e. only the original gray values are used. For values larger than *1*, the algorithm starts at the lowest resolution and searches for a position with the lowest matching error. At the next higher resolution this position is refined. This is continued up to the maximum resolution (WhichLevels = *'all'*). As an alternative Method the mode WhichLevels with value *'original'* can be used. In this case not only the position with the lowest error but all points below MaxError are analyzed further on in the next higher resolution. This method is slower but it is more stable and the possibility to miss the correct position is very low. In this case it is often possible to start with a lower resolution (higher level in Pyramid, i.e. larger value for NumLevels) which leads to a reduced runtime. Besides the values *'all'* and *'original'* for WhichLevels you can specify the pyramid level explicitly where to switch between a "match all" and the "best match". Here *0* corresponds to *'original'* and NumLevels - 1 is equivalent to *'all'*. A value in-between is in most cases a good compromise between speed and a stable detection. A larger value for WhichLevels results in a reduced runtime, a smaller value results in a more stable detection. The value of NumLevels has to be equal or smaller than the value used to create the template.

The position of the found matching position is returned in Row and Column. The corresponding error is given in Error. If no point below MaxError is found a value of *255* for Error and *0* for Row and Column is returned. If the desired object is missed (no object found or wrong position) you have to set MaxError higher or WhichLevels lower. Check also if the illumination has changed (see set_offset_template).

The maximum error of the position (without noise) is 0.1 pixel. The average error is 0.03 pixel.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Input image inside of which the pattern has to be found.

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Template number.

▷ **MaxError** (input_control) . . . . . real ⇝ *real*
Maximal average difference of the gray values.
Default: 30.0
Suggested values: MaxError ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 17.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0}
Value range: 0 ≤ MaxError ≤ 255
Minimum increment: 1
Recommended increment: 3

▷ **SubPixel** (input_control) . . . . . string ⇝ *string*
Exactness in subpixels in case of *'true'*.
Default: 'false'
List of values: SubPixel ∈ {'true', 'false'}

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer*
Number of the used resolution levels.
Default: 4
Suggested values: NumLevels ∈ {1, 2, 3, 4, 5, 6}

▷ **WhichLevels** (input_control) . . . . . integer ⇝ *integer* / string
Resolution level up to which the method "best match" is used.
Default: 2
Suggested values: WhichLevels ∈ {'all', 'original', 0, 1, 2, 3, 4, 5, 6}

▷ **Row** (output_control) . . . . . point.y ⇝ *real*
Row position of the best match.

▷ **Column** (output_control) . . . . . point.x ⇝ *real*
Column position of the best match.

▷ **Error** (output_control) . . . . . real ⇝ *real*
Average divergence of the gray values in the best match.

Result

If the parameter values are correct, the operator best_match_mg returns the value 2 (H_MSG_TRUE).

If the input is empty (no input images are available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`create_template`, `read_template`, `adapt_template`, `draw_region`, `draw_rectangle1`, `reduce_domain`, `set_reference_template`, `set_offset_template`

Alternatives

`fast_match`, `fast_match_mg`, `best_match`, `best_match_pre_mg`, `best_match_rot`, `best_match_rot_mg`, `exhaustive_match`, `exhaustive_match_mg`

Module

Matching

best_match_pre_mg (ImagePyramid : : TemplateID, MaxError, SubPixel, NumLevels, WhichLevels : Row, Column, Error)

Searching the best gray value matches in a pre generated pyramid.

best_match_pre_mg is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

best_match_pre_mg applies gray value matching using an image pyramid. best_match_pre_mg works analogously to best_match_mg, but it makes use of pre calculated pyramid which has to be generated beforehand using gen_gauss_pyramid. This reduces runtime if more than one match has to be done or the pyramid has be used otherwise. The pyramid has to be generated using the zooming factor *0.5* and the mode *'constant'*.

Parameters

▷ **ImagePyramid** (input_object) . . . . . singlechannelimage-array ⇝ *object :* byte
Image pyramid inside of which the pattern has to be found.

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Template number.

▷ **MaxError** (input_control) . . . . . real ⇝ *real*
Maximal average difference of the gray values.
Default: 30.0
Suggested values: MaxError ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 17.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0}
Value range: 0 ≤ MaxError ≤ 255
Minimum increment: 1
Recommended increment: 3

▷ **SubPixel** (input_control) . . . . . string ⇝ *string*
Exactness in subpixels in case of *'true'*.
Default: 'false'
List of values: SubPixel ∈ {'true', 'false'}

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer*
Number of the used resolution levels.
Default: 3
Suggested values: NumLevels ∈ {1, 2, 3, 4, 5, 6}

▷ **WhichLevels** (input_control) . . . . . integer ⇝ *integer* / string
Resolution level up to which the method "best match" is used.
Default: 'original'
Suggested values: WhichLevels ∈ {'all', 'original', 0, 1, 2, 3, 4, 5, 6}

▷ **Row** (output_control) . . . . . point.y ⇝ *real*
Row position of the best match.

▷ **Column** (output_control) . . . . . point.x ⇝ *real*
Column position of the best match.

▷ **Error** (output_control) . . . . . real ⇝ *real*
Average divergence of the gray values in the best match.

Result

If the parameter values are correct, the operator `best_match_pre_mg` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input images are available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_gauss_pyramid`, `create_template`, `read_template`, `adapt_template`, `draw_region`, `draw_rectangle1`, `reduce_domain`, `set_reference_template`

Alternatives

`fast_match`, `fast_match_mg`, `exhaustive_match`, `exhaustive_match_mg`

Module

Matching

best_match_rot (Image : : TemplateID, AngleStart, AngleExtend, MaxError, SubPixel : Row, Column, Angle, Error)

Searching the best matching of a template and an image with rotation.

best_match_rot is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator `best_match_rot` performs a matching of the template of `TemplateID` and `Image`. It works similar to `best_match` with the extension that the pattern can be rotated. The parameters `AngleStart` and `AngleExtend` define the maximum rotation of the pattern: `AngleStart` specifies the maximum counter clockwise rotation and `AngleExtend` the maximum clockwise rotation relative to this angle. Both values have to smaller or equal to the values used for the creation of the pattern (see `create_template_rot`). In addition to `best_match` `best_match_rot` returns the rotation angle of the pattern in `Angle` (radiant). The accuracy of this angle depends on the parameter AngleStep of `create_template_rot`. In the case of `SubPixel` = *'true'* the position and the angle are calculated with "sub pixel" accuracy.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Input image inside of which the pattern has to be found.

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Template number.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest Rotation of the pattern.
Default: -0.39
Suggested values: `AngleStart` ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtend** (input_control) . . . . . angle.rad ⇝ *real*
Maximum positive Extension of `AngleStart`.
Default: 0.79
Suggested values: `AngleExtend` ∈ {6.28, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtend > 0

▷ **MaxError** (input_control) . . . . . real ⇝ *real*
Maximum average difference of the gray values.
Default: 30.0
Suggested values: MaxError ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 17.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0}
Value range: 0 ≤ MaxError ≤ 255
Minimum increment: 1
Recommended increment: 3

▷ **SubPixel** (input_control) . . . . . string ⇝ *string*
Subpixel accuracy in case of *'true'*.
Default: 'false'
List of values: SubPixel ∈ {'true', 'false'}

▷ **Row** (output_control) . . . . . point.y(-array) ⇝ *real*
Row position of the best match.

▷ **Column** (output_control) . . . . . point.x(-array) ⇝ *real*
Column position of the best match.

▷ **Angle** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Rotation angle of pattern.

▷ **Error** (output_control) . . . . . real(-array) ⇝ *real*
Average divergence of the gray values of the best match.

Result

If the parameter values are correct, the operator best_match_rot returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

create_template_rot, read_template, set_offset_template, set_reference_template, adapt_template, draw_region, draw_rectangle1, reduce_domain

Alternatives

best_match_rot_mg

See also

best_match, best_match_mg

Module

Matching

best_match_rot_mg (Image : : TemplateID, AngleStart, AngleExtend, MaxError, SubPixel, NumLevels : Row, Column, Angle, Error)

Searching the best matching of a template and a pyramid with rotation.

best_match_rot_mg is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator best_match_rot_mg performs a matching of the template of TemplateID and Image. It works similar to best_match_mg with the extension that the pattern can be rotated analogously to best_match_rot. The parameters AngleStart and AngleExtend define the maximum rotation of the pattern: AngleStart specifies the maximum counter clockwise rotation and AngleExtend the maximum clockwise rotation relative to this angle. Both values have to smaller or equal to the values used for the creation of the pattern (see create_template_rot). In addition to best_match_mg best_match_rot_mg returns the rotation angle of the pattern in Angle (radiant).

The value of MaxError must be set larger in comparison with the operator best_match_rot, because often the error is larger at higher levels of the pyramid.

In the case of SubPixel = *'true'* the position and the angle are calculated with "sub pixel" accuracy.

The value of NumLevels has to equal or smaller than the value used to create the template.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte
Input image inside of which the pattern has to be found.

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Template number.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest Rotation of the pattern.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtend** (input_control) . . . . . angle.rad ⇝ *real*
Maximum positive Extension of AngleStart.
Default: 0.79
Suggested values: AngleExtend ∈ {6.28, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtend > 0

▷ **MaxError** (input_control) . . . . . real ⇝ *real*
Maximum average difference of the gray values.
Default: 40.0
Suggested values: MaxError ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 17.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0}
Value range: 0 ≤ MaxError ≤ 255
Minimum increment: 1
Recommended increment: 1

▷ **SubPixel** (input_control) . . . . . string ⇝ *string*
Subpixel accuracy in case of *'true'*.
Default: 'false'
List of values: SubPixel ∈ {'true', 'false'}

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer*
Number of the used resolution levels.
Default: 3
Suggested values: NumLevels ∈ {1, 2, 3, 4, 5, 6}

▷ **Row** (output_control) . . . . . point.y(-array) ⇝ *real*
Row position of the best match.

▷ **Column** (output_control) . . . . . point.x(-array) ⇝ *real*
Column position of the best match.

▷ **Angle** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Rotation angle of pattern.

▷ **Error** (output_control) . . . . . real(-array) ⇝ *real*
Average divergence of the gray values of the best match.

Result

If the parameter values are correct, the operator best_match_rot_mg returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

create_template_rot, set_reference_template, set_offset_template, adapt_template, draw_region, draw_rectangle1, reduce_domain

Alternatives

best_match_rot, best_match_mg, find_ncc_model, find_ncc_models

See also

fast_match

Module

Matching

clear_template (: : TemplateID :)

Deallocation of the memory of a template.

clear_template is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator clear_template deallocates the memory of a template which has been created by create_template or create_template_rot. After execution of the operator clear_template the template can no longer be used. The value of TemplateID is not valid. However, the number can be used again by further calls of create_template or create_template_rot.

Parameters

▷ **TemplateID** (input_control) ..... template ⇝ *handle*
Template number.

Result

If the number of the template is valid, the operator clear_template returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TemplateID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_template, create_template_rot, read_template, write_template

Module

Matching

create_template (Template : : FirstError, NumLevel, Optimize, GrayValues : TemplateID)

Preparing a pattern for template matching.

create_template is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator create_template preprocesses a pattern (Template), which is passed as an image, for the template matching. After the transformation, a number (TemplateID) is assigned to the template for being used in the further process. The shape and the size of Template can be chosen arbitrarily. You have to be aware, that the matching is only applied to that part of an image where Template fits completely into the image.

The template has been chosen such that it contains no pixels of the (changing) background. Here you can make use of the arbitrary shape of a template which is not restricted to a rectangle. To create a template region you can use segmentation operators like `threshold` or one of the draw_* operators. In the case of sub pixel accurate matching `Template` has in addition to be one pixel smaller than the pattern (i.e. one pixel border to the changing background). This can be done e.g., by applying the operator `erosion_circle`.

The parameter `NumLevel` specifies the number of pyramid levels (`NumLevel` = 1 means only original gray values) which can be used for matching. The number of levels used later for matching will be below or equal this value. If the pattern becomes too small due to zooming, the maximum number of pyramid levels is automatically reduced (without error message).

The parameter `GrayValues` defines, whether the original gray values (*'original'*, *'normalized'*) or the edge amplitude (*'gradient'*, *'sobel'*) is used. With *'original'* the sum of the differences is used as feature which is very stable and fast if there is no change in illumination. *'normalized'* is used if the illumination changes. The method is a bit slower and not quite as stable. Note that *'normalized'* allows to compensate additive illumination changes. If also multiplicative variations of the gray values occur, correlation-based matching should be used (`create_ncc_model`). If there is no change in illumination the mode *'original'* should be used. The edge amplitude is another method to be invariant to changes in illumination. The disadvantage is the increased execution time and the higher sensitivity to changes in the shape of the pattern. The mode *'gradient'* is slightly faster but more sensitive to noise.

The maximum error for matching has typically to be chosen higher when using the edge amplitude. The mode chosen by `GrayValues` leads automatically to calling the appropriate filter during the matching, if necessary.

As an alternative to the gradient approach the operator `set_offset_template` can be used, if the change in illumination is known.

The parameter `Optimize` specifies if the pattern has to be optimized for runtime. This optimization results in a longer time to create the template but reduces the time for matching. In addition the optimization leads to a more stable matching, i.e., the possibilty to miss good matches is reduced. The optimization process selects the most stable and significant gray values to be tested first during the matching process. Using this technique a wrong match can be eliminated very early.

The reference position for the template is its center of gravity. I.e. if you apply the template to the original image the center of gravity is returned. This default reference can be adapted using the operator `set_reference_template`.

In sub pixel mode a special position correction is calculated which is added after each matching: The template is applied to the original image and the difference between the found position and the center of gravity is used as a correction vector. This is important for patterns in a textured context or for asymmetric pattern. For most templates this correction vector is near null.

Before the use of the template, which is stored independently of the image size, it can be adapted explicitly to the size of a definite image size by using `adapt_template`.

Parameters

- ▷ **`Template`** (input_object) . . . . . singlechannelimage ⇝ *object :* byte
  Input image whose domain will be processed for the pattern matching.
- ▷ **`FirstError`** (input_control) . . . . . integer ⇝ *integer*
  Not yet in use.
  Default: 255
  List of values: `FirstError` ∈ {255}
- ▷ **`NumLevel`** (input_control) . . . . . integer ⇝ *integer*
  Maximal number of pyramid levels.
  Default: 4
  List of values: `NumLevel` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
- ▷ **`Optimize`** (input_control) . . . . . string ⇝ *string*
  Kind of optimizing.
  Default: 'sort'
  List of values: `Optimize` ∈ {'none', 'sort'}
- ▷ **`GrayValues`** (input_control) . . . . . string ⇝ *string*
  Kind of gray values.
  Default: 'original'
  List of values: `GrayValues` ∈ {'original', 'normalized', 'gradient', 'sobel'}

▷ **TemplateID** (output_control) . . . . . template ⇝ *handle*
Template number.

Result

If the parameters are valid, the operator create_template returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

draw_region, reduce_domain, threshold

Possible Successors

adapt_template, set_reference_template, clear_template, write_template, set_offset_template, best_match, best_match_mg, fast_match, fast_match_mg

Alternatives

create_ncc_model, create_template_rot, read_template

Module

Matching

create_template_rot (Template : : NumLevel, AngleStart, AngleExtend, AngleStep, Optimize, GrayValues : TemplateID)

Preparing a pattern for template matching with rotation.

create_template_rot is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator create_template_rot preprocesses a pattern, which is passed as an image, for the template matching. An extension to create_template the matching can applied to rotated patterns. The parameters AngleStart and AngleExtend define the maximum rotation of the pattern: AngleStart specifies the maximum counter clockwise rotation and AngleExtend the maximum clockwise rotation relative to this angle. Therefore AngleExtend has to be smaller than 2π. With the parameter AngleStep the maximum angle resolution (on the highest resolution level) can be specified.

You have to be aware, that all possible rotations are calculated beforehand to reduce runtime during matching. This leads to a higher execution time for create_template_rot and high memory requirements for the template. The amount of memory depends on the parameters AngleExtend and AngleStep. The number of pyramid levels can be neglected. If A is the number of pixels of Template, the memory M needed for the template in byte is about:

$$M = \frac{A * 12 * \texttt{AngleExtend}}{\texttt{AngleStep}}$$

After the transformation, a number (TemplateID) is assigned to the template for being used in the further process.

A description of the other parameters can be found at the operator create_template.

Attention

You have to be aware, that depending on the resolution a large number of pre calculated patterns have to be created which might result in a large amount of memory needed.

Parameters

- ▷ **Template** (input_object) . . . . . singlechannelimage ⇝ *object : byte*
  Input image whose domain will be processed for the pattern matching.
- ▷ **NumLevel** (input_control) . . . . . integer ⇝ *integer*
  Maximal number of pyramid levels.
  Default: 4
  List of values: NumLevel ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
- ▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
  Smallest Rotation of the pattern.
  Default: -0.39
  Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}
- ▷ **AngleExtend** (input_control) . . . . . angle.rad ⇝ *real*
  Maximum positive Extension of AngleStart.
  Default: 0.79
  Suggested values: AngleExtend ∈ {6.28, 3.14, 1.57, 0.79, 0.39}
  Restriction: AngleExtend > 0
- ▷ **AngleStep** (input_control) . . . . . angle.rad ⇝ *real*
  Step rate (angle precision) of matching.
  Default: 0.0982
  Suggested values: AngleStep ∈ {0.3927, 0.1963, 0.0982, 0.0491, 0.0245}
  Restriction: AngleStep > 0
- ▷ **Optimize** (input_control) . . . . . string ⇝ *string*
  Kind of optimizing.
  Default: 'sort'
  List of values: Optimize ∈ {'none', 'sort'}
- ▷ **GrayValues** (input_control) . . . . . string ⇝ *string*
  Kind of gray values.
  Default: 'original'
  List of values: GrayValues ∈ {'original', 'normalized', 'gradient', 'sobel'}
- ▷ **TemplateID** (output_control) . . . . . template ⇝ *handle*
  Template number.

Result

If the parameters are valid, the operator create_template_rot returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

draw_region, reduce_domain, threshold

Possible Successors

best_match_rot, best_match_rot_mg, adapt_template, set_reference_template, clear_template, set_offset_template, write_template

Alternatives

create_ncc_model, create_template

Module

Matching

deserialize_template (: : SerializedItemHandle : TemplateID)

Deserialize a serialized template.

deserialize_template is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

deserialize_template deserializes a template, that was serialized by serialize_template (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized template is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created template with the handle TemplateID.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **TemplateID** (output_control) ..... template ⇝ *handle*
Template number.

Result

If the parameters are valid, the operator deserialize_template returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_template

Possible Successors

adapt_template, set_reference_template, set_offset_template, best_match, fast_match, best_match_rot

Module

Matching

fast_match (Image : Matches : TemplateID, MaxError :)

Searching all good matches of a template and an image.

fast_match is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator fast_match performs a matching of the template of TemplateID and Image. Hereby the template will be moved over the points of Image so that the template always lies completely inside of Image. The matching criterion ("displaced frame difference") is defined as follows:

$$error[row, col] = \frac{\sum_{u,v} |\texttt{Image}[row-u, col-v] - \texttt{TemplateID}[u,v]|}{area(TemplateID)}$$

The difference between fast_match and exhaustive_match is that the matching for one position is stopped if the error is too high. This leads to a reduced runtime but one might miss correct matches. The runtime of the operator depends mainly on the size of the domain of Image. Therefore it is important to restrict the domain as far as possible, i.e. to apply the operator only in a very confined "region of interest". The parameter MaxError determines the maximal error which the searched position is allowed to show. The lower this value is, the faster the operator runs.

All points which show a matching error smaller than MaxError will be returned in the output region Matches. This region can be used for further processing. For example by using a connection and best_match to find all the matching objects. If no point has a match error below MaxError the empty region (i.e no points) is returned.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
Input image inside of which the pattern has to be found.

▷ **Matches** (output_object) ..... region(-array) ⇝ *object*
All points whose error lies below a certain threshold.

▷ **TemplateID** (input_control) ..... template ⇝ *handle*
Template number.

▷ **MaxError** (input_control) ..... real ⇝ *real*
Maximal average difference of the gray values.
Default: 20.0
Suggested values: MaxError ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 17.0, 20.0, 30.0}
Value range: 0 ≤ MaxError ≤ 255
Minimum increment: 1
Recommended increment: 1

Result

If the parameter values are correct, the operator fast_match returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

create_template, read_template, adapt_template, draw_region, draw_rectangle1, reduce_domain

Possible Successors

connection, best_match

Alternatives

best_match, best_match_mg, fast_match_mg, exhaustive_match, exhaustive_match_mg

Module

Matching

fast_match_mg (Image : Matches : TemplateID, MaxError, NumLevel :)

Searching all good gray value matches in a pyramid.

fast_match_mg is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator fast_match_mg like the operator fast_match performs a matching of the template of TemplateID and Image. In contrast to fast_match, however, the search for good matches starts in scaled down images (pyramid). The number of levels of the pyramid will be determined by NumLevel. Hereby the value 1 indicates that no pyramid will be used. In this case the operator fast_match_mg is equivalent to the operator fast_match. The value 2 triggers the search in the image with half the framesize. The search for all those points showing an error small enough in the scaled down image (error smaller than MaxError) will be refined at the corresponding positions in the original image (Image).

The runtime of matching depends on the parameter MaxError: the larger the value the longer is the processing time, because more points of the pattern have to be tested. If MaxError is to low the pattern will not be found. The value has therefore to be optimized for every application.

NumLevel indicates the number of levels of the pyramid, including the original image. Optionally a second value can be given. This value specifies the number (0..n) of the lowest level which is used the matching. The region found up to this level will then be zoomed to the size of the original level. This can used to increase the runtime in the case that the accuracy does not have to be so high.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object : byte*
Input image inside of which the pattern has to be found.

▷ **Matches** (output_object) . . . . . region(-array) ⇝ *object*
All points which have an error below a certain threshold.

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Template number.

▷ **MaxError** (input_control) . . . . . real ⇝ *real*
Maximal average difference of the gray values.
Default: 30.0
Suggested values: MaxError ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 17.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0}
Value range: 0 ≤ MaxError ≤ 255
Minimum increment: 1
Recommended increment: 3

▷ **NumLevel** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of levels in the pyramid.
Default: 3
Suggested values: NumLevel ∈ {1, 2, 3, 4, 5, 6, 7, 8}

Result

If the parameter values are correct, the operator fast_match_mg returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

create_template, read_template, adapt_template, draw_region, draw_rectangle1, reduce_domain

Alternatives

best_match, best_match_mg, fast_match, exhaustive_match, exhaustive_match_mg

Module

Matching

read_template (: : FileName : TemplateID)

Reading a template from file.

read_template is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator read_template reads a matching template from file which has been written with write_template. The default HALCON file extension for a template is 'gvt'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
file name.
File extension: .gvt

▷ **TemplateID** (output_control) . . . . . template ⇝ *handle*
Template number.

Result

If the file name is valid, the operator read_template returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

adapt_template, set_reference_template, set_offset_template, best_match, fast_match, best_match_rot

Module

Matching

serialize_template (: : TemplateID : SerializedItemHandle)

Serialize a template.

serialize_template is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

serialize_template serializes the data of a template (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_template is converted to a serialized item. The template is defined by the handle TemplateID. The serialized template is returned by the handle SerializedItemHandle and can be deserialized by deserialize_template.

Parameters

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Handle of the template.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_template returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_template, create_template_rot

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_template

Module

Matching

set_offset_template (: : TemplateID, GrayOffset :)

Gray value offset for template.

set_offset_template is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

set_offset_template adds a gray value offset to the template to eliminate gray value changes in the image. The parameter GrayOffset specifies a difference relative to the gray values of the pattern when it was created with create_template (not relative to the last call of set_offset_template). The values of GrayOffset has to be chosen according to the gray values of the image: A brighter image results in a positive value, a darker image results in a negative value. set_offset_template has to be called each time the gray values of the image changes. The gray values can be measured in a reference area using intensity or min_max_gray

Parameters

▷ **TemplateID** (input_control) ..... template ⇝ *handle*
Template number.

▷ **GrayOffset** (input_control) ..... number ⇝ *integer*
Offset of gray values.
Default: 0
Suggested values: GrayOffset ∈ {-10, -5, -2, -1, 0, 1, 2, 5, 10}
Value range: -255 ≤ GrayOffset ≤ 255
Minimum increment: 1
Recommended increment: 1

Result

If the parameter values are correct, the operator set_offset_template returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TemplateID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_template, adapt_template, read_template

Possible Successors

best_match, best_match_mg, best_match_rot, fast_match, fast_match_mg

Module

Matching

set_reference_template (: : TemplateID, Row, Column :)

Define reference position for a matching template.

set_reference_template is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

set_reference_template allows to define a new reference position for a template. As default after calling create_template or create_template_rot the center of gravity of the template is used. Using set_reference_template the reference position can be redefined. In the case of the center of gravity as reference the vector $(0, 0)$ is returned after matching for a null translation of the pattern relative to the image.

Parameters

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Template number.

▷ **Row** (input_control) . . . . . point.y ⇝ *real*
Reference position of template (row).

▷ **Column** (input_control) . . . . . point.x ⇝ *real*
Reference position of template (column).

Result

If the parameter values are correct, the operator set_reference_template returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TemplateID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_template, create_template_rot, read_template, adapt_template

Possible Successors

best_match, best_match_mg, best_match_rot, fast_match, fast_match_mg

Module

Matching

write_template (: : TemplateID, FileName :)

Writing a template to file.

write_template is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11. New applications should use the shape-based or NCC-based operators instead.

The operator write_template writes a matching template to file which can be read again with read_template. The default HALCON file extension for a template is 'gvt'.

Parameters

▷ **TemplateID** (input_control) . . . . . template ⇝ *handle*
Template number.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
file name.
File extension: .gvt

Result

If the file name is valid (permission to write), the operator `write_template` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_template`, `create_template_rot`

Module

Matching

17.10 Matching, Component-Based

`clear_all_component_models` (: : :)

This operator is inoperable. It had the following function: Free the memory of all component models.

`clear_all_component_models` is obsolete, inoperable, and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

.

Attention

`clear_all_component_models` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_component_models` *must not* be used in any application.

Result

`clear_all_component_models` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

`clear_all_training_components` (: : :)

This operator is inoperable. It had the following function: Free the memory of all component training results.

`clear_all_training_components` is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use `clear_all_training_components`. The operator will be removed with HALCON 26.11.

.

Attention

`clear_all_training_components` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_training_components` *must not* be used in any application.

Result

`clear_all_training_components` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

clear_component_model (: : ComponentModelID :)

Free the memory of a component model.

clear_component_model is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

The operator clear_component_model frees the memory of a component model that was created by create_component_model or create_trained_component_model. After calling clear_component_model, the model can no longer be used. The handle ComponentModelID becomes invalid.

Parameters

▷ **ComponentModelID** (input_control) ..... component_model ⇝ *handle*
Handle of the component model.

Result

If the handle of the model is valid, the operator clear_component_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ComponentModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

Matching

clear_training_components (: : ComponentTrainingID :)

Free the memory of a component training result.

clear_training_components is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

The operator clear_training_components frees the memory of a training result that was created by train_model_components. After calling clear_training_components, the training result can no longer be used. The handle ComponentTrainingID becomes invalid.

Parameters

▷ **ComponentTrainingID** (input_control) . . . . . component_training ⇝ *handle*
Handle of the training result.

Result

If the handle of the training result is valid, the operator `clear_training_components` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ComponentTrainingID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

Matching

```
cluster_model_components (
    TrainingImages : ModelComponents : ComponentTrainingID,
    AmbiguityCriterion, MaxContourOverlap, ClusterThreshold : )
```

Adopt new parameters that are used to create the model components into the training result.

`cluster_model_components` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

With `cluster_model_components` you can modify parameters *after* a first training has been performed using `train_model_components`. `cluster_model_components` sets the criterion `AmbiguityCriterion` that is used to solve the ambiguities, the maximum contour overlap `MaxContourOverlap`, and the cluster threshold of the training result `ComponentTrainingID` to the specified values. A detailed description of these parameters can be found in the documentation of `train_model_components`. By modifying these parameters, the way in which the initial components are merged into rigid model components changes. For example, the greater the cluster threshold is chosen, the fewer initial components are merged. You can select suitable parameter values interactively by repeatedly calling `inspect_clustered_components` with different parameter values and then setting the chosen values by using `get_training_components`.

The rigid model components are returned in `ModelComponents`. In order to receive reasonable results, it is essential that the same training images that were used to perform the training with `train_model_components` are passed in `TrainingImages`. The pose of the newly clustered components within the training images is determined using the shape-based matching. As in `train_model_components`, one can decide whether the shape models should be pregenerated by using `set_system('pregenerate_shape_models',...)`. Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. `set_system('border_shape_models',...)` can be used to determine whether the models must lie completely within the training images or whether they can extend partially beyond the image border.

Parameters

▷ **TrainingImages** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Training images that were used for training the model components.

▷ **ModelComponents** (output_object) . . . . . region(-array) ⇝ *object*
Contour regions of rigid model components.

▷ **ComponentTrainingID** (input_control) . . . . . component_training ⇝ *handle*
Handle of the training result.

▷ **AmbiguityCriterion** (input_control) ..... string ⇝ *string*
Criterion for solving the ambiguities.
Default: 'rigidity'
List of values: AmbiguityCriterion ∈ {'distance', 'orientation', 'distance_orientation', 'rigidity'}

▷ **MaxContourOverlap** (input_control) ..... real ⇝ *real*
Maximum contour overlap of the found initial components.
Default: 0.2
Suggested values: MaxContourOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= MaxContourOverlap && MaxContourOverlap <= 1

▷ **ClusterThreshold** (input_control) ..... real ⇝ *real*
Threshold for clustering the initial components.
Default: 0.5
Suggested values: ClusterThreshold ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: 0 <= ClusterThreshold && ClusterThreshold <= 1

Result

If the parameter values are correct, the operator cluster_model_components returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

```
create_component_model ( ModelImage,
    ComponentRegions : : VariationRow, VariationColumn,
    VariationAngle, AngleStart, AngleExtent, ContrastLowComp,
    ContrastHighComp, MinSizeComp, MinContrastComp, MinScoreComp,
    NumLevelsComp, AngleStepComp, OptimizationComp, MetricComp,
    PregenerationComp : ComponentModelID, RootRanking )
```

Prepare a component model for matching based on explicitly specified components and relations.

create_component_model is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

create_component_model prepares patterns, which are passed in the form of a model image ModelImage and several regions ComponentRegions, as a component model for matching. The output parameter ComponentModelID is a handle for this model, which is used in subsequent calls to find_component_model. In contrast to create_trained_component_model, no preceding training with train_model_components needs to be performed before calling create_component_model.

Each of the regions passed in ComponentRegions describes a separate model component. Later, the index of a component region in ComponentRegions is used to denote the model component. The reference point of a component is the center of gravity of its associated region, which is passed in ComponentRegions. It can be calculated by calling area_center.

The relative movements (relations) of the model components can be set by passing VariationRow, VariationColumn, and VariationAngle. Because directly passing the relations is complicated, instead of the relations the variations of the model components are passed. The variations describe the movements of the components independently from each other relative to their poses in the model image ModelImage. The parameters VariationRow and VariationColumn describe the movement of the components in row and column direction by $\pm\frac{1}{2}$VariationRow and $\pm\frac{1}{2}$VariationColumn, respectively. The parameter VariationAngle

describes the angle variation of the component by $\pm\frac{1}{2}$VariationAngle. Based on these values, the relations are automatically computed. The three parameters must either contain one element, in which case the parameter is used for all model components, or must contain the same number of elements as ComponentRegions, in which case each parameter element refers to the corresponding model component in ComponentRegions.

The parameters AngleStart and AngleExtent determine the range of possible rotations of the component model in an image.

Internally, a separate shape model is built for each model component (see create_shape_model). Therefore, the parameters ContrastLowComp, ContrastHighComp, MinSizeComp, MinContrastComp, MinScoreComp, NumLevelsComp, AngleStepComp, OptimizationComp, MetricComp, and PregenerationComp correspond to the parameters of create_shape_model, with the following differences: First, in the parameter Contrast of create_shape_model the upper as well as the lower threshold for the hysteresis threshold method can be passed. Additionally, a third value, which suppresses small connected contour regions, can be passed. In contrast, the operator create_component_model offers three separate parameters ContrastHighComp, ContrastLowComp, and MinScoreComp in order to set these three values. Consequently, also the automatic computation of the contrast threshold(s) is different. If both hysteresis threshold should be automatically determined, both ContrastLowComp and ContrastHighComp must be set to *'auto'*. In contrast, if only one threshold value should be determined, ContrastLowComp must be set to *'auto'* while ContrastHighComp must be set to an arbitrary value different from *'auto'*. Secondly, the parameter Optimization of create_shape_model provides the possibility to reduce the number of model points as well as the possibility to completely pregenerate the shape model. In contrast, the operator create_trained_component_model uses a separate parameter PregenerationComp in order to decide whether the shape models should be completely pregenerated or not. A third difference concerning the parameter MinScoreComp should be noted. When using the shape-based matching, this parameter needs not be passed when preparing the shape model using create_shape_model, but only during the search using find_shape_model. In contrast, when preparing the component model it is favorable to analyze rotational symmetries of the model components and similarities between the model components. However, this analysis only leads to meaningful results if the value for MinScoreComp that is used during the search (see find_component_model) is already approximately known.

In addition to the parameters ContrastLowComp, ContrastHighComp, and MinSizeComp also the parameters MinContrastComp, NumLevelsComp, AngleStepComp, and OptimizationComp can be automatically determined by passing *'auto'* for the respective parameters.

All component-specific input parameters (parameter names terminate with the suffix Comp) must either contain one element, in which case the parameter is used for all model components, or must contain the same number of elements as the number of regions in ComponentRegions, in which case each parameter element refers to the corresponding element in ComponentRegions.

In addition to the individual shape models, the component model also contains information about the way the single model components must be searched relative to each other using find_component_model in order to minimize the computation time of the search. For this, the components are represented in a tree structure. First, the component that stands at the root of this search tree (root component) is searched. Then, the remaining components are searched relative to the pose of their predecessor in the search tree.

The root component can be passed as an input parameter of find_component_model during the search. To what extent a model component is suited to act as the root component depends on several factors. In principle, a model component that can be found in the image with a high probability should be chosen. Therefore, a component that is sometimes occluded to a high degree or that is missing in some cases is not well suited to act as the root component. Additionally, the computation time that is associated with the root component during the search can serve as a criterion. A ranking of the model components that is based on the latter criterion is returned in RootRanking. In this parameter the indices of the model components are sorted in descending order according to their associated search time, i.e., RootRanking[0] contains the index of the model component that, chosen as root component, allows the fastest search. Note that the ranking returned in RootRanking represents only a coarse estimation. Furthermore, the calculation of the root ranking assumes that the image size as well as the value of the system parameter *'border_shape_models'* are identical when calling create_component_model and find_component_model.

Parameters

▷ **ModelImage** (input_object) ..... (multichannel-)image ⇝ *object :* byte / uint2
Input image from which the shape models of the model components should be created.

▷ **ComponentRegions** (input_object) ..... region-array ⇝ *object*
Input regions from which the shape models of the model components should be created.

▷ **VariationRow** (input_control) . . . . . integer(-array) ⇝ *integer*
Variation of the model components in row direction.
Suggested values: VariationRow ∈ {0, 1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 100, 150}
Restriction: VariationRow >= 0

▷ **VariationColumn** (input_control) . . . . . integer(-array) ⇝ *integer*
Variation of the model components in column direction.
Suggested values: VariationColumn ∈ {0, 1, 2, 3, 4, 5, 10, 20, 30, 40, 50, 100, 150}
Restriction: VariationColumn >= 0

▷ **VariationAngle** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Angle variation of the model components.
Suggested values: VariationAngle ∈ {0.0, 0.017, 0.035, 0.05, 0.07, 0.09, 0.17, 0.35, 0.52, 0.67, 0.87}
Restriction: VariationAngle >= 0

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the component model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation of the component model.
Default: 0.79
Suggested values: AngleExtent ∈ {6.28, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **ContrastLowComp** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Lower hysteresis threshold for the contrast of the components in the model image.
Default: 'auto'
Suggested values: ContrastLowComp ∈ {'auto', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}
Restriction: ContrastLowComp > 0

▷ **ContrastHighComp** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Upper hysteresis threshold for the contrast of the components in the model image.
Default: 'auto'
Suggested values: ContrastHighComp ∈ {'auto', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}
Restriction: ContrastHighComp > 0 && ContrastHighComp >= ContrastLowComp

▷ **MinSizeComp** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Minimum size of the contour regions in the model.
Default: 'auto'
Suggested values: MinSizeComp ∈ {'auto', 0, 5, 10, 20, 30, 40}
Restriction: MinSizeComp >= 0

▷ **MinContrastComp** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Minimum contrast of the components in the search images.
Default: 'auto'
Suggested values: MinContrastComp ∈ {'auto', 10, 20, 20, 40}
Restriction: MinContrastComp <= ContrastLowComp && MinContrastComp >= 0

▷ **MinScoreComp** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the components to be found.
Default: 0.5
Suggested values: MinScoreComp ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= MinScoreComp && MinScoreComp <= 1

▷ **NumLevelsComp** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Maximum number of pyramid levels for the components.
Default: 'auto'
List of values: NumLevelsComp ∈ {'auto', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStepComp** (input_control) . . . . . angle.rad(-array) ⇝ *real* / string
Step length of the angles (resolution) for the components.
Default: 'auto'
Suggested values: AngleStepComp ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStepComp >= 0

▷ **OptimizationComp** (input_control) . . . . . string ⇝ *string*
Kind of optimization for the components.
Default: 'auto'
List of values: OptimizationComp ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high'}

▷ **MetricComp** (input_control) . . . . . string(-array) ⇝ *string*
Match metric used for the components.
Default: 'use_polarity'
List of values: MetricComp ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **PregenerationComp** (input_control) . . . . . string(-array) ⇝ *string*
Complete pregeneration of the shape models for the components if equal to *'true'*.
Default: 'false'
List of values: PregenerationComp ∈ {'true', 'false'}

▷ **ComponentModelID** (output_control) . . . . . component_model ⇝ *handle*
Handle of the component model.

▷ **RootRanking** (output_control) . . . . . integer(-array) ⇝ *integer*
Ranking of the model components expressing the suitability to act as the root component.

Result

If the parameters are valid, the operator create_component_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

Matching

create_trained_component_model (: : ComponentTrainingID, AngleStart, AngleExtent, MinContrastComp, MinScoreComp, NumLevelsComp, AngleStepComp, OptimizationComp, MetricComp, PregenerationComp : ComponentModelID, RootRanking)

Prepare a component model for matching based on trained components.

create_trained_component_model is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

create_trained_component_model prepares the training result, which is passed in ComponentTrainingID, as a component model for matching. The output parameter ComponentModelID is a handle for this model, which is used in subsequent calls to find_component_model. In contrast to create_component_model, the model components must have been previously trained using train_model_components before calling create_trained_component_model.

The parameters AngleStart and AngleExtent determine the range of possible rotations of the component model in an image.

Internally, a separate shape model is built for each model component (see create_shape_model). Therefore, the parameters MinContrastComp, MinScoreComp, NumLevelsComp, AngleStepComp, OptimizationComp, MetricComp, and PregenerationComp correspond to the parameters of create_shape_model, with the following differences: First, the parameter Optimization of create_shape_model provides the possibility to reduce the number of model points as well as the possibility to completely pregenerate the shape model. In contrast, the operator create_trained_component_model

uses a separate parameter PregenerationComp in order to decide whether the shape models should be completely pregenerated or not. A second difference concerning the parameter MinScoreComp should be noted. When using the shape-based matching, this parameter needs not be passed when preparing the shape model using create_shape_model but only during the search using find_shape_model. In contrast, when preparing the component model it is favorable to analyze rotational symmetries of the model components and similarities between the model components. However, this analysis only leads to meaningful results if the value for MinScoreComp that is used during the search (see find_component_model) is already approximately known. After the search with find_component_model the pose parameters of the components in a search image are returned. Note that the pose parameters refer to the reference points of the components. The reference point of a component is the center of gravity of its associated region that is returned in ModelComponents of train_model_components.

The parameters MinContrastComp, NumLevelsComp, AngleStepComp, and OptimizationComp can be automatically determined by passing *'auto'* for the respective parameters.

All component-specific input parameters (parameter names terminate with the suffix Comp) must either contain one element, in which case the parameter is used for all model components, or must contain the same number of elements as the number of model components contained in ComponentTrainingID, in which case each parameter element refers to the corresponding component in ComponentTrainingID.

In addition to the individual shape models, the component model also contains information about the way the single model components must be searched relative to each other using find_component_model in order to minimize the computation time of the search. For this, the components are represented in a tree structure. First, the component that stands at the root of this search tree (root component) is searched. Then, the remaining components are searched relative to the pose of their predecessor in the search tree.

The root component can be passed as an input parameter of find_component_model during the search. To what extent a model component is suited to act as root component depends on several factors. In principle, a model component that can be found in the image with a high probability should be chosen. Therefore, a component that is sometimes occluded to a high degree or that is missing in some cases is not well suited to act as root component. Additionally, the computation time that is associated with the root component during the search can serve as a criterion. A ranking of the model components that is based on the latter criterion is returned in RootRanking. In this parameter the indices of the model components are sorted in descending order according to their associated computation time, i.e., RootRanking[0] contains the index of the model component that, chosen as root component, allows the fastest search. Note that the ranking returned in RootRanking represents only a coarse estimation. Furthermore, the calculation of the root ranking assumes that the image size as well as the value of the system parameter *'border_shape_models'* are identical when calling create_trained_component_model and find_component_model.

Parameters

▷ **ComponentTrainingID** (input_control) . . . . . component_training ⇝ *handle*
Handle of the training result.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the component model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation of the component model.
Default: 0.79
Suggested values: AngleExtent ∈ {6.28, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **MinContrastComp** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Minimum contrast of the components in the search images.
Default: 'auto'
Suggested values: MinContrastComp ∈ {'auto', 10, 20, 20, 40}
Restriction: MinContrastComp >= 0

▷ **MinScoreComp** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the components to be found.
Default: 0.5
Suggested values: MinScoreComp ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= MinScoreComp && MinScoreComp <= 1

▷ **NumLevelsComp** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Maximum number of pyramid levels for the components.
Default: 'auto'
List of values: NumLevelsComp ∈ {'auto', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStepComp** (input_control) . . . . . angle.rad(-array) ⇝ *real* / string
Step length of the angles (resolution) for the components.
Default: 'auto'
Suggested values: AngleStepComp ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStepComp >= 0

▷ **OptimizationComp** (input_control) . . . . . string ⇝ *string*
Kind of optimization for the components.
Default: 'auto'
List of values: OptimizationComp ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high'}

▷ **MetricComp** (input_control) . . . . . string(-array) ⇝ *string*
Match metric used for the components.
Default: 'use_polarity'
List of values: MetricComp ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **PregenerationComp** (input_control) . . . . . string(-array) ⇝ *string*
Complete pregeneration of the shape models for the components if equal to *'true'*.
Default: 'false'
List of values: PregenerationComp ∈ {'true', 'false'}

▷ **ComponentModelID** (output_control) . . . . . component_model ⇝ *handle*
Handle of the component model.

▷ **RootRanking** (output_control) . . . . . integer(-array) ⇝ *integer*
Ranking of the model components expressing the suitability to act as the root component.

Result

If the parameters are valid, the operator create_trained_component_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

Matching

deserialize_component_model (
: : SerializedItemHandle : ComponentModelID)

Deserialize a serialized component model.

deserialize_component_model is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

deserialize_component_model deserializes a component model, that was serialized by serialize_component_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized component model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created component model with the handle SerializedItemHandle.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **ComponentModelID** (output_control) ..... component_model ⇝ *handle*
Handle of the component model.

Result

If the parameters are valid, the operator deserialize_component_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

deserialize_training_components (
: : SerializedItemHandle : ComponentTrainingID)

Deserialize a component training result.

deserialize_training_components is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

deserialize_training_components deserializes a component training result, that was serialized by serialize_training_components (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized component training result is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created component training result with the handle ComponentTrainingID.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **ComponentTrainingID** (output_control) ..... component_training ⇝ *handle*
Handle of the training result.

Result

If the parameters are valid, the operator deserialize_training_components returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

find_component_model (Image : : ComponentModelID, RootComponent, AngleStartRoot, AngleExtentRoot, MinScore, NumMatches, MaxOverlap, IfRootNotFound, IfComponentNotFound, PosePrediction, MinScoreComp, SubPixelComp, NumLevelsComp, GreedinessComp : ModelStart, ModelEnd, Score, RowComp, ColumnComp, AngleComp, ScoreComp, ModelComp)

Find the best matches of a component model in an image.

find_component_model is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

The operator find_component_model finds the best NumMatches instances of the component model ComponentModelID in the input image Image.

The result of the search can be visualized using get_found_component_model. Also the operator can be used to extract the component matches of a certain component model instance.

Further information about input parameters

ComponentModelID: Handle of the component model.

The model must have been created previously by calling create_trained_component_model or create_component_model, or read in using read_component_model.

RootComponent: Index of the root component.

The components of the component model ComponentModelID are represented in a tree structure. The component that stands at the root of this search tree is the root component.

The root component is searched within the full parameter space, i.e., at all allowed positions and in the allowed range of orientations (see below). In contrast, the remaining components are searched relative to the pose of their predecessor in the search tree within a restricted parameter space that is computed from the relations (recursive search).

To what extent a model component is suited to act as root component depends on several factors. In principle, a model component that can be found in the image with a high probability, should be chosen. Therefore, a component that is sometimes occluded to a high degree or that is missing in some cases is not well suited to act as root component. For the behavior in such cases, see the entry IfRootNotFound below.

A different possible criterion is the computation time that is associated with the root component during the search. A ranking of the model components that is based on the latter criterion is returned in RootRanking of the operator create_trained_component_model or create_component_model, respectively. If the complete ranking is passed in RootComponent, the first value RootComponent[0] is automatically selected as the root component.

AngleStartRoot and AngleExtentRoot: Specify the allowed angle range (in [rad]) within which the root component is searched.

If necessary, the range of rotations is clipped to the range given when the component model was created with create_trained_component_model or create_component_model, respectively. The angle range for each component can be queried with get_shape_model_params. The necessary handle of the corresponding shape model can be obtained using get_component_model_params.

MinScore: Determines what score a potential match of the component model must at least have to be regarded as an instance.

If the component model can be expected never to be occluded in the images, MinScore may be set as high as *0.8* or even *0.9*. The value of this parameter only slightly influences the computation time. An exception is the case of IfRootNotFound set to *'select_new_root'* (see below).

NumMatches: Determines the maximum number of returned instances.

If fewer than NumMatches are found, only that number is returned, i.e., the parameter MinScore takes precedence over NumMatches. In case more than NumMatches instances with a score greater than MinScore are found in the image, only the best NumMatches instances are returned. However, if all model instances exceeding MinScore in the image should be found, NumMatches must be set to *0*.

MaxOverlap: Determines by what fraction two instances may at most overlap, whereby this fraction is a number between 0 and 1.

In some cases, found instances only differ in the pose of one or a few components. If two instances overlap each other by more than MaxOverlap only the best instance is returned. This means, for MaxOverlap = *0*,

the found instances may not overlap at all, while for MaxOverlap = *1* no check for overlap is performed, and hence all instances are returned. The calculation of the overlap is based on the smallest enclosing rectangles of arbitrary orientation (see smallest_rectangle2) of the found component instances.

IfRootNotFound: specifies the behavior of the operator when dealing with a missing or strongly occluded root component.

Possible values:

- *'stop_search'*: It is assumed that the root component is always found in the image. Consequently, for instances for which the root component could not be found the search for the remaining components is not continued.
- *'select_new_root'*: Different components are successively chosen as the root component and searched within the full search space. The order in which the selection of the root component is performed corresponds to the order passed in RootRanking. The poses of the found instances of all root components are then used to start the recursive search for the remaining components. Hence, it is possible to find instances even if the original root component is not found. However, the computation time of the search increases significantly in comparison to the search when choosing *'stop_search'*. This is especially the case for small values of MinScore, as more root components must be searched. If during the search more root components are needed as given in RootComponent, the root components are completed by the automatically computed order (see create_trained_component_model or create_component_model).

IfComponentNotFound: Specifies how components are searched when the predecessor component was not found (e.g., because she is missing or strongly occluded).

Possible values:

- *'prune_branch'*: Such components are not searched at all and are also treated as 'not found'.
- *'search_from_upper'*: Such components are searched relative to the pose of the predecessor component of the predecessor component.
- *'search_from_best'*: Such components are searched relative to the pose of the already found component from which the relative search can be performed with minimum computational effort.

PosePrediction: Determines whether the pose of not found components should be estimated.

Possible values:

- *'none'*: Only the poses of the found components are returned.
- *'from_neighbors'*: The poses of not found components are estimated and returned with a score of ScoreComp = *0.0*. The pose estimation is based on the poses of the found neighboring components in the search tree.
- *'from_all'*: The poses of not found components are estimated and returned with a score of ScoreComp = *0.0*. The pose estimation is based on the poses of all found components.

MinScoreComp: Minimal necessary score of the components for the instances to be found.

This parameter has the same meaning as MinScore in find_shape_model.

Settable is either one element or the same number of elements as model components in ComponentModelID. In the first case the parameter is used for all components. In the second case, each parameter element refers to the corresponding component in ComponentModelID.

SubPixelComp: Determines whether the extraction shall be done subpixel precise and in the given case the maximal allowed object deformation in pixels.

This parameter has the same meaning as SubPixel in find_shape_model. Therefore the maximal allowed object deformation has to be given as integer in the same string. Settable is either one element or the same number of elements as model components in ComponentModelID. In the first case the parameter is used for all components. In the second case, each parameter element refers to the corresponding component in ComponentModelID.

Example: ['least_squares', 'max_deformation 2'].

NumLevelsComp: Determine the pyramid levels for the components used in the matching.

This parameter has the same meaning as NumLevels in find_shape_model.

It determines the number of pyramid levels for the components to be used in the matching. Settable is either one element or the same number of elements as model components in ComponentModelID. In the first case the parameter is used for all components. In the second case, each parameter element refers to the corresponding component in ComponentModelID.

Optional, one can set value pairs for this parameter: In this case, the first value still determines the number of pyramid levels to be used. The second value specifies the lowest pyramid level, to which the found matches are tracked. In doing so, one can set either a single value pair or a value pair for each model component in ComponentModelID. If different value pairs should be used for different components, they must be specified in the same tuple. In case ComponentModelID contains exactly two components and in NumLevelsComp two values are set, these values are interpreted as different number of pyramid levels to be used and not as a value pair.

Example: ComponentModelID contains two components, for which different pyramid levels shall be considered. For the first component 5 levels up to the level 2 shall be used. For the second component 4 levels up to the level 1 shall be used. In this case is NumLevelsComp = *[5,2,4,1]*.

GreedinessComp: "Greediness" of the search heuristic for the components: value from 0 to 1. Thereby 0 means: safe but slow, 1: fast but matches may be missed.

This parameter has the same meaning as Greediness in find_shape_model.

Settable is either one element or the same number of elements as model components in ComponentModelID. In the first case the parameter is used for all components. In the second case, each parameter element refers to the corresponding component in ComponentModelID.

Further information about output parameters

ModelStart and ModelEnd: Return the first and last index and therewith the index range of all component matches associated to the same instance of the component model.

The component matches corresponding to the first found instance of the component model are given by the interval of indices [ModelStart[0],ModelEnd[0]]. Thereby the indices refer to the values of the parameters RowComp, ColumnComp, AngleComp, ScoreComp, and ModelComp.

Example: The component model consists for three components. Two instances have been found on the image, where for one instance only two components (component 0 and component 2) could be found. Then the returned parameters could look like this:

RowComp = *[100,200,300,150,250]*	ModelStart = *[0,3]*
ColumnComp = *[200,210,220,400,425]*	ModelEnd = *[2,4]*
AngleComp = *[0,0.1,-0.2,0.1,0.2]*	ModelComp = *[0,1,2,0,2]*
ScoreComp = *[1,1,1,1,1]*	Score = *[1,1]*

From the right column it is visible, that in the left column:

- Values with index 0 to 2 correspond to the components 0 to 2 of instance 1.
- Values with index 3 to 4 correspond to the components 0 and 2 of instance 2.

Score: Score of the found instances of the component model.

Score contains the weighted mean of the component scores, the values in ScoreComp. The weighting is performed according to the number of model points within the respective component.

RowComp, ColumnComp, and AngleComp: The position (RowComp, ColumnComp) and rotation (AngleComp) of the model components of all found component model instances.

The coordinates RowComp and ColumnComp are the coordinates of the component origin (reference point) in the search image. The component origin depends on the model creation:

- with create_trained_component_model by training: The component origin is the center of gravity of the respective returned contour region in ModelComponents of the operator train_model_components.
- with create_component_model manually: The component origin is the center of gravity of the corresponding passed component region ComponentRegions of the operator create_component_model.

Since the relations between the components in ComponentModelID refer to this reference point, the origin of the components must not be modified by using set_shape_model_origin.

ScoreComp: Score of each found component instance.

The score is a number between 0 and 1, and is an approximate measure of how much of the component is visible in the image. If, for example, half of the component is occluded, the score cannot exceed 0.5.

ModelComp: Index of the found component.

The tuple contains the indices of the respective model components (see create_component_model and train_model_components, respectively). By this the values in RowComp, ColumnComp, AngleComp, and ScoreComp can be associated to the different model components. See also the example given for ModelStart and ModelEnd.

Information concerning the search

Internally, the shape-based matching is used for the component-based matching in order to search the individual components (see find_shape_model).

The domain of the Image determines the search space for the reference point, i.e., the allowed positions, of the root component.

Usually the component model is searched only within those points of the image domain in which the model fits completely into the image. This means that the components will not be found if they extend beyond the borders of the image, even if they would achieve a score greater than MinScoreComp (see above).

Note that, if for a certain pyramid level the component model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed with set_system ('border_shape_models','true'), which will cause components that extend beyond the image border to be found if they achieve a score greater than MinScoreComp. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that this mode increases the runtime of the search.

Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels the model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., dilation_circle.

When tracking the matches through the image pyramid, on each level, some less promising matches are rejected based on NumMatches. Thus, it is possible that some matches are rejected that would have had a higher score on the lowest pyramid level. Due to this, for example, the found match for NumMatches set to *1* might be different from the match with the highest score returned when setting NumMatches to *0* or *> 1*.

Recommendations

If multiple objects with a similar score are expected, but only the one with the highest score should be returned, it might be preferable to raise NumMatches, and then select the match with the highest score.

To get a meaningful score value and to avoid erroneous matches, we recommend to always combine the allowance of a deformation with a subpixel extraction that applies a least-squares adjustment. If the subpixel extraction and/or the maximum object deformation is specified separately for each component, for each component in ComponentModelID exactly one value for the subpixel extraction must be passed in SubPixelComp. After each value for the subpixel extraction optionally a second value can be passed, which describes the maximum object deformation of the corresponding mode. If for a certain component no value for the maximum object deformation is passed, the component is searched without taking deformations into account. Further details can be found in the documentation of find_shape_models.

Parameters

- ▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
  Input image in which the component model should be found.
- ▷ **ComponentModelID** (input_control) . . . . . component_model ⇝ *handle*
  Handle of the component model.
- ▷ **RootComponent** (input_control) . . . . . integer(-array) ⇝ *integer*
  Index of the root component.
  Suggested values: RootComponent ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8}
- ▷ **AngleStartRoot** (input_control) . . . . . angle.rad(-array) ⇝ *real*
  Smallest rotation of the root component
  Default: -0.39
  Suggested values: AngleStartRoot ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}
- ▷ **AngleExtentRoot** (input_control) . . . . . angle.rad(-array) ⇝ *real*
  Extent of the rotation of the root component.
  Default: 0.79
  Suggested values: AngleExtentRoot ∈ {6.28, 3.14, 1.57, 0.79, 0.39, 0.0}
  Restriction: AngleExtentRoot >= 0

▷ **MinScore** (input_control) . . . . . real ⇝ *real*
Minimum score of the instances of the component model to be found.
Default: 0.5
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= MinScore && MinScore <= 1

▷ **NumMatches** (input_control) . . . . . integer ⇝ *integer*
Number of instances of the component model to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real ⇝ *real*
Maximum overlap of the instances of the component models to be found.
Default: 0.5
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= MaxOverlap && MaxOverlap <= 1

▷ **IfRootNotFound** (input_control) . . . . . string ⇝ *string*
Behavior if the root component is missing.
Default: 'stop_search'
List of values: IfRootNotFound ∈ {'stop_search', 'select_new_root'}

▷ **IfComponentNotFound** (input_control) . . . . . string ⇝ *string*
Behavior if a component is missing.
Default: 'prune_branch'
List of values: IfComponentNotFound ∈ {'prune_branch', 'search_from_upper', 'search_from_best'}

▷ **PosePrediction** (input_control) . . . . . string ⇝ *string*
Pose prediction of components that are not found.
Default: 'none'
List of values: PosePrediction ∈ {'none', 'from_neighbors', 'from_all'}

▷ **MinScoreComp** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the components to be found.
Default: 0.5
Suggested values: MinScoreComp ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= MinScoreComp && MinScoreComp <= 1

▷ **SubPixelComp** (input_control) . . . . . string(-array) ⇝ *string*
Subpixel accuracy of the component poses if not equal to *'none'*.
Default: 'least_squares'
Suggested values: SubPixelComp ∈ {'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high', 'max_deformation 1', 'max_deformation 2', 'max_deformation 3', 'max_deformation 4', 'max_deformation 5', 'max_deformation 6'}

▷ **NumLevelsComp** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of pyramid levels for the components used in the matching (and lowest pyramid level to use if $|\texttt{NumLevelsComp}| = 2n$).
Default: 0
List of values: NumLevelsComp ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **GreedinessComp** (input_control) . . . . . real(-array) ⇝ *real*
"Greediness" of the search heuristic for the components (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: GreedinessComp ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= GreedinessComp && GreedinessComp <= 1

▷ **ModelStart** (output_control) . . . . . integer(-array) ⇝ *integer*
Start index of each found instance of the component model in the tuples describing the component matches.

▷ **ModelEnd** (output_control) ..... integer(-array) ⇝ *integer*
End index of each found instance of the component model in the tuples describing the component matches.

▷ **Score** (output_control) ..... real(-array) ⇝ *real*
Score of the found instances of the component model.

▷ **RowComp** (output_control) ..... point.y(-array) ⇝ *real*
Row coordinate of the found component matches.

▷ **ColumnComp** (output_control) ..... point.x(-array) ⇝ *real*
Column coordinate of the found component matches.

▷ **AngleComp** (output_control) ..... angle.rad(-array) ⇝ *real*
Rotation angle of the found component matches.

▷ **ScoreComp** (output_control) ..... real(-array) ⇝ *real*
Score of the found component matches.

▷ **ModelComp** (output_control) ..... integer(-array) ⇝ *integer*
Index of the found components.

Result

If the parameter values are correct, the operator `find_component_model` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input image available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

```
gen_initial_components ( 
    ModelImage : InitialComponents : ContrastLow, ContrastHigh,
    MinSize, Mode, GenericName, GenericValue : )
```

Extract the initial components of a component model.

`gen_initial_components` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

In general, there are two possibilities to use `gen_initial_components`. The first possibility should be chosen if the components of the component model are not known. Then `gen_initial_components` automatically extracts the initial components of a component model from a model image. The second possibility can be chosen if the components of the component model are approximately known. Then `gen_initial_components` can be used to find suitable parameter values for the model feature extraction in `train_model_components` and `create_component_model`. Hence, the second possibility is comparable to the function of `inspect_shape_model` within the shape-based matching.

When using the first possibility, `gen_initial_components` extracts the initial components of a component model from a model image `ModelImage`. As already mentioned, this is especially useful if the components of the component model are not known. In this case, the resulting initial components can be used to automatically train the component model with `train_model_components`, which extracts the (final) model components and the relations between them. `gen_initial_components` returns the initial components in a region object tuple `InitialComponents` that contains a representation for each initial component in form of contour regions.

For the automatic determination of the initial components, the domain of the model image `ModelImage` must contain the entire compound object including all components. `Mode` specifies the method used for the automatic computation. Currently, only the mode *'connection'* is available. In this mode the automatic computation is performed in two steps: In the first step, features are extracted using the parameters `ContrastLow`, `ContrastHigh`, and `MinSize`. These three parameters define the contours of which the initial components should consist and should be chosen such that only the significant features of the model image are contained in the initial components. `ContrastLow` and `ContrastHigh` specify the gray value contrast of the points that should

be contained in the initial components. The contrast is a measure for local gray value differences between the object and the background and between different parts of the object. The model image is segmented using a method similar to the hysteresis threshold method used in edges_image. Here, ContrastLow determines the lower threshold, while ContrastHigh determines the upper threshold. If the same value is passed for ContrastLow and ContrastHigh a simple thresholding operation is performed. For more information about the hysteresis threshold method, see hysteresis_threshold. MinSize can be used to select only significant features for the initial components based on the size of the connected contour regions, i.e., connected contour regions with fewer than MinSize points are suppressed.

The resulting connected contour regions are iteratively merged in the second step. For this, two contour regions are merged if the distance between both regions is smaller than a certain threshold (see below). Finally, the merged regions are returned in InitialComponents and can be used to train the component model by passing them to train_model_components.

To control the internal image processing, the parameters GenericName and GenericValue are used. This is done by passing the names of the control parameters to be changed in GenericName as a list of strings. In GenericValue the values are passed at the corresponding index positions.

Normally, none of the values needs to be changed. A change should only be applied in case of unsatisfying results of the automatic determination of the initial components. The two parameters that can be changed are *'merge_distance'* and *'merge_fraction'*; both are used during the iterative merging of contour regions (see above). First, the fraction of contour pixels of one contour region that at most have a distance of *'merge_distance'* from another contour region is computed. If this fraction exceeds the value that is passed in *'merge_fraction'* the two contour regions are merged. Consequently, the higher *'merge_distance'* and the lower *'merge_fraction'* is chosen the more contour regions are merged. The default value of *'merge_distance'* is *5* and the default value of *'merge_fraction'* is *0.5* (corresponds to 50 percent).

When using the second possibility, i.e., the components of the component model are approximately known, the training by using train_model_components can be performed without previously executing gen_initial_components. If this is desired, the initial components can be specified by the user and directly passed to train_model_components. Furthermore, if the components as well as the relative movements (relations) of the components are known, gen_initial_components as well as train_model_components need not be executed. In fact, by immediately passing the components as well as the relations to create_component_model, the component model can be created without any training. In both cases, however, gen_initial_components can be used to evaluate the effect of the feature extraction parameters ContrastLow, ContrastHigh, and MinSize of train_model_components and create_component_model, and hence to find suitable parameter values for a certain application.

For this, the image regions for the (initial) components must be explicitly given, i.e., for each (initial) component a separate image from which the (initial) component should be created is passed. In this case, ModelImage contains multiple image objects. The domain of each image object is used as the region of interest for calculating the corresponding (initial) component. The image matrix of all image objects in the tuple must be identical, i.e., ModelImage cannot be constructed in an arbitrary manner using concat_obj, but must be created from the same image using add_channels or equivalent calls. If this is not the case, an error message is returned. If the parameters ContrastLow, ContrastHigh, or MinSize only contain one element, this value is applied to the creation of all (initial) components. In contrast, if different values for different (initial) components should be used, tuples of values can be passed for these three parameters. In this case, the tuples must have a length that corresponds to the number of (initial) components, i.e., the number of image objects in ModelImage. The contour regions of the (initial) components are returned in InitialComponents.

Thus, the second possibility is equivalent to the function of inspect_shape_model within the shape-based matching. However, in contrast to inspect_shape_model, gen_initial_components does not return the contour regions on multiple image pyramid levels. Therefore, if the number of pyramid levels to be used should be chosen manually, preferably inspect_shape_model should be called individually for each (initial) component.

For both described possibilities the parameters ContrastLow, ContrastHigh, and MinSize can be automatically determined. If both hysteresis threshold should be automatically determined, both ContrastLow and ContrastHigh must be set to *'auto'*. In contrast, if only one threshold value should be determined, ContrastLow must be set to *'auto'* while ContrastHigh must be set to an arbitrary value different from *'auto'*.

If the input image ModelImage has one channel the representation of the model is created with the method that is used in create_component_model or create_trained_component_model for the metrics

'use_polarity', *'ignore_global_polarity'*, and *'ignore_local_polarity'*. If the input image has more than one channel the representation is created with the method that is used for the metric *'ignore_color_polarity'*.

Parameters

▷ **ModelImage** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object* : byte / uint2
Input image from which the initial components should be extracted.

▷ **InitialComponents** (output_object) . . . . . region-array ⇝ *object*
Contour regions of initial components.

▷ **ContrastLow** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Lower hysteresis threshold for the contrast of the initial components in the image.
Default: 'auto'
Suggested values: ContrastLow ∈ {'auto', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}
Restriction: ContrastLow > 0

▷ **ContrastHigh** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Upper hysteresis threshold for the contrast of the initial components in the image.
Default: 'auto'
Suggested values: ContrastHigh ∈ {'auto', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}
Restriction: ContrastHigh > 0 && ContrastHigh >= ContrastLow

▷ **MinSize** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Minimum size of the initial components.
Default: 'auto'
Suggested values: MinSize ∈ {'auto', 0, 5, 10, 20, 30, 40}
Restriction: MinSize >= 0

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Type of automatic segmentation.
Default: 'connection'
List of values: Mode ∈ {'connection'}

▷ **GenericName** (input_control) . . . . . string(-array) ⇝ *string*
Names of optional control parameters.
Default: []
List of values: GenericName ∈ {'merge_distance', 'merge_fraction'}

▷ **GenericValue** (input_control) . . . . . number(-array) ⇝ *real* / integer
Values of optional control parameters.
Default: []

Result

If the parameter values are correct, the operator gen_initial_components returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

inspect_shape_model

Module

Matching

get_component_model_params (: : ComponentModelID : MinScoreComp, RootRanking, ShapeModelIDs)

Return the parameters of a component model.

get_component_model_params is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

The operator get_component_model_params returns the parameters of the component model ComponentModelID. In particular, this output can be used to check the parameters RootRanking and MinScoreComp after reading the component model with read_component_model. Additionally, the operator returns the shape model IDs ShapeModelIDs of the model components. The order of the returned shape model IDs corresponds to the indices of the components within the component model ComponentModelID. The IDs can be used to query their shape model parameters with get_shape_model_params.

Parameters

▷ **ComponentModelID** (input_control) ..... component_model ⇝ *handle*
Handle of the component model.

▷ **MinScoreComp** (output_control) ..... real(-array) ⇝ *real*
Minimum score of the instances of the components to be found.

▷ **RootRanking** (output_control) ..... integer(-array) ⇝ *integer*
Ranking of the model components expressing their suitability to act as root component.

▷ **ShapeModelIDs** (output_control) ..... shape_model(-array) ⇝ *handle*
Handles of the shape models of the individual model components.

Result

If the handle of the component model is valid, the operator get_component_model_params returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

Matching

get_component_model_tree (: Tree, Relations : ComponentModelID, RootComponent, Image : StartNode, EndNode, Row, Column, Phi, Length1, Length2, AngleStart, AngleExtent)

Return the search tree of a component model.

get_component_model_tree is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

get_component_model_tree returns the search tree Tree and the associated relations Relations of the component model that is passed in ComponentModelID in form of regions as well as in numerical form. get_component_model_tree is particularly useful in order to visualize the search order of the components, which was automatically computed in create_trained_component_model or create_component_model.

Because the search tree depends on the selected root component, the root component must be passed in RootComponent. The nodes in the tree Tree represent the model components, the connecting lines between the nodes indicate which components are searched relative to each other. The position of the nodes corresponds to the position of the components in the model image (if Image = *'model_image'* or Image = *0*) or in a training image (if Image ≥ *1*). In the latter case, the component model must have been created based on a component training result with create_trained_component_model.

Let *n* be the number of components in ComponentModelID. The region object tuple Relations of length *n* is designed as follows: For each component a separate region is returned. The positions of all components in the image are represented by circles with a radius of 3 pixels. For each component other than the root component RootComponent, additionally the position relation and the orientation relation relative to the predecessor component in the search tree are represented. The position relation is represented by a rectangle, the orientation relation is represented by a circle sector with a radius of 30 pixels. The center of the circle is placed at the mean

relative position of the component. The rectangle describes the movement of the reference point of the respective component relative to the pose of its predecessor component, the circle sector describes the variation of the relative orientation. A relative orientation of 0 corresponds to the relative orientation of both components in the model image.

In addition to the regions, the search tree as well as the associated relations are also returned in numerical form. The search tree is described by the two tuples `StartNode` and `EndNode`, both of length *n*, which contain the start and the end node of all arcs in the tree. The nodes contain the indices of the components. This means that during the search the component that is described by the end node is searched relative to the pose of the component that is described by the start node (predecessor component). Since the root component is not searched relative to any other component, and hence does not have a predecessor component, the associated start node is set to *-1*. The relations are returned in `Row`, `Column`, `Phi`, `Length1`, `Length2`, `AngleStart`, and `AngleExtent`. These parameters are tuples of length *n*, and contain the relations of all components relative to their associated predecessor component, where the order of the values within the tuples is determined by the index of the corresponding component. The position relation is described by the parameters of the corresponding rectangle `Row`, `Column`, `Phi`, `Length1`, and `Length2` (see `gen_rectangle2`). The orientation relation is described by the starting angle `AngleStart` and the angle extent `AngleExtent`.

For the root component as well as for components that do not have a predecessor in the current image or that have not been found in the current image, an empty region is returned and the corresponding values of the seven parameters are set to *0*.

Parameters

- ▷ **`Tree`** (output_object) . . . . . region ⇝ *object*
  Search tree.
- ▷ **`Relations`** (output_object) . . . . . region-array ⇝ *object*
  Relations of components that are connected in the search tree.
- ▷ **`ComponentModelID`** (input_control) . . . . . component_model ⇝ *handle*
  Handle of the component model.
- ▷ **`RootComponent`** (input_control) . . . . . integer(-array) ⇝ *integer*
  Index of the root component.
  Suggested values: `RootComponent` ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8}
- ▷ **`Image`** (input_control) . . . . . string ⇝ *string* / integer
  Image for which the tree is to be returned.
  Default: 'model_image'
  Suggested values: `Image` ∈ {'model_image', 0, 1, 2, 3, 4, 5, 6, 7, 8}
- ▷ **`StartNode`** (output_control) . . . . . integer(-array) ⇝ *integer*
  Component index of the start node of an arc in the search tree.
- ▷ **`EndNode`** (output_control) . . . . . integer(-array) ⇝ *integer*
  Component index of the end node of an arc in the search tree.
- ▷ **`Row`** (output_control) . . . . . rectangle2.center.y(-array) ⇝ *real*
  Row coordinate of the center of the rectangle representing the relation.
- ▷ **`Column`** (output_control) . . . . . rectangle2.center.x(-array) ⇝ *real*
  Column index of the center of the rectangle representing the relation.
- ▷ **`Phi`** (output_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real*
  Orientation of the rectangle representing the relation (radians).
  Assertion: - pi / 2 < Phi && Phi <= pi / 2
- ▷ **`Length1`** (output_control) . . . . . rectangle2.hwidth(-array) ⇝ *real*
  First radius (half length) of the rectangle representing the relation.
  Assertion: Length1 >= 0.0
- ▷ **`Length2`** (output_control) . . . . . rectangle2.hheight(-array) ⇝ *real*
  Second radius (half width) of the rectangle representing the relation.
  Assertion: Length2 >= 0.0 && Length2 <= Length1
- ▷ **`AngleStart`** (output_control) . . . . . angle.rad(-array) ⇝ *real*
  Smallest relative orientation angle.
- ▷ **`AngleExtent`** (output_control) . . . . . angle.rad(-array) ⇝ *real*
  Extent of the relative orientation angle.

Result

If the parameters are valid, the operator `get_component_model_tree` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

get_component_relations (: Relations : ComponentTrainingID, ReferenceComponent, Image : Row, Column, Phi, Length1, Length2, AngleStart, AngleExtent)

Return the relations between the model components that are contained in a training result.

`get_component_relations` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

`get_component_relations` returns the relations between model components after training them with `train_model_components`. With the parameter `ReferenceComponent`, you can select a reference component. `get_component_relations` then returns the relations between the reference component and all other components in the model image (if `Image` = *'model_image'* or `Image` = *0*) or in a training image (if `Image` ≥ *1*). In order to obtain the relations in the *i*th training image, `Image` must be set to *i*. The result of the training returned by `train_model_components` must be passed in `ComponentTrainingID`. `ReferenceComponent` describes the index of the reference component and must be within the range of 0 and *n*-1, if *n* is the number of model components (see `train_model_components`).

The relations are returned in form of regions in `Relations` as well as in form of numerical values in `Row`, `Column`, `Phi`, `Length1`, `Length2`, `AngleStart`, and `AngleExtent`.

The region object tuple `Relations` is designed as follows. For each component a separate region is returned. Consequently, `Relations` contains *n* regions, where the order of the regions within the tuple is determined by the index of the corresponding components. The positions of all components in the image are represented by circles with a radius of 3 pixels. For each component other than the reference component `ReferenceComponent`, additionally the position relation and the orientation relation relative to the reference component are represented. The position relation is represented by a rectangle and the orientation relation is represented by a circle sector with a radius of 30 pixels. The center of the circle is placed at the mean relative position of the component. The rectangle describes the movement of the reference point of the respective component relative to the pose of the reference component, while the circle sector describes the variation of the relative orientation (see `train_model_components`). A relative orientation of 0 corresponds to the relative orientation of both components in the model image. If both components appear in the same relative orientation in all images, the circle sector consequently degenerates to a straight line.

In addition to the region object tuple `Relations`, the relations are also returned in form of numerical values in `Row`, `Column`, `Phi`, `Length1`, `Length2`, `AngleStart`, and `AngleExtent`. These parameters are tuples of length *n* and contain the relations of all components relative to the reference component, where the order of the values within the tuples is determined by the index of the corresponding component. The position relation is described by the parameters of the corresponding rectangle `Row`, `Column`, `Phi`, `Length1`, and `Length2` (see `gen_rectangle2`). The orientation relation is described by the starting angle `AngleStart` and the angle extent `AngleExtent`. For the reference component only the position within the image is returned in `Row` and `Column`. All other values are set to *0*.

If the reference component has not been found in the current image, an array of empty regions is returned and the corresponding parameter values are set to *0*.

The operator `get_component_relations` is particularly useful in order to visualize the result of the training that was performed with `train_model_components`. With this, it is possible to evaluate the variations that are contained in the training images. Sometimes it might be reasonable to restart the training with `train_model_components` while using a different set of training images.

Parameters

▷ **Relations** (output_object) . . . . . region(-array) ⇝ *object*
Region representation of the relations.

▷ **ComponentTrainingID** (input_control) . . . . . component_training ⇝ *handle*
Handle of the training result.

▷ **ReferenceComponent** (input_control) . . . . . integer ⇝ *integer*
Index of reference component.
Restriction: ReferenceComponent >= 0

▷ **Image** (input_control) . . . . . string ⇝ *string* / integer
Image for which the component relations are to be returned.
Default: 'model_image'
Suggested values: Image ∈ {'model_image', 0, 1, 2, 3, 4, 5, 6, 7, 8}

▷ **Row** (output_control) . . . . . rectangle2.center.y(-array) ⇝ *real*
Row coordinate of the center of the rectangle representing the relation.

▷ **Column** (output_control) . . . . . rectangle2.center.x(-array) ⇝ *real*
Column index of the center of the rectangle representing the relation.

▷ **Phi** (output_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real*
Orientation of the rectangle representing the relation (radians).
Assertion: - pi / 2 < Phi && Phi <= pi / 2

▷ **Length1** (output_control) . . . . . rectangle2.hwidth(-array) ⇝ *real*
First radius (half length) of the rectangle representing the relation.
Assertion: Length1 >= 0.0

▷ **Length2** (output_control) . . . . . rectangle2.hheight(-array) ⇝ *real*
Second radius (half width) of the rectangle representing the relation.
Assertion: Length2 >= 0.0 && Length2 <= Length1

▷ **AngleStart** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Smallest relative orientation angle.

▷ **AngleExtent** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Extent of the relative orientation angles.

Result

If the handle of the training result is valid, the operator `get_component_relations` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`gen_rectangle2`

Module

Matching

get_found_component_model (: FoundComponents : ComponentModelID, ModelStart, ModelEnd, RowComp, ColumnComp, AngleComp, ScoreComp, ModelComp, ModelMatch, MarkOrientation : RowCompInst, ColumnCompInst, AngleCompInst, ScoreCompInst)

Return the components of a found instance of a component model.

get_found_component_model is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

`get_found_component_model` returns the components of a found instance of the component model `ComponentModelID` in form of contour regions in `FoundComponents` as well as in numerical form.

The operator get_found_component_model is particularly useful in order to visualize the matches that have been obtained by find_component_model.

The pose of the returned components corresponds to their pose in the search image as returned by find_component_model. Hence, the parameters ModelStart, ModelEnd, RowComp, ColumnComp, AngleComp, ScoreComp, and ModelComp must be passed to get_found_component_model as they have been returned by find_component_model. In ModelMatch the index of the found instance of the component model must be passed. Consequently, ModelMatch must lie within the range between 0 and *m*-1, where *m* is the number of elements in ModelStart and ModelEnd, and hence corresponds to the number of found model instances. For example, if the best match should be returned, ModelMatch should be set to *0*.

When dealing with rotationally symmetric components, one may wish to mark the current orientation of the found component. This can be achieved by setting MarkOrientation to *'true'*. In this case, the contour region of each component is complemented by an arrow at its reference point that points in the reference direction. The reference direction of a component is based on the orientation of the component in the model image (see train_model_components or create_component_model) and is represented by an arrow that starts at the reference point and points to the right in the horizontal direction.

For convenience, the pose parameters as well as the score of each component of the found model instance are additionally returned in numerical form in RowCompInst, ColumnCompInst, AngleCompInst, and ScoreCompInst. The four tuples are always of length *n*, where *n* is the number of components in the component model ComponentModelID. If a component could not be found during the search, an empty region is passed in the corresponding element of FoundComponents and the value of the corresponding element in RowCompInst, ColumnCompInst, AngleCompInst, and ScoreCompInst is set to *0*.

Parameters

▷ **FoundComponents** (output_object) ..... region-array ⇝ *object*
Found components of the selected component model instance.

▷ **ComponentModelID** (input_control) ..... component_model ⇝ *handle*
Handle of the component model.

▷ **ModelStart** (input_control) ..... integer(-array) ⇝ *integer*
Start index of each found instance of the component model in the tuples describing the component matches.

▷ **ModelEnd** (input_control) ..... integer(-array) ⇝ *integer*
End index of each found instance of the component model to the tuples describing the component matches.

▷ **RowComp** (input_control) ..... point.y(-array) ⇝ *real*
Row coordinate of the found component matches.

▷ **ColumnComp** (input_control) ..... point.x(-array) ⇝ *real*
Column coordinate of the found component matches.

▷ **AngleComp** (input_control) ..... angle.rad(-array) ⇝ *real*
Rotation angle of the found component matches.

▷ **ScoreComp** (input_control) ..... real(-array) ⇝ *real*
Score of the found component matches.

▷ **ModelComp** (input_control) ..... integer(-array) ⇝ *integer*
Index of the found components.

▷ **ModelMatch** (input_control) ..... integer ⇝ *integer*
Index of the found instance of the component model to be returned.

▷ **MarkOrientation** (input_control) ..... string ⇝ *string*
Mark the orientation of the components.
Default: 'false'
List of values: MarkOrientation ∈ {'true', 'false'}

▷ **RowCompInst** (output_control) ..... point.y(-array) ⇝ *real*
Row coordinate of all components of the selected model instance.

▷ **ColumnCompInst** (output_control) ..... point.x(-array) ⇝ *real*
Column coordinate of all components of the selected model instance.

▷ **AngleCompInst** (output_control) ..... angle.rad(-array) ⇝ *real*
Rotation angle of all components of the selected model instance.

▷ **ScoreCompInst** (output_control) ..... real(-array) ⇝ *real*
Score of all components of the selected model instance.

Result

If the parameters are valid, the operator `get_found_component_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

get_training_components (
: TrainingComponents : ComponentTrainingID, Components, Image,
MarkOrientation : Row, Column, Angle, Score)

Return the initial or model components in a certain image.

`get_training_components` is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

`get_training_components` returns all initial components (if `Components` = *'initial_components'*) or all model components (if `Components` = *'model_components'*) in `TrainingComponents` in form of contour regions as well as in numerical form. Alternatively, by directly passing the index of an initial component, all found poses of that initial component (i.e., the poses before solving the ambiguities in `train_model_components`) are returned.

The pose of the returned components corresponds to their pose in the model image (if `Image` = *'model_image'* or `Image` = *0*) or in a training image (if `Image` $\geq$ *1*). In order to obtain the components in the pose at which they were found in the *i*th training image, `Image` must be set to *i*. Furthermore, when dealing with rotationally symmetric components, one may wish to mark the current orientation of the found component. This can be achieved by setting `MarkOrientation` to *'true'*. In this case, the contour region of each component is complemented by an arrow at its reference point pointing in the reference direction. The reference direction of a component is based on the orientation of the component in the model image and is represented by an arrow that starts at the reference point and points to the right in the horizontal direction.

In addition to the contour regions, the pose and the score of all found components is returned in `Row`, `Column`, `Angle`, and `Score` (see `find_shape_model`). If `Components` was set to *'initial_components'* or *'model_components'*, the tuples `TrainingComponents`, `Row`, `Column`, `Angle`, and `Score` always contain the same number of elements as initial components or model components contained in `ComponentTrainingID`, respectively. If one component was not found in the image, an empty region is returned in the corresponding element of `TrainingComponents` and the elements of the four output control parameters are set to the value *0*. In contrast, if the index of an initial component is passed in `Components`, these tuples contain as many elements as matches of the corresponding initial component were found in the image.

The operator `get_training_components` is particularly useful in order to visualize the result of the training `ComponentTrainingID`, which was performed with `train_model_components`. With this, it is possible to evaluate the suitability of the training images or to inspect the influence of the parameters of `train_model_components`. Sometimes it might be reasonable to restart the training with `train_model_components` using a different set of training images or after adjusting the parameters.

Parameters

▷ **`TrainingComponents`** (output_object) ..... region(-array) ⇝ *object*
Contour regions of the initial components or of the model components.

▷ **`ComponentTrainingID`** (input_control) ..... component_training ⇝ *handle*
Handle of the training result.

▷ **`Components`** (input_control) ..... string ⇝ *string* / integer
Type of returned components or index of an initial component.
Default: 'model_components'
Suggested values: `Components` ∈ {'model_components', 'initial_components', 0, 1, 2, 3, 4, 5}

▷ **Image** (input_control) . . . . . string ⇝ *string* / integer
Image for which the components are to be returned.
Default: 'model_image'
Suggested values: Image ∈ {'model_image', 0, 1, 2, 3, 4, 5, 6, 7, 8}

▷ **MarkOrientation** (input_control) . . . . . string ⇝ *string*
Mark the orientation of the components.
Default: 'false'
List of values: MarkOrientation ∈ {'true', 'false'}

▷ **Row** (output_control) . . . . . point.y(-array) ⇝ *real*
Row coordinate of the found instances of all initial components or model components.

▷ **Column** (output_control) . . . . . point.x(-array) ⇝ *real*
Column coordinate of the found instances of all initial components or model components.

▷ **Angle** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Rotation angle of the found instances of all components.

▷ **Score** (output_control) . . . . . real(-array) ⇝ *real*
Score of the found instances of all components.

Result

If the handle of the training result is valid, the operator get_training_components returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

inspect_clustered_components (
: ModelComponents : ComponentTrainingID, AmbiguityCriterion,
MaxContourOverlap, ClusterThreshold :)

Inspect the rigid model components obtained from the training.

inspect_clustered_components is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

inspect_clustered_components creates a representation of the rigid model components based on the training result ComponentTrainingID in form of contour regions. The resulting rigid model components are computed depending on the criterion that is used to solve the ambiguities AmbiguityCriterion, the maximum allowable contour overlap MaxContourOverlap, and the cluster threshold ClusterThreshold (see train_model_components). The cluster threshold, for example, influences the merging of the initial components. The greater the threshold is chosen, the fewer initial components are merged. The determined rigid model components are returned in ModelComponents.

Hence, after the components have been trained once by using train_model_components, inspect_clustered_components can be used to estimate the effect of different values for the parameters AmbiguityCriterion, MaxContourOverlap, and ClusterThreshold without performing the complete training procedure several times. Once the desired parameter values have been found, they can be efficiently adopted into the training result by using cluster_model_components.

Parameters

▷ **ModelComponents** (output_object) . . . . . region(-array) ⇝ *object*
Contour regions of rigid model components.

▷ **ComponentTrainingID** (input_control) . . . . . component_training ⇝ *handle*
Handle of the training result.

▷ **AmbiguityCriterion** (input_control) ..... string ⇝ *string*
Criterion for solving the ambiguities.
Default: 'rigidity'
List of values: AmbiguityCriterion ∈ {'distance', 'orientation', 'distance_orientation', 'rigidity'}

▷ **MaxContourOverlap** (input_control) ..... real ⇝ *real*
Maximum contour overlap of the found initial components.
Default: 0.2
Suggested values: MaxContourOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= MaxContourOverlap && MaxContourOverlap <= 1

▷ **ClusterThreshold** (input_control) ..... real ⇝ *real*
Threshold for clustering the initial components.
Default: 0.5
Suggested values: ClusterThreshold ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: 0 <= ClusterThreshold && ClusterThreshold <= 1

Result

If the handle of the training result is valid, the operator inspect_clustered_components returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

modify_component_relations (: : ComponentTrainingID, ReferenceComponent, ToleranceComponent, PositionTolerance, AngleTolerance :)

Modify the relations within a training result.

modify_component_relations is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

modify_component_relations modifies the relations between the model components within the component training result ComponentTrainingID. The selection of the relation(s) that should be changed is performed by setting ReferenceComponent and ToleranceComponent, respectively. This means that the relative movement of component ToleranceComponent with respect to the component ReferenceComponent is modified.

The size of the change is specified as follows: By specifying a position tolerance PositionTolerance, the semi-axes of the rectangle that describes the reference point's movement (see train_model_components) are enlarged by PositionTolerance pixels. Accordingly, by specifying an orientation tolerance AngleTolerance, the angle range that describes the variation of the relative orientation (see train_model_components) is enlarged by AngleTolerance to both sides. Consequently, negative tolerance values lead to a decreased size of the relations. The operator modify_component_relations is particularly useful when the training images that were used during the training do not cover the entire spectrum of the relative movements.

In order to select the relations that should be modified, values for ReferenceComponent and ToleranceComponent can be passed in one of the following ways: For each of both parameters either one value, several values, or the string *'all'* can be passed. The following table summarizes the different possibilities by giving the affected relations for different combinations of parameter values. Here, four model components are assumed (0, 1, 2, and 3). If, for example, ReferenceComponent is set to *0* and ToleranceComponent is set to *1*, then the relation (0,1), which corresponds to the relative movement of component 1 with respect to component 0, will be modified.

ReferenceComponent	ToleranceComponent	Affected Relation(s)
'all'	'all'	(0,1) (0,2) (0,3) (1,0) (1,2) (1,3) (2,0) (2,1) (2,3) (3,0) (3,1) (3,2)
'all'	[1,2]	(0,1) (0,2) (1,2) (2,1) (3,1) (3,2)
[0,1]	'all'	(0,1) (0,2) (0,3) (1,0) (1,2) (1,3)
0	1	(0,1)
0	[1,2]	(0,1) (0,2)
[0,1]	2	(0,2) (1,2)
[0,1,2]	[1,2,3]	(0,1) (1,2) (2,3)

The number of tolerance values passed in `PositionTolerance` and `AngleTolerance` must be either 1 or be equal to the number of affected relations. In the former case all affected relations are modified by the same value, whereas in the latter case each relation can be modified individually by passing different values within a tuple.

Parameters

▷ **`ComponentTrainingID`** (input_control) . . . . . component_training ⇝ *handle*
Handle of the training result.

▷ **`ReferenceComponent`** (input_control) . . . . . string(-array) ⇝ *string* / integer
Model component(s) relative to which the movement(s) should be modified.
Default: 'all'
Suggested values: `ReferenceComponent` ∈ {'all', 0, 1, 2, 3, 4, 5, 6}

▷ **`ToleranceComponent`** (input_control) . . . . . string(-array) ⇝ *string* / integer
Model component(s) of which the relative movement(s) should be modified.
Default: 'all'
Suggested values: `ToleranceComponent` ∈ {'all', 0, 1, 2, 3, 4, 5, 6}

▷ **`PositionTolerance`** (input_control) . . . . . real(-array) ⇝ *real*
Change of the position relation in pixels.
Suggested values: `PositionTolerance` ∈ {1.0, 2.0, 3.0, 4.0, 5.0, 10.0, 20.0, 30.0}

▷ **`AngleTolerance`** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Change of the orientation relation in radians.
Suggested values: `AngleTolerance` ∈ {0.017, 0.035, 0.052, 0.070, 0.087, 0.175, 0.349}

Result

If the handle of the training result is valid, the operator `modify_component_relations` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ComponentTrainingID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

Matching

read_component_model (: : FileName : ComponentModelID)

Read a component model from a file.

read_component_model is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

The operator read_component_model reads a component model, which has been written with write_component_model, from the file FileName and returns it in ComponentModelID. The default HALCON file extension for the component model is 'cbm'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .cbm

▷ **ComponentModelID** (output_control) . . . . . component_model ⇝ *handle*
Handle of the component model.

Result

If the file name is valid, the operator read_component_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

Matching

read_training_components (: : FileName : ComponentTrainingID)

Read a component training result from a file.

read_training_components is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

The operator read_training_components reads a component training result, which has been written with write_training_components, from the file FileName and returns it in ComponentTrainingID. The default HALCON file extension for the component training result is 'ct'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .ct

▷ **ComponentTrainingID** (output_control) . . . . . component_training ⇝ *handle*
Handle of the training result.

Result

If the file name is valid, the operator `read_training_components` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

Matching

serialize_component_model (
: : ComponentModelID : SerializedItemHandle)

Serialize a component model.

serialize_component_model is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

`serialize_component_model` serializes the data of a component model (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The same data that is written in a file by `write_component_model` is converted to a serialized item. The component model is defined by the handle `ComponentModelID`. The serialized component model is returned by the handle `SerializedItemHandle` and can be deserialized by `deserialize_component_model`.

Parameters

▷ **ComponentModelID** (input_control) . . . . . component_model ⇝ *handle*
Handle of the component model.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_component_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

serialize_training_components (
: : ComponentTrainingID : SerializedItemHandle)

Serialize a component training result.

serialize_training_components is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

serialize_training_components serializes the data of a component training result (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_training_components is converted to a serialized item. The component training result is defined by the handle ComponentTrainingID. The serialized component training result is returned by the handle SerializedItemHandle and can be deserialized by deserialize_training_components.

Parameters

▷ **ComponentTrainingID** (input_control) ..... component_training ⇝ *handle*
Handle of the training result.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_training_components returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

train_model_components (ModelImage, InitialComponents, TrainingImages : ModelComponents : ContrastLow, ContrastHigh, MinSize, MinScore, SearchRowTol, SearchColumnTol, SearchAngleTol, TrainingEmphasis, AmbiguityCriterion, MaxContourOverlap, ClusterThreshold : ComponentTrainingID)

Train components and relations for the component-based matching.

train_model_components is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

train_model_components extracts the final (rigid) model components and trains their mutual relations, i.e., their relative movements, on the basis of the initial components by considering several training images. The result of the training is returned in the handle ComponentTrainingID. The training result can be subsequently used to create the actual component model using create_trained_component_model.

train_model_components should be used in cases where the relations of the components are not known and should be trained automatically. In contrast, if the relations are known no training needs to be performed with train_model_components. Instead, the component model can be directly created with create_component_model.

If the initial components have been automatically created by using gen_initial_components, InitialComponents contains the contour regions of the initial components. In contrast, if the initial components should be defined by the user, they can be directly passed in InitialComponents. However, instead of the contour regions for each initial component, its enclosing region must be passed in the tuple. The (contour) regions refer to the model image ModelImage. If the initial components have been obtained using gen_initial_components, the model image should be the same as in gen_initial_components. Please note that each initial component is part of at most one rigid model component. This is because during the training initial components can be merged into rigid model components if required (see below). However, they cannot be split and distributed to several rigid model components.

train_model_components uses the following approach to perform the training: In the first step, the initial components are searched in all training images. In some cases, one initial component may be found in an training image more than once. Thus, in the second step, the resulting ambiguities are solved, i.e., the most probable pose of each initial component is found. Consequently, after solving the ambiguities, in all training images at most one

pose of each initial component is available. In the next step the poses are analyzed and those initial components that do not show any relative movement are clustered to the final rigid model components. Finally, in the last step the relations between the model components are computed by analyzing their relative poses over the sequence of training images. The parameters that are associated with the mentioned steps are explained in the following.

The training is performed based on several training images, which are passed in `TrainingImages`. Each training image must show at most one instance of the compound object and should contain the full range of allowed relative movements of the model components. If, for example, the component model of an on/off switch should be trained, one training image that shows the switch turned off is sufficient if the switch in the model image is turned on, or vice versa.

The principle of the training is to find the initial components in all training images and to analyze their poses. For this, for each initial component a shape model is created (see `create_shape_model`), which is then used to determine the poses (position and orientation) of the initial components in the training images (see `find_shape_model`). Depending on the mode that is set by using `set_system ('pregenerate_shape_models',...)`, the shape model is either pregenerated completely or computed online during the search. The mode influences the computation time as well as the robustness of the training. Furthermore, it should be noted that if single-channel image are used in `ModelImage` as well as in `TrainingImages` the metric *'use_polarity'* is used internally for `create_shape_model`, while if multichannel images are used in either `ModelImage` or `TrainingImages` the metric *'ignore_color_polarity'* is used. Finally, it should be noted that while the number of channels in `ModelImage` and `TrainingImages` may be different, e.g., to facilitate model generation from synthetically generated images, the number of channels in all the images in `TrainingImages` must be identical. For further details see `create_shape_model`. The creation of the shape models can be influenced by choosing appropriate values for the parameters `ContrastLow`, `ContrastHigh`, and `MinSize`. These parameters have the same meaning as in `gen_initial_components` and can be automatically determined by passing *'auto'*: If both hysteresis threshold should be automatically determined, both `ContrastLow` and `ContrastHigh` must be set to *'auto'*. In contrast, if only one threshold value should be determined, `ContrastLow` must be set to *'auto'* while `ContrastHigh` must be set to an arbitrary value different from *'auto'*. If the initial components have been automatically created by `gen_initial_components`, the parameters `ContrastLow`, `ContrastHigh`, and `MinSize` should be set to the same values as in `gen_initial_components`.

To influence the search for the initial components, the parameters `MinScore`, `SearchRowTol`, `SearchColumnTol`, `SearchAngleTol`, and `TrainingEmphasis` can be set. The parameter `MinScore` determines what score a potential match must at least have to be regarded as an instance of the initial component in the training image. The larger `MinScore` is chosen, the faster the training is. If the initial components can be expected never to be occluded in the training images, `MinScore` may be set as high as 0.8 or even 0.9 (see `find_shape_model`).

By default, the components are searched only at points in which the component lies completely within the respective training image. This means that a component will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than `MinScore`. This behavior can be changed with `set_system ('border_shape_models','true')`, which will cause components that extend beyond the image border to be found if they achieve a score greater than `MinScore`. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the training will increase in this mode.

When dealing with a high number of initial components and many training images, the training may take a long time (up to several minutes). In order to speed up the training it is possible to restrict the search space for the single initial components in the training images. For this, the poses of the initial components in the model image are used as reference pose. The parameters `SearchRowTol` and `SearchColumnTol` specify the position tolerance region relative to the reference position in which the search is performed. Assume, for example, that the position of an initial component in the model image is (100,200) and `SearchRowTol` is set to *20* and `SearchColumnTol` is set to *10*. Then, this initial component is searched in the training images only within the axis-aligned rectangle that is determined by the upper left corner (80,190) and the lower right corner (120,210). The same holds for the orientation angle range, which can be restricted by specifying the angle tolerance `SearchAngleTol` to the angle range of [-`SearchAngleTol`,+`SearchAngleTol`]. Thus, it is possible to considerably reduce the computational effort during the training by an adequate acquisition of the training images. If one of the three parameters is set to *-1*, no restriction of the parameter space is applied in the corresponding dimension.

The input parameters `ContrastLow`, `ContrastHigh`, `MinSize`, `MinScore`, `SearchRowTol`, `SearchColumnTol`, and `SearchAngleTol` must either contain one element, in which case the parameter is used for all initial components, or must contain the same number of elements as the initial components contained

in InitialComponents, in which case each parameter element refers to the corresponding initial component in InitialComponents.

The parameter TrainingEmphasis offers another possibility to influence the computation time of the training and to simultaneously affect its robustness. If TrainingEmphasis is set to *'speed'*, on the one hand the training is comparatively fast, on the other hand it may happen in some cases that some initial components are not found in the training images or are found at a wrong pose. Consequently, this would lead to an incorrect computation of the rigid model components and their relations. The poses of the found initial components in the individual training images can be examined by using get_training_components. If erroneous matches occur the training should be restarted with TrainingEmphasis set to *'reliability'*. This results in a higher robustness at the cost of a longer computation time.

Furthermore, during the pose determination of the initial components ambiguities may occur if the initial components are rotationally symmetric or if several initial components are identical or at least similar to each other. To solve the ambiguities, the most probable pose is calculated for each initial component in each training image. For this, the individual ambiguous poses are evaluated. The pose of an initial component receives a good evaluation if the relative pose of the initial component with respect to the other initial components is similar to the corresponding relative pose in the model image. The method to evaluate this similarity can be chosen with AmbiguityCriterion. In almost all cases the best results are obtained with *'rigidity'*, which assumes the rigidity of the compound object. The more the rigidity of the compound object is violated by the pose of the initial component, the worse its evaluation is. In the case of *'distance'*, only the distance between the initial components is considered during the evaluation. Hence, the pose of the initial component receives a good evaluation if its distances to the other initial components is similar to the corresponding distances in the model image. Accordingly, when choosing *'orientation'*, only the relative orientation is considered during the evaluation. Finally, the simultaneous consideration of distance and orientation can be achieved by choosing *'distance_orientation'*. In contrast to *'rigidity'*, the relative pose of the initial components is not considered when using *'distance_orientation'*.

The process of solving the ambiguities can be further influenced by the parameter MaxContourOverlap. This parameter describes the extent by which the contours of two initial component matches may overlap each other. Let the letters 'I' and 'T', for example, be two initial components that should be searched in a training image that shows the string 'IT'. Then, the initial component 'T' should be found at its correct pose. In contrast, the initial component 'I' will be found at its correct pose ('I') but also at the pose of the 'T' because of the similarity of the two components. To discard the wrong match of the initial component 'I', an appropriate value for MaxContourOverlap can be chosen: If overlapping matches should be tolerated, MaxContourOverlap should be set to *1*. If overlapping matches should be completely avoided, MaxContourOverlap should be set to *0*. By choosing a value between *0* and *1*, the maximum percentage of overlapping contour pixels can be adjusted.

The decision which initial components can be clustered to rigid model components is made based on the poses of the initial components in the model image and in the training images. Two initial components are merged if they do not show any relative movement over all images. Assume that in the case of the above mentioned switch the training image would show the same switch state as the model image, the algorithm would merge the respective initial components because it assumes that the entire switch is one rigid model component. The extent by which initial components are merged can be influenced with the parameter ClusterThreshold. This cluster threshold is based on the probability that two initial components belong to the same rigid model component. Thus, ClusterThreshold describes the minimum probability which two initial components must have in order to be merged. Since the threshold is based on a probability value, it must lie in the interval between *0* and *1*. The greater the threshold is chosen, the smaller the number of initial components that are merged. If a threshold of *0* is chosen, all initial components are merged into one rigid component, while for a threshold of *1* no merging is performed and each initial component is adopted as one rigid model component.

The final rigid model components are returned in ModelComponents. Later, the index of a component region in ModelComponents is used to denote the model component. The poses of the components in the training images can be examined by using get_training_components.

After the determination of the model components their relative movements are analyzed by determining the movement of one component with respect to a second component for each pair of components. For this, the components are referred to their reference points. The reference point of a component is the center of gravity of its contour region, which is returned in ModelComponents. It can be calculated by calling area_center. Finally, the relative movement is represented by the smallest enclosing rectangle of arbitrary orientation of the reference point movement and by the smallest enclosing angle interval of the relative orientation of the second component over all images. The determined relations can be inspected by using get_component_relations.

Parameters

▷ **ModelImage** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image from which the shape models of the initial components should be created.

▷ **InitialComponents** (input_object) . . . . . region-array ⇝ *object*
Contour regions or enclosing regions of the initial components.

▷ **TrainingImages** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Training images that are used for training the model components.

▷ **ModelComponents** (output_object) . . . . . region(-array) ⇝ *object*
Contour regions of rigid model components.

▷ **ContrastLow** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Lower hysteresis threshold for the contrast of the initial components in the image.
Default: 'auto'
Suggested values: `ContrastLow` ∈ {'auto', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}
Restriction: ContrastLow > 0

▷ **ContrastHigh** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Upper hysteresis threshold for the contrast of the initial components in the image.
Default: 'auto'
Suggested values: `ContrastHigh` ∈ {'auto', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}
Restriction: ContrastHigh > 0 && ContrastHigh >= ContrastLow

▷ **MinSize** (input_control) . . . . . integer(-array) ⇝ *integer* / string
Minimum size of connected contour regions.
Default: 'auto'
Suggested values: `MinSize` ∈ {'auto', 0, 5, 10, 20, 30, 40}
Restriction: MinSize >= 0

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the initial components to be found.
Default: 0.5
Suggested values: `MinScore` ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= MinScore && MinScore <= 1

▷ **SearchRowTol** (input_control) . . . . . integer(-array) ⇝ *integer*
Search tolerance in row direction.
Default: -1
Suggested values: `SearchRowTol` ∈ {0, 10, 20, 30, 50, 100}
Restriction: SearchRowTol == -1 || SearchColumnTol >= 0

▷ **SearchColumnTol** (input_control) . . . . . integer(-array) ⇝ *integer*
Search tolerance in column direction.
Default: -1
Suggested values: `SearchColumnTol` ∈ {0, 10, 20, 30, 50, 100}
Restriction: SearchColumnTol == -1 || SearchColumnTol >= 0

▷ **SearchAngleTol** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Angle search tolerance.
Default: -1
Suggested values: `SearchAngleTol` ∈ {0.0, 0.17, 0.39, 0.78, 1.57}
Restriction: SearchAngleTol == -1 || SearchAngleTol >= 0

▷ **TrainingEmphasis** (input_control) . . . . . string ⇝ *string*
Decision whether the training emphasis should lie on a fast computation or on a high robustness.
Default: 'speed'
List of values: `TrainingEmphasis` ∈ {'speed', 'reliability'}

▷ **AmbiguityCriterion** (input_control) . . . . . string ⇝ *string*
Criterion for solving ambiguous matches of the initial components in the training images.
Default: 'rigidity'
List of values: `AmbiguityCriterion` ∈ {'distance', 'orientation', 'distance_orientation', 'rigidity'}

▷ **MaxContourOverlap** (input_control) . . . . . real ⇝ *real*
Maximum contour overlap of the found initial components in a training image.
Default: 0.2
Suggested values: MaxContourOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Minimum increment: 0.01
Recommended increment: 0.05
Restriction: 0 <= MaxContourOverlap && MaxContourOverlap <= 1

▷ **ClusterThreshold** (input_control) . . . . . real ⇝ *real*
Threshold for clustering the initial components.
Default: 0.5
Suggested values: ClusterThreshold ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: 0 <= ClusterThreshold && ClusterThreshold <= 1

▷ **ComponentTrainingID** (output_control) . . . . . component_training ⇝ *handle*
Handle of the training result.

Result

If the parameter values are correct, the operator train_model_components returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

Matching

write_component_model (: : ComponentModelID, FileName :)

Write a component model to a file.

write_component_model is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

The operator write_component_model writes the component model ComponentModelID to the file FileName. The model can be read again with read_component_model. The default HALCON file extension for the component model is 'cbm'.

Parameters

▷ **ComponentModelID** (input_control) . . . . . component_model ⇝ *handle*
Handle of the component model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: .cbm

Result

If the file name is valid (write permission), the operator write_component_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

write_training_components (: : ComponentTrainingID, FileName :)

Write a component training result to a file.

write_training_components is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 26.11.

The operator write_training_components writes the component training result ComponentTrainingID to the file FileName. The training result can be read again with read_training_components. The default HALCON file extension for the component training result is 'ct'.

Parameters

▷ **ComponentTrainingID** (input_control) ..... component_training ⇝ *handle*
Handle of the training result.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
File name.
File extension: .ct

Result

If the file name is valid (write permission), the operator write_training_components returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

17.11 Morphology

closing_golay (Region : RegionClosing : GolayElement, Rotation :)

Close a region with an element from the Golay alphabet.

closing_golay is obsolete and is only provided for reasons of backward compatibility.

closing_golay is defined as a Minkowski addition followed by a Minkowski subtraction. First the Minkowski addition of the input region (Region) with the structuring element from the Golay alphabet defined by GolayElement and Rotation is computed. Then the Minkowski subtraction of the result and the structuring element rotated by 180° is performed.

The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number Rotation determines which rotation of the element should be used, and whether the foreground (even) or background version (odd) of the selected element should be used. The Golay elements, together with all possible rotations, are described with the operator golay_elements.

closing_golay serves to close holes smaller than the structuring element, and to smooth regions' boundaries.

Attention

Not all values of Rotation are valid for any Golay element. For some of the values of Rotation, the resulting regions are identical to the input regions.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be closed.

▷ **RegionClosing** (output_object) ..... region(-array) ⇝ *object*
Closed regions.

▷ **GolayElement** (input_control) ..... string ⇝ *string*
Structuring element from the Golay alphabet.
Default: 'h'
List of values: `GolayElement` ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}

▷ **Rotation** (input_control) ..... integer ⇝ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default: 0
List of values: `Rotation` ∈ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(6 \cdot \sqrt{F}) \ .$$

Result

`closing_golay` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`closing`

See also

`erosion_golay`, `dilation_golay`, `opening_golay`, `hit_or_miss_golay`, `thinning_golay`, `thickening_golay`, `golay_elements`

Module

Foundation

dilation_golay (Region : RegionDilation : GolayElement, Iterations, Rotation :)

Dilate a region with an element from the Golay alphabet.

dilation_golay is obsolete and is only provided for reasons of backward compatibility.

`dilation_golay` dilates a region with the selected element `GolayElement` from the Golay alphabet. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number Rotation determines which rotation of the element should be used, and whether the foreground (even) or background version (odd) of the selected element should be used. The Golay elements, together with all possible rotations, are described with the operator golay_elements. The operator works by shifting the structuring element over the region to be processed (Region). For all positions of the structuring element that intersect the region, the corresponding reference point (relative to the structuring element) is added to the output region. This means that the union of all translations of the structuring element within the region is computed.

The parameter Iterations determines the number of iterations which are to be performed with the structuring element. The result of iteration $n-1$ is used as input for iteration n.

Attention

Not all values of Rotation are valid for any Golay element. For some of the values of Rotation, the resulting regions are identical to the input regions.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Regions to be dilated.
- ▷ **RegionDilation** (output_object) ..... region(-array) ⇝ *object*
  Dilated regions.
- ▷ **GolayElement** (input_control) ..... string ⇝ *string*
  Structuring element from the Golay alphabet.
  Default: 'h'
  List of values: GolayElement ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}
- ▷ **Iterations** (input_control) ..... integer ⇝ *integer*
  Number of iterations.
  Default: 1
  Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
  Value range: 1 ≤ Iterations (lin)
  Minimum increment: 1
  Recommended increment: 1
- ▷ **Rotation** (input_control) ..... integer ⇝ *integer*
  Rotation of the Golay element. Depending on the element, not all rotations are valid.
  Default: 0
  List of values: Rotation ∈ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(3 \cdot \sqrt{F}) \ .$$

Result

dilation_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

dilation1, dilation2, dilation_seq

See also

erosion_golay, opening_golay, closing_golay, hit_or_miss_golay, thinning_golay, thickening_golay, golay_elements

Module

Foundation

dilation_seq (Region : RegionDilation : GolayElement, Iterations :)

Dilate a region sequentially.

dilation_seq is obsolete and is only provided for reasons of backward compatibility.

dilation_seq computes the sequential dilation of the input region Region with the selected structuring element GolayElement from the Golay alphabet. This is done by executing the operator dilation_golay with all rotations of the structuring element Iterations times. The following structuring elements can be selected:

'l', 'd', 'c', 'f', 'h', 'k'.

In order to compute the skeleton of a region, usually the elements 'l' and 'm' are used. Only the "foreground elements" (even rotation numbers) are used. The elements 'i' and 'e' result in unchanged output regions. The elements 'l', 'm' and 'f2' are identical for the foreground. The Golay elements, together with all possible rotations, are described with the operator golay_elements.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be dilated.

▷ **RegionDilation** (output_object) ..... region(-array) ⇝ *object*
Dilated regions.

▷ **GolayElement** (input_control) ..... string ⇝ *string*
Structuring element from the Golay alphabet.
Default: 'h'
List of values: GolayElement ∈ {'l', 'd', 'c', 'f', 'h', 'k'}

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(\texttt{Iterations} \cdot 20 \cdot \sqrt{F}) \ .$$

Result

dilation_seq returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

dilation1, dilation2, dilation_golay

See also

erosion_seq, hit_or_miss_seq, thinning_seq

Module

Foundation

erosion_golay (Region : RegionErosion : GolayElement, Iterations, Rotation :)

Erode a region with an element from the Golay alphabet.

erosion_golay is obsolete and is only provided for reasons of backward compatibility.

erosion_golay erodes a region with the selected element GolayElement from the Golay alphabet. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number Rotation determines which rotation of the element should be used, and whether the foreground (even) or background version (odd) of the selected element should be used. The Golay elements, together with all possible rotations, are described with the operator golay_elements. The operator works by shifting the structuring element over the region to be processed (Region). For all positions of the structuring element fully contained in the region, the corresponding reference point (relative to the structuring element) is added to the output region. This means that the intersection of all translations of the structuring element within the region is computed.

The parameter Iterations determines the number of iterations which are to be performed with the structuring element. The result of iteration $n-1$ is used as input for iteration n.

Attention

Not all values of Rotation are valid for any Golay element. For some of the values of Rotation, the resulting regions are identical to the input regions.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be eroded.

▷ **RegionErosion** (output_object) ..... region(-array) ⇝ *object*
Eroded regions.

▷ **GolayElement** (input_control) ..... string ⇝ *string*
Structuring element from the Golay alphabet.
Default: 'h'
List of values: GolayElement ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Rotation** (input_control) ..... integer ⇝ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default: 0
List of values: Rotation ∈ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(3 \cdot \sqrt{F}) \ .$$

Result

`erosion_golay` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`erosion_seq`, `erosion1`, `erosion2`

See also

`dilation_golay`, `opening_golay`, `closing_golay`, `hit_or_miss_golay`, `thinning_golay`, `thickening_golay`, `golay_elements`

Module

Foundation

erosion_seq (Region : RegionErosion : GolayElement, Iterations :)

Erode a region sequentially.

`erosion_seq` is obsolete and is only provided for reasons of backward compatibility.

`erosion_seq` computes the sequential erosion of the input region `Region` with the selected structuring element `GolayElement` from the Golay alphabet. This is done by executing the operator `erosion_golay` with all rotations of the structuring element `Iterations` times. The following structuring elements can be selected:

'l', 'd', 'c', 'f', 'h', 'k'.

Only the "foreground elements" (even rotation numbers) are used. The elements 'i' and 'e' result in unchanged output regions. The elements 'l', 'm' and 'f2' are identical for the foreground. The Golay elements, together with all possible rotations, are described with the operator `golay_elements`.

Parameters

▷ **`Region`** (input_object) ..... region(-array) ⇝ *object*
Regions to be eroded.

▷ **`RegionErosion`** (output_object) ..... region(-array) ⇝ *object*
Eroded regions.

▷ **`GolayElement`** (input_control) ..... string ⇝ *string*
Structuring element from the Golay alphabet.
Default: 'h'
List of values: `GolayElement` ∈ {'l', 'd', 'c', 'f', 'h', 'k'}

▷ **Iterations** (input_control) . . . . . integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(\texttt{Iterations} \cdot 20 \cdot \sqrt{F}) \ .$$

Result

erosion_seq returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, watersheds, class_ndim_norm

Possible Successors

connection, reduce_domain, select_shape, area_center

Alternatives

erosion_golay, erosion1, erosion2

See also

dilation_seq, hit_or_miss_seq, thinning_seq

Module

Foundation

fitting (Region, StructElements : RegionFitted : :)

Perform a closing after an opening with multiple structuring elements.

fitting is obsolete and is only provided for reasons of backward compatibility.

fitting performs an opening and a closing successively on the input regions. The eight structuring elements normally used for this operation can be generated with the operator gen_struct_elements. However, other user-defined structuring elements can also be used. Let R be the input region(s) and let M_i denote the structuring elements. Furthermore, let P be the result of the opening and Q be the final result. Then the operator can be formalized as follows:

$$P = \bigcup_{i=1}^{n} (R \circ M_i)$$
$$Q = \bigcap_{i=1}^{n} (P \bullet M_i)$$

Regions larger than the structuring elements are preserved, while small gaps are closed.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **StructElements** (input_object) ..... region(-array) ⇝ *object*
Structuring elements.

▷ **RegionFitted** (output_object) ..... region(-array) ⇝ *object*
Fitted regions.

Result

`fitting` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_struct_elements`, `gen_region_points`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`opening`, `closing`, `connection`, `select_shape`

Module

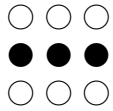
Foundation

gen_struct_elements (: StructElements : Type, Row, Column :)

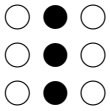
Generate standard structuring elements.

gen_struct_elements is obsolete and is only provided for reasons of backward compatibility.

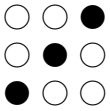
`gen_struct_elements` serves to generate eight structuring elements normally used in the operator `fitting`. The default value *'noise'* of the parameter `Type` generates elements especially suited for the elimination of noise.



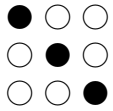
M_1



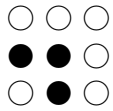
M_2



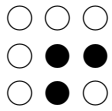
M_3



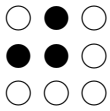
M_4



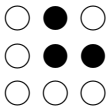
M_5



M_6



M_7



M_8

Parameters

▷ **StructElements** (output_object) ..... region(-array) ⇝ *object*
Generated structuring elements.

▷ **Type** (input_control) ..... string ⇝ *string*
Type of structuring element to generate.
Default: 'noise'
List of values: Type ∈ {'noise'}

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Row coordinate of the reference point.
Default: 1
Suggested values: Row ∈ {0, 1, 10, 50, 100, 200, 300, 400}
(lin)

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column coordinate of the reference point.
Default: 1
Suggested values: Column ∈ {0, 1, 10, 50, 100, 200, 300, 400}
(lin)

Result

gen_struct_elements returns 2 (H_MSG_TRUE) if all parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fitting, hit_or_miss, opening, closing, erosion2, dilation2

See also

golay_elements

Module

Foundation

golay_elements (: StructElement1, StructElement2 : GolayElement, Rotation, Row, Column :)

Generate the structuring elements of the Golay alphabet.

golay_elements is obsolete and is only provided for reasons of backward compatibility.

golay_elements generates the structuring elements from the Golay alphabet. The parameter GolayElement determines the name of the structuring element, while Rotation determines its rotation. The structuring elements are intended for use in hit_or_miss: In StructElement1 the structuring element for the foreground is returned, while in StructElement2 the structuring element for the background is returned. Row and Column determine the reference point of the structuring element.

The rotations are numbered from 0 to 15. This does not mean, however, that there are 16 different rotations: Even values denote rotations of the foreground elements, while odd values denote rotations of the background elements.

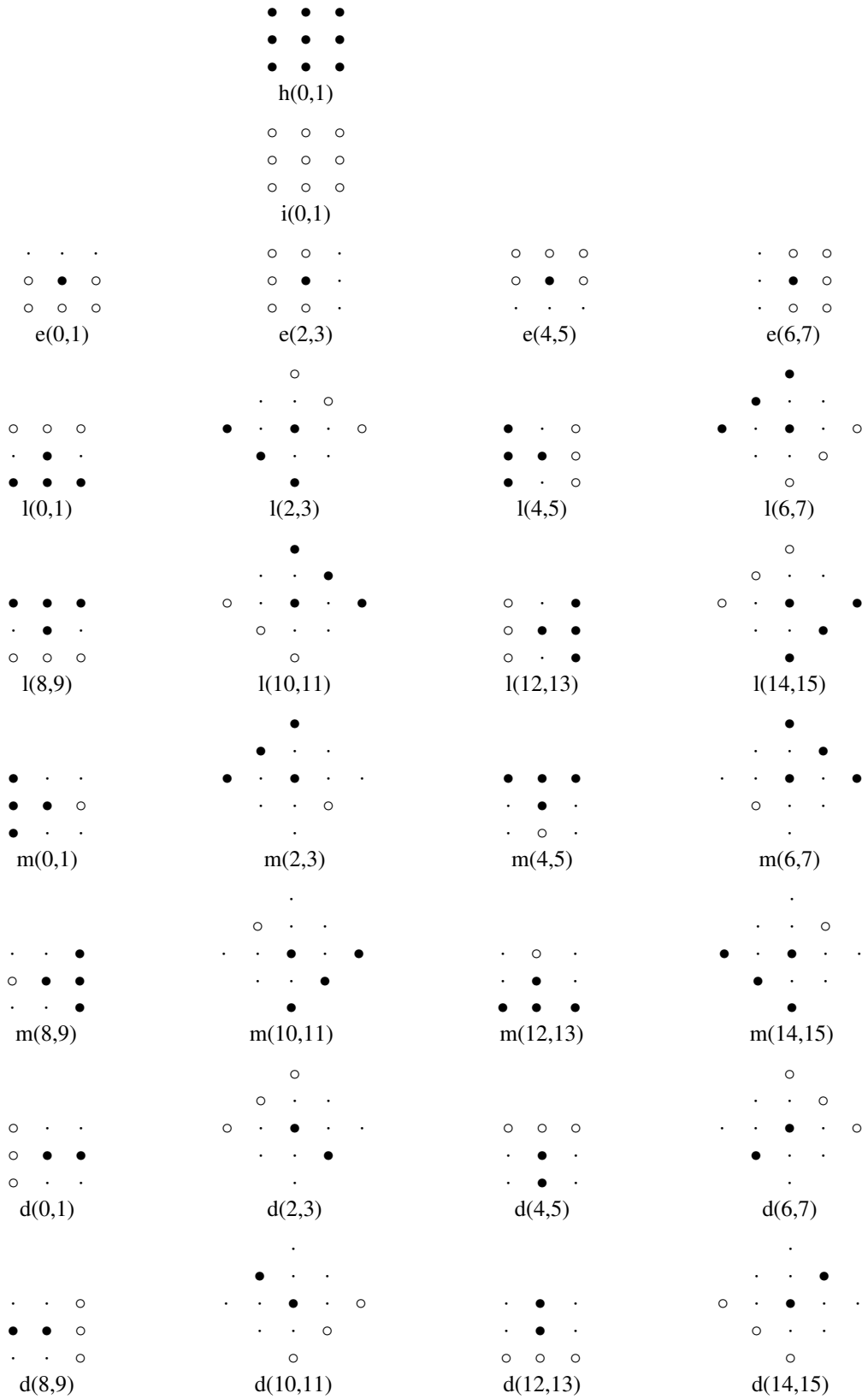
For golay_elements only even values are accepted, and determine the Golay element for StructElement1. The next larger odd value is used for StructElement2. There are no rotations for the Golay elements 'h' and 'i'. Therefore, only the values 0 and 1 are possible as "rotations" (and hence only 0 for golay_elements). The element 'e' has only four possible rotations, and hence the rotation must be between 0 and 7 (for golay_elements the values 0, 2, 4, or 6 must be used).

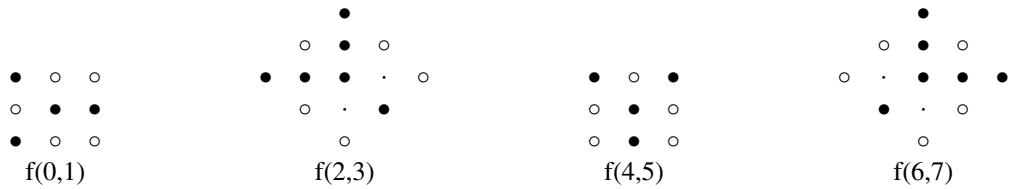
The tables below show the elements of the Golay alphabet with all possible rotations. The characters used have the following meaning:

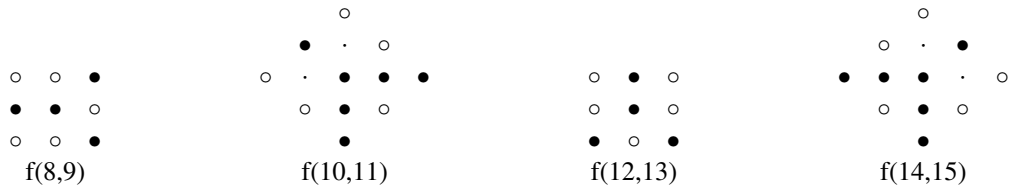
- Foreground pixel

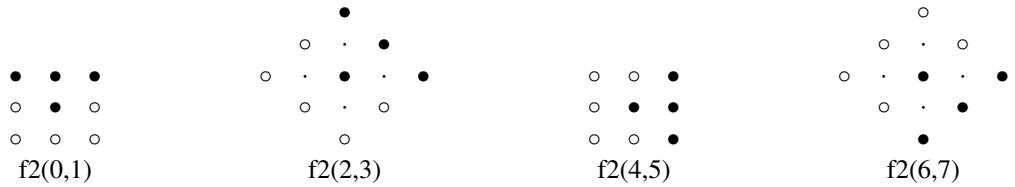
- ○ Background pixel
- · Don't care pixel

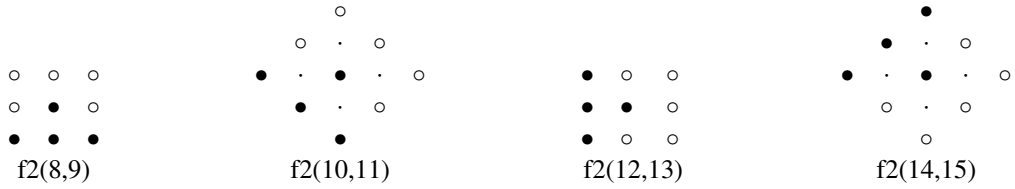
The names of the elements and their rotation numbers are displayed below the respective element. The elements with even number contain the foreground pixels, while the elements with odd numbers contain the background pixels.

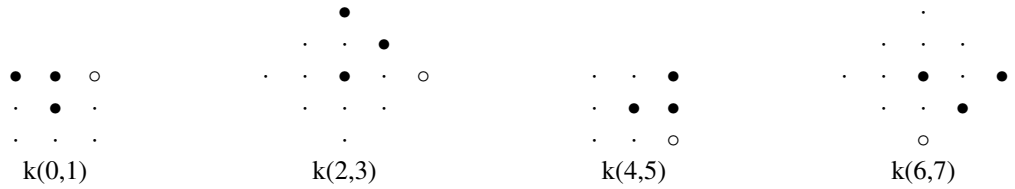


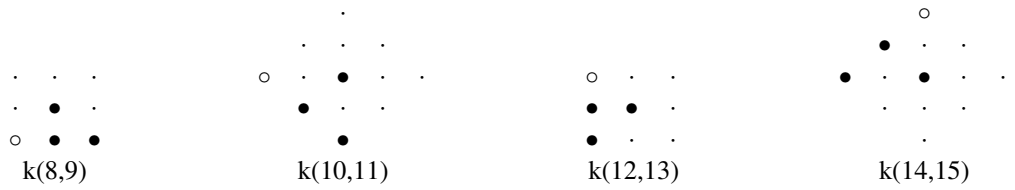

f(0,1) f(2,3) f(4,5) f(6,7)

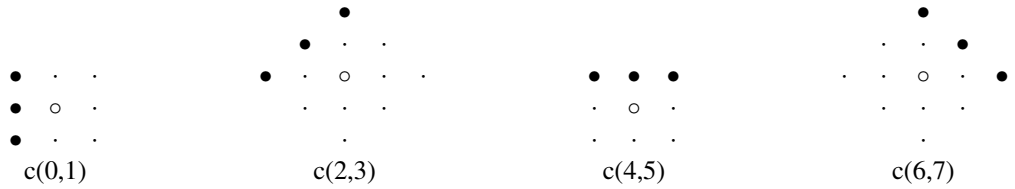

f(8,9) f(10,11) f(12,13) f(14,15)

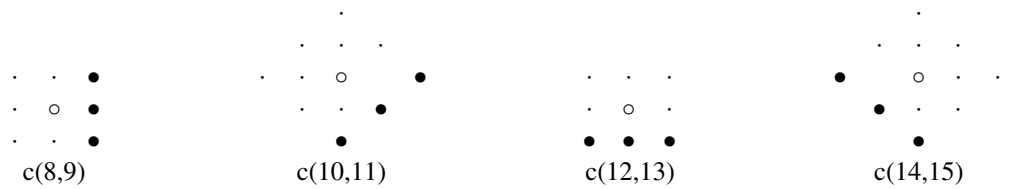

f2(0,1) f2(2,3) f2(4,5) f2(6,7)


f2(8,9) f2(10,11) f2(12,13) f2(14,15)


k(0,1) k(2,3) k(4,5) k(6,7)


k(8,9) k(10,11) k(12,13) k(14,15)


c(0,1) c(2,3) c(4,5) c(6,7)


c(8,9) c(10,11) c(12,13) c(14,15)

Parameters

▷ **`StructElement1`** (output_object) ..... region ⇝ *object*
Structuring element for the foreground.

▷ **`StructElement2`** (output_object) ..... region ⇝ *object*
Structuring element for the background.

▷ **GolayElement** (input_control) . . . . . string ⇝ *string*
Name of the structuring element.
Default: 'l'
List of values: GolayElement ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}

▷ **Rotation** (input_control) . . . . . integer ⇝ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default: 0
List of values: Rotation ∈ {0, 2, 4, 6, 8, 10, 12, 14}

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row coordinate of the reference point.
Default: 16
Suggested values: Row ∈ {0, 16, 32, 128, 256}
Value range: 0 ≤ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column coordinate of the reference point.
Default: 16
Suggested values: Column ∈ {0, 16, 32, 128, 256}
Value range: 0 ≤ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Result

golay_elements returns 2 (H_MSG_TRUE) if all parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

hit_or_miss

Alternatives

gen_region_points, gen_struct_elements, gen_region_polygon_filled

See also

dilation_golay, erosion_golay, opening_golay, closing_golay, hit_or_miss_golay, thickening_golay

References

J. Serra: "Image Analysis and Mathematical Morphology". Volume I. Academic Press, 1982

Module

Foundation

hit_or_miss_golay (Region : RegionHitMiss : GolayElement, Rotation :)

Hit-or-miss operation for regions using the Golay alphabet.

hit_or_miss_golay is obsolete and is only provided for reasons of backward compatibility.

hit_or_miss_golay performs the hit-or-miss-transformation for the input regions Region (using structuring elements from the Golay alphabet). First, an erosion with the foreground of the structuring element GolayElement is done on the input region Region. Then an erosion with the background of the structuring element GolayElement is performed on the complement of the input region. The intersection of the two resulting regions is the result RegionHitMiss of hit_or_miss_golay. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e','i', 'f', 'f2', 'h', 'k'.

The rotation number Rotation determines which rotation of the element should be used. The hit-or-miss-transformation selects precisely the points for which the conditions given by the selected Golay element are fulfilled.

Attention

Not all values of Rotation are valid for any Golay element.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Regions to be processed.
- ▷ **RegionHitMiss** (output_object) ..... region(-array) ⇝ *object*
  Result of the hit-or-miss operation.
- ▷ **GolayElement** (input_control) ..... string ⇝ *string*
  Structuring element from the Golay alphabet.
  Default: 'h'
  List of values: GolayElement ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}
- ▷ **Rotation** (input_control) ..... integer ⇝ *integer*
  Rotation of the Golay element. Depending on the element, not all rotations are valid.
  Default: 0
  List of values: Rotation ∈ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(6 \cdot \sqrt{F}) \ .$$

Result

hit_or_miss_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

hit_or_miss_seq, hit_or_miss

See also

erosion_golay, dilation_golay, opening_golay, closing_golay, thinning_golay, thickening_golay, golay_elements

Module

Foundation

hit_or_miss_seq (Region : RegionHitMiss : GolayElement :)

Hit-or-miss operation for regions using the Golay alphabet (sequential).

hit_or_miss_seq is obsolete and is only provided for reasons of backward compatibility.

hit_or_miss_golay performs the hit-or-miss-transformation for the input regions Region using all rotations of a structuring element from the Golay alphabet. The result of the operator is the union of all intermediate results of the respective rotations. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The Golay elements, together with all possible rotations, are described with the operator golay_elements.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **RegionHitMiss** (output_object) ..... region(-array) ⇝ *object*
Result of the hit-or-miss operation.

▷ **GolayElement** (input_control) ..... string ⇝ *string*
Structuring element from the Golay alphabet.
Default: 'h'
List of values: GolayElement ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}

Complexity

Let F be the area of an input region, and R be the number of rotations. Then the runtime complexity for one region is:

$$O(R \cdot 6 \cdot \sqrt{F}) \ .$$

Result

hit_or_miss_seq returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

hit_or_miss_golay, hit_or_miss

See also

thinning_seq, thickening_seq

Module

Foundation

morph_hat (Region, StructElement : RegionMorphHat : :)

Compute the union of bottom_hat and top_hat.

morph_hat is obsolete and is only provided for reasons of backward compatibility.

morph_hat computes the union of the regions that are removed by an opening operation with the regions that are added by a closing operation. Hence this is the union of the results of top_hat and bottom_hat. The position of StructElement does not influence the result.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Attention

The individual regions are processed separately.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Regions to be processed.
- ▷ **StructElement** (input_object) ..... region ⇝ *object*
  Structuring element (position-invariant).
- ▷ **RegionMorphHat** (output_object) ..... region(-array) ⇝ *object*
  Union of top hat and bottom hat.

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"

main()
{
  cout << "Reproduction of 'dilation_circle ()'" << endl;
  cout << "First = original image " << endl;
  cout << "Red   = after segmentation " << endl;
  cout << "Blue  = after erosion " << endl;

  HByteImage img("monkey");
  HWindow    w;

  HRegion      circ   = HRegion::GenCircle (10, 10, 1.5);
  HRegionArray regs   = (img >= 128).Connection();
  HRegionArray tophat = regs.TopHat (circ);
  HRegionArray bothat = regs.BottomHat (circ);
  HRegionArray unionX = tophat.Union2 (bothat);

                        img.Display (w);      w.Click ();
  w.SetColor ("red");   regs.Display (w);     w.Click ();
  w.SetColor ("blue");  tophat.Display (w);   w.Click ();
  w.SetColor ("green"); bothat.Display (w);   w.Click ();
  w.SetColor ("white"); unionX.Display (w);   w.Click ();

  return(0);
}
```

Result

morph_hat returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_region_polygon_filled

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

top_hat, bottom_hat, union2

See also

opening, closing

Module

Foundation

morph_skeleton (Region : RegionSkeleton : :)

Compute the morphological skeleton of a region.

morph_skeleton is obsolete and is only provided for reasons of backward compatibility.

morph_skeleton computes the skeleton of the input regions (Region) using morphological transformations. The computation yields a disconnected skeleton (gaps in the diagonals) having a width of one or two pixels. The calculation uses the Golay element 'h', i.e., an 8-neighborhood. This is equivalent to the maximum-norm.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **RegionSkeleton** (output_object) ..... region(-array) ⇝ *object*
Resulting morphological skeleton.

Result

morph_skeleton returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

skeleton, reduce_domain, select_shape, area_center, connection

Alternatives

skeleton

Module

Foundation

morph_skiz (Region : RegionSkiz : Iterations1, Iterations2 :)

Thinning of a region.

morph_skiz is obsolete and is only provided for reasons of backward compatibility.

morph_skiz first performs a sequential thinning (thinning_seq) of the input region with the element 'l' of the Golay alphabet. The number of iterations is determined by the parameter Iterations1. Then a sequential thinning of the resulting region with the element 'e' of the Golay alphabet is carried out. The number of iterations for this step is determined by the parameter Iterations2. The skiz operation serves to compute a kind of skeleton of the input regions, and to prune the branches of the resulting skeleton. If the skiz operation is applied to the complement of the region, the region and the resulting skeleton are separated.

If very large values or 'maximal' are passed for Iterations1 or Iterations2, the processing stops if no more changes occur.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be thinned.

▷ **RegionSkiz** (output_object) ..... region(-array) ⇝ *object*
Result of the skiz operator.

▷ **Iterations1** (input_control) ..... integer ⇝ *integer* / string
Number of iterations for the sequential thinning with the element 'l' of the Golay alphabet.
Default: 100
Suggested values: Iterations1 ∈ {'maximal', 0, 1, 2, 3, 5, 7, 10, 15, 20, 30, 40, 50, 70, 100, 150, 200, 300, 400}
Value range: 0 ≤ Iterations1 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Iterations2** (input_control) ..... integer ⇝ *integer* / string
Number of iterations for the sequential thinning with the element 'e' of the Golay alphabet.
Default: 1
Suggested values: Iterations2 ∈ {'maximal', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 0 ≤ Iterations2 (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let F be the area of the input region. Then the runtime complexity for one region is

$$O((\texttt{Iterations1} + \texttt{Iterations2}) \cdot 3 \cdot \sqrt{F}) \ .$$

Result

morph_skiz returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

pruning, reduce_domain, select_shape, area_center, connection, background_seg, complement

Alternatives

skeleton, thinning_seq, morph_skeleton, interjacent

See also

thinning, hit_or_miss_seq, difference

Module

Foundation

opening_golay (Region : RegionOpening : GolayElement, Rotation :)

Open a region with an element from the Golay alphabet.

opening_golay is obsolete and is only provided for reasons of backward compatibility.

opening_golay is defined as a Minkowski subtraction followed by a Minkowski addition. First the Minkowski subtraction of the input region (Region) with the structuring element from the Golay alphabet defined by GolayElement and Rotation is computed. Then the Minkowski addition of the result and the structuring element rotated by 180° is performed.

The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number Rotation determines which rotation of the element should be used, and whether the foreground (even) or background version (odd) of the selected element should be used. The Golay elements, together with all possible rotations, are described with the operator golay_elements.

opening_golay serves to eliminate regions smaller than the structuring element, and to smooth regions' boundaries.

Attention

Not all values of Rotation are valid for any Golay element. For some of the values of Rotation, the resulting regions are identical to the input regions.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Regions to be opened.
- ▷ **RegionOpening** (output_object) ..... region(-array) ⇝ *object*
  Opened regions.
- ▷ **GolayElement** (input_control) ..... string ⇝ *string*
  Structuring element from the Golay alphabet.
  Default: 'h'
  List of values: GolayElement ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}
- ▷ **Rotation** (input_control) ..... integer ⇝ *integer*
  Rotation of the Golay element. Depending on the element, not all rotations are valid.
  Default: 0
  List of values: Rotation ∈ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(6 \cdot \sqrt{F}) \ .$$

Result

opening_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

opening_seg, opening

See also

erosion_golay, dilation_golay, closing_golay, hit_or_miss_golay, thinning_golay, thickening_golay, golay_elements

Module

Foundation

opening_seg (Region, StructElement : RegionOpening : :)

Separate overlapping regions.

opening_seg is obsolete and is only provided for reasons of backward compatibility.

The opening_seg operation is defined as a sequence of the following operators: erosion1, connection and dilation1 (see example). Only one iteration is done in erosion1 and dilation1.

opening_seg serves to separate overlapping regions whose area of overlap is smaller than StructElement. It should be noted that the resulting regions can overlap without actually merging (see expand_region). opening_seg uses the center of gravity as the reference point of the structuring element.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be opened.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element (position-invariant).

▷ **RegionOpening** (output_object) ..... region-array ⇝ *object*
Opened regions.

Example

```
* Simulation of opening_seg
* opening_seg(Region,StructElement,RegionOpening):
    erosion1(Region,StructElement,H1,1)
    connection(H1,H2)
    dilation1(H2,StructElement,RegionOpening,1)
```

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \sqrt{\sqrt{F1}}) \ .$$

Result

opening_seg returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_region_polygon_filled

Possible Successors

expand_region, reduce_domain, select_shape, area_center, connection

Alternatives

erosion1, connection, dilation1

Module

Foundation

thickening (Region, StructElement1, StructElement2 : RegionThick : Row, Column, Iterations :)

Add the result of a hit-or-miss operation to a region.

thickening is obsolete and is only provided for reasons of backward compatibility.

thickening performs a thickening of the input regions using morphological operations. The operator first applies a hit-or-miss-transformation to Region (cf. hit_or_miss), and then adds the detected points to the input region. The parameter Iterations determines the number of iterations performed.

For the choice of the structuring elements StructElement1 and StructElement2, as well as for Row and Column, the same restrictions described under hit_or_miss apply.

The structuring elements (StructElement1 and StructElement2) can be generated by calling golay_elements, for example.

Attention

If the reference point is contained in StructElement1 the input region remains unchanged.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **StructElement1** (input_object) ..... region ⇝ *object*
Structuring element for the foreground.

▷ **StructElement2** (input_object) ..... region ⇝ *object*
Structuring element for the background.

▷ **RegionThick** (output_object) ..... region(-array) ⇝ *object*
Result of the thickening operator.

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Row coordinate of the reference point.
Default: 16
Suggested values: Row ∈ {0, 2, 4, 8, 16, 32, 128}
Value range: 0 ≤ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column coordinate of the reference point.
Default: 16
Suggested values: Column ∈ {0, 2, 4, 8, 16, 32, 128}
Value range: 0 ≤ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50, 70, 100, 200, 400}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let F be the area of an input region, $F1$ the area of the structuring element 1, and $F2$ the area of the structuring element 2. Then the runtime complexity for one object is:

$$O\left(\text{Iterations} \cdot \sqrt{F} \cdot \left(\sqrt{F1} + \sqrt{F2}\right)\right) .$$

Result

thickening returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

golay_elements, threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_struct_elements, gen_region_polygon_filled

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

thickening_golay, thickening_seq

See also

hit_or_miss

Module

Foundation

thickening_golay (Region : RegionThick : GolayElement, Rotation :)

Add the result of a hit-or-miss operation to a region (using a Golay structuring element).

thickening_golay is obsolete and is only provided for reasons of backward compatibility.

thickening_golay performs a thickening of the input regions using morphological operations and structuring elements from the Golay alphabet. The operator first applies a hit-or-miss-transformation to Region (cf.

hit_or_miss_golay), and then adds the detected points to the input region. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number Rotation determines which rotation of the element should be used. The Golay elements, together with all possible rotations, are described with the operator golay_elements.

Attention

Not all values of Rotation are valid for any Golay element.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **RegionThick** (output_object) ..... region(-array) ⇝ *object*
Result of the thickening operator.

▷ **GolayElement** (input_control) ..... string ⇝ *string*
Structuring element from the Golay alphabet.
Default: 'h'
List of values: GolayElement ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}

▷ **Rotation** (input_control) ..... integer ⇝ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default: 0
List of values: Rotation ∈ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(6 \cdot \sqrt{F}) \ .$$

Result

thickening_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

thickening, thickening_seq

See also

erosion_golay, hit_or_miss_golay

Module

Foundation

thickening_seq (Region : RegionThick : GolayElement, Iterations :)

Add the result of a hit-or-miss operation to a region (sequential).

thickening_seq is obsolete and is only provided for reasons of backward compatibility.

thickening_seq calculates the sequential thickening of the input regions with a structuring element from the Golay alphabet (GolayElement). To do so, thickening_seq calls the operator thickening_golay with all possible rotations of the structuring element Iterations times. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The Golay elements, together with all possible rotations, are described with the operator golay_elements. For all elements of the Golay alphabet, except for 'c', the foreground and background masks are exchanged in order to have an effect for them on the outer boundary of the region. The element 'c' can be used to generate the convex hull of the input region if enough iterations are performed.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Regions to be processed.
- ▷ **RegionThick** (output_object) ..... region(-array) ⇝ *object*
  Result of the thickening operator.
- ▷ **GolayElement** (input_control) ..... string ⇝ *string*
  Structuring element from the Golay alphabet.
  Default: 'h'
  List of values: GolayElement ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}
- ▷ **Iterations** (input_control) ..... integer ⇝ *integer*
  Number of iterations.
  Default: 1
  Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50, 70, 100, 200}
  Value range: 1 ≤ Iterations (lin)
  Minimum increment: 1
  Recommended increment: 1

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(Iterations \cdot 6 \cdot \sqrt{F}) \ .$$

Result

thickening_seq returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

thickening_golay, thickening

See also

erosion_golay, thinning_seq

Module

Foundation

```
thinning ( Region, StructElement1,
    StructElement2 : RegionThin : Row, Column, Iterations : )
```

Remove the result of a hit-or-miss operation from a region.

thinning is obsolete and is only provided for reasons of backward compatibility.

thinning performs a thinning of the input regions using morphological operations. The operator first applies a hit-or-miss-transformation to Region (cf. hit_or_miss), and then removes the detected points from the input region. The parameter Iterations determines the number of iterations performed.

For the choice of the structuring elements StructElement1 and StructElement2, as well as for Row and Column, the same restrictions described under hit_or_miss apply.

Structuring elements (StructElement1, StructElement2) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be processed.

▷ **StructElement1** (input_object) . . . . . region ⇝ *object*
Structuring element for the foreground.

▷ **StructElement2** (input_object) . . . . . region ⇝ *object*
Structuring element for the background.

▷ **RegionThin** (output_object) . . . . . region(-array) ⇝ *object*
Result of the thinning operator.

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row coordinate of the reference point.
Default: 0
Value range: $0 \leq$ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column coordinate of the reference point.
Default: 0
Value range: $0 \leq$ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Iterations** (input_control) . . . . . integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$
Value range: $1 \leq$ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let F be the area of an input region, $F1$ the area of the structuring element 1, and $F2$ the area of the structuring element 2. Then the runtime complexity for one object is:

$$O\left(\text{Iterations} \cdot \sqrt{F} \cdot \left(\sqrt{F1} + \sqrt{F2}\right)\right) .$$

Result

thinning returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_region_polygon_filled

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

thinning_golay, thinning_seq

See also

hit_or_miss

Module

Foundation

thinning_golay (Region : RegionThin : GolayElement, Rotation :)

Remove the result of a hit-or-miss operation from a region (using a Golay structuring element).

thinning_golay is obsolete and is only provided for reasons of backward compatibility.

thinning_golay performs a thinning of the input regions using morphological operations and structuring elements from the Golay alphabet. The operator first applies a hit-or-miss-transformation to Region (cf. hit_or_miss_golay), and then removes the detected points from the input region. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number Rotation determines which rotation of the element should be used. The Golay elements, together with all possible rotations, are described with the operator golay_elements.

Attention

Not all values of Rotation are valid for any Golay element.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **RegionThin** (output_object) ..... region(-array) ⇝ *object*
Result of the thinning operator.

▷ **GolayElement** (input_control) ..... string ⇝ *string*
Structuring element from the Golay alphabet.
Default: 'h'
List of values: GolayElement ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}

▷ **Rotation** (input_control) ..... integer ⇝ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default: 0
List of values: Rotation ∈ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(6 \cdot \sqrt{F}) \ .$$

Result

thinning_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

thinning_seq, thinning

See also

erosion_golay, hit_or_miss_golay

Module

Foundation

thinning_seq (Region : RegionThin : GolayElement, Iterations :)

Remove the result of a hit-or-miss operation from a region (sequential).

thinning_seq is obsolete and is only provided for reasons of backward compatibility.

thinning_seq calculates the sequential thinning of the input regions with a structuring element from the Golay alphabet (GolayElement). To do so, thinning_seq calls the operator thinning_golay with all possible rotations of the structuring element Iterations times. If Iterations is chosen large enough, the operator calculates the skeleton of a region if the structuring elements 'l' or 'm' are used. For the element 'c' the background and foreground are exchanged in order to have an effect on the interior boundary of a region. If a very large value or 'maximal' is passed for Iterations the iteration stops if no more changes occur. The following structuring elements are available:

'l' Skeleton, similar to skeleton. This structuring element is also used in morph_skiz.

'm' A skeleton with many "hairs" and multiple (parallel) branches.

'd' A skeleton without multiple branches, but with many gaps, similar to morph_skeleton.

'c' Uniform erosion of the region.

'e' One pixel wide lines are shortened. This structuring element is also used in morph_skiz.

'i' Isolated points are removed. (Only Iterations = 1 is useful.)

'f' Y-junctions are eliminated. (Only Iterations = 1 is useful.)

'f2' One pixel long branches and corners are removed. (Only Iterations = 1 is useful.)

'h' A kind of inner boundary, which, however, is thicker than the result of boundary, is generated. (Only Iterations = 1 is useful.)

'k' Junction points are eliminated, but also new ones are generated.

The Golay elements, together with all possible rotations, are described with the operator golay_elements.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be processed.

▷ **RegionThin** (output_object) . . . . . region(-array) ⇝ *object*
Result of the thinning operator.

▷ **GolayElement** (input_control) . . . . . string ⇝ *string*
Structuring element from the Golay alphabet.
Default: 'l'
List of values: `GolayElement` ∈ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}

▷ **Iterations** (input_control) . . . . . integer ⇝ *integer* / string
Number of iterations. For 'f', 'f2', 'h' and 'i' the only useful value is 1.
Default: 20
Suggested values: `Iterations` ∈ {'maximal', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50, 70, 100, 150, 200}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(Iterations \cdot 6 \cdot \sqrt{F}) \ .$$

Result

`thinning_seq` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_struct_elements`, `gen_region_polygon_filled`

Possible Successors

`pruning`, `reduce_domain`, `select_shape`, `area_center`, `connection`, `complement`

Alternatives

`skeleton`, `morph_skiz`, `expand_region`

See also

`hit_or_miss_seq`, `erosion_golay`, `difference`, `thinning_golay`, `thinning`, `thickening_seq`

Module

Foundation

17.12 OCR

close_ocr (: : OcrHandle :)

Deallocation of the memory of an OCR classifier.

close_ocr is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator close_ocr deallocates the memory of the classifier having the number OcrHandle. Hereby all corresponding data will be deleted. However, if necessary, they can be saved in advance using the operator write_ocr. The number OcrHandle will be invalid after the call; but later the system can use it again for new classifiers.

Attention

All data of the classifier will be deleted in main memory (not on the hard disk).

Parameters

▷ **OcrHandle** (input_control) ..... ocr_box ⇝ *handle*
ID of the OCR classifier to be deleted.

Result

If the parameter OcrHandle is valid, the operator close_ocr returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OcrHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

write_ocr_trainf

Possible Successors

read_ocr

Module

OCR/OCV

create_ocr_class_box (: : WidthPattern, HeightPattern, Interpolation, Features, Character : OcrHandle)

Create a new OCR-classifier.

create_ocr_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator create_ocr_class_box creates a new OCR classifier. For a description of this classifier see operator learn_class_box. This classifier must then be trained with the help of the operators traind_ocr_class_box or trainf_ocr_class_box.

The parameters WidthPattern and HeightPattern indicate the size of the input-layer of the network. This size is used for the features *'projection_horizontal'*, *'projection_vertical'*, *'pixel'*, *'pixel_invar'*, and *'pixel_binary'* to transform the character to a standard size. The bigger the standard size is, the more characters can be distinguished. Hereby the amount of time necessary for the training (as well as the number of training random samples) and the time necessary for the recognition, however, will increase as well. The parameter Interpolation indicates the interpolation mode concerning the adaptation of characters in the image to the network. For more detailed information on this parameter see also affine_trans_image. The value *0* results in the same interpolation as *'nearest_neighbor'* in affine_trans_image, i.e., no interpolation is performed. For *1*, the same behavior as *'constant'* in affine_trans_image is obtained, i.e., equally weighted interpolation between adjacent pixels is used. Finally, *2* results in the same interpolation as *'weighted'*, i.e., Gaussian interpolation between adjacent pixels is used. The parameter Interpolation must be chosen such that no aliasing occurs when the character

is scaled to the standard size. Typically, this means that Interpolation should be set to *1*, except in cases where the characters are scaled down by a large amount, in which case Interpolation = *2* should be chosen. Interpolation = *0* should only be chosen if the characters will not be scaled.

The parameter Character determines all the characters which have to be recognized. Normally the transmitted strings consist of one character (e.g., alphabet). But also strings of any length can be learned. The number of distinguishable characters (number of strings in Character) is limited to *2048*.

The parameter Features helps to chose additional features besides gray values in order to recognize characters.

The following features are available:

'default' *'ratio'* and *'pixel_invar'* are selected.

'ratio' Ratio of the character.

'width' Width of the character (not invariant to scaling).

'height' Height of the character (not invariant to scaling).

'zoom_factor' Difference in size between the current character and the values of WidthPattern and HeightPattern (not invariant to scaling).

'foreground' Relative number of pixels in the foreground.

'foreground_grid_9' Relative number of foreground pixels in a 3×3 grid within the surrounding rectangle of the character.

'foreground_grid_16' Relative number of foreground pixels in a 4×4 grid within the surrounding rectangle of the character.

'anisometry' Form feature anisometry.

'compactness' Form feature compactness.

'convexity' Form feature convexity.

'moments_region_2nd_invar' Normed 2nd geometric moments of the region. See also moments_region_2nd_invar.

'moments_region_2nd_rel_invar' Normed 2nd relative geometric moments of the region. See also moments_region_2nd_rel_invar.

'moments_region_3rd_invar' Normed 3rd geometric moments of the region. See also moments_region_3rd_invar.

'moments_central' Normed central geometric moments of the region. See also moments_region_central.

'phi' Sine and cosine of the orientation (angle) of the character.

'num_connect' Number of connecting components.

'num_holes' Number of holes.

'projection_horizontal' Horizontal projection of the gray values.

'projection_horizontal_invar' Horizontal projection of the gray values with are automatically scaled to maximum range.

'projection_vertical' Vertical projection of the gray values.

'projection_vertical_invar' Vertical projection of the gray values with are automatically scaled to maximum range.

'cooc' Values of the binary cooccurrence matrix.

'moments_gray_plane' Normed gray value moments and the angles of the gray value level.

'num_runs' Number of chords in the region normed to the height.

'chord_histo' Frequency of the chords per row (not scale-invariant).

'pixel' Gray value of the character.

'pixel_invar' Gray values of the character with automatic maximal scaling of the gray values.

'pixel_binary' Region of the character as a binary image zoomed to a size of WidthPattern $\times$ HeightPattern.

'gradient_8dir' Gradients are computed on the character image. The gradient directions are discretized into 8 directions. The amplitude image is decomposed into 8 channels according to these discretized directions. 25 samples on a 5×5 grid are extracted from each channel. These samples are used as features.

Parameters

▷ **WidthPattern** (input_control) . . . . . integer ⇝ *integer*
Width of the input layer of the network.
Default: 8
Suggested values: WidthPattern ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20}
Value range: 1 ≤ WidthPattern ≤ 100

▷ **HeightPattern** (input_control) . . . . . integer ⇝ *integer*
Height of the input layer of the network.
Default: 10
Suggested values: HeightPattern ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20}
Value range: 1 ≤ HeightPattern ≤ 100

▷ **Interpolation** (input_control) . . . . . integer ⇝ *integer*
Interpolation mode concerning scaling of characters.
Default: 1
List of values: Interpolation ∈ {0, 1, 2}

▷ **Features** (input_control) . . . . . string(-array) ⇝ *string*
Additional features.
Default: 'default'
List of values: Features ∈ {'default', 'zoom_factor', 'ratio', 'width', 'height', 'foreground', 'foreground_grid_9', 'foreground_grid_16', 'anisometry', 'compactness', 'convexity', 'moments_region_2nd_invar', 'moments_region_2nd_rel_invar', 'moments_region_3rd_invar', 'moments_central', 'phi', 'num_connect', 'num_holes', 'projection_horizontal', 'projection_vertical', 'projection_horizontal_invar', 'projection_vertical_invar', 'chord_histo', 'num_runs', 'pixel', 'pixel_invar', 'pixel_binary', 'gradient_8dir', 'cooc', 'moments_gray_plane'}

▷ **Character** (input_control) . . . . . string-array ⇝ *string*
All characters of a set.
Default: ['a','b','c']

▷ **OcrHandle** (output_control) . . . . . ocr_box ⇝ *handle*
ID of the created OCR classifier.

Result

If the parameters are correct, the operator create_ocr_class_box returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

reset_obj_db

Possible Successors

traind_ocr_class_box, trainf_ocr_class_box, info_ocr_class_box, write_ocr, ocr_change_char

Alternatives

create_ocr_class_mlp, create_ocr_class_svm

See also

affine_trans_image, ocr_change_char, moments_region_2nd_invar, moments_region_2nd_rel_invar, moments_region_3rd_invar, moments_region_central

Module

OCR/OCV

create_text_model (: : : TextModel)

Create a text model.

create_text_model is obsolete and is only provided for reasons of backward compatibility. New applications should use create_text_model_reader instead.

create_text_model creates a new TextModel which describes the text to be segmented by find_text. Set and query Parameters of TextModel via set_text_model_param and get_text_model_param.

Calls of create_text_model are equivalent to (and should be replaced by) calls of create_text_model_reader with Mode set to *'manual'*.

Parameters

▷ **TextModel** (output_control) ..... text_model ⇝ *handle*
New text model.

Result

create_text_model returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Module

OCR/OCV

deserialize_ocr (: : SerializedItemHandle : OcrHandle)

Deserialize a serialized OCR classifier.

deserialize_ocr is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

deserialize_ocr deserializes an OCR classifier, that was serialized by serialize_ocr (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized OCR classifier is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created OCR classifier with the handle OcrHandle.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **OcrHandle** (output_control) ..... ocr_box ⇝ *handle*
ID of the OCR classifier.

Result

If the parameters are valid, the operator deserialize_ocr returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

reset_obj_db, fread_serialized_item, receive_serialized_item, serialize_ocr

Possible Successors

do_ocr_multi, do_ocr_single, traind_ocr_class_box, trainf_ocr_class_box

See also

serialize_ocr, do_ocr_multi, traind_ocr_class_box, trainf_ocr_class_box

Module

OCR/OCV

do_ocr_multi (Character, Image : : OcrHandle : Class, Confidence)

Classify characters.

do_ocr_multi is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator do_ocr_multi assigns a class to every Character (character). For gray value features the gray values from the surrounding rectangles of the regions are used. The gray values will be taken from the parameter Image. For each character the corresponding class will be returned in Class and a confidence value will be returned in Confidence. The confidence value indicates the similarity between the input pattern and the assigned character.

Parameters

- ▷ **Character** (input_object) ..... region(-array) ⇝ *object*
  Characters to be recognized.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object* : byte / uint2
  Gray values for the characters.
- ▷ **OcrHandle** (input_control) ..... ocr_box ⇝ *handle*
  ID of the OCR classifier.
- ▷ **Class** (output_control) ..... string(-array) ⇝ *string*
  Class (name) of the characters.
  Number of elements: Class == Character
- ▷ **Confidence** (output_control) ..... real(-array) ⇝ *real*
  Confidence values of the characters.
  Number of elements: Confidence == Character

Result

If the input parameters are correct, the operator do_ocr_single returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

traind_ocr_class_box, trainf_ocr_class_box, read_ocr, connection, sort_region

Alternatives

do_ocr_single

See also

write_ocr

Module

OCR/OCV

do_ocr_single (Character, Image : : OcrHandle : Classes, Confidences)

Classify one character.

do_ocr_single is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator do_ocr_single assigns classes to the Character (characters). For gray value features gray values of the surrounding rectangles of the regions will be used. The gray values will be taken from the parameter Image. For each character the two classes with the highest confidences will be returned in Classes. The corresponding confidences will be returned in Confidences. The confidence value indicates the similarity between the input pattern and the assigned character.

Parameters

- ▷ **Character** (input_object) ..... region ⇝ *object*
  Character to be recognized.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
  Gray values of the characters.
- ▷ **OcrHandle** (input_control) ..... ocr_box ⇝ *handle*
  ID of the OCR classifier.
- ▷ **Classes** (output_control) ..... string-array ⇝ *string*
  Classes (names) of the characters.
  Number of elements: 2
- ▷ **Confidences** (output_control) ..... real-array ⇝ *real*
  Confidence values of the characters.
  Number of elements: 2

Result

If the input parameters are correct, the operator do_ocr_single returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

traind_ocr_class_box, trainf_ocr_class_box, read_ocr, connection, sort_region

Alternatives

do_ocr_multi

See also

write_ocr

Module

OCR/OCV

info_ocr_class_box (: : OcrHandle : WidthPattern, HeightPattern, Interpolation, WidthMaxChar, HeightMaxChar, Features, Characters)

Get information about an OCR classifier.

info_ocr_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator info_ocr_class_box returns some information about an OCR classifier. The parameters are equivalent to those of create_ocr_class_box. The parameters WidthMaxChar and HeightMaxChar indicate the extension of the largest trained character. These values can be used to control the segmentation.

Parameters

▷ **OcrHandle** (input_control) . . . . . ocr_box ⇝ *handle*
ID of the OCR classifier.

▷ **WidthPattern** (output_control) . . . . . integer ⇝ *integer*
Width of the scaled characters.

▷ **HeightPattern** (output_control) . . . . . integer ⇝ *integer*
Height of the scaled characters.

▷ **Interpolation** (output_control) . . . . . integer ⇝ *integer*
Interpolation mode for scaling the characters.

▷ **WidthMaxChar** (output_control) . . . . . integer ⇝ *integer*
Width of the largest trained character.

▷ **HeightMaxChar** (output_control) . . . . . integer ⇝ *integer*
Height of the largest trained character.

▷ **Features** (output_control) . . . . . string-array ⇝ *string*
Used features.

▷ **Characters** (output_control) . . . . . string-array ⇝ *string*
All characters of the set.

Result

The operator `info_ocr_class_box` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_ocr`, `create_ocr_class_box`

Possible Successors

`write_ocr`

Module

OCR/OCV

ocr_change_char (: : OcrHandle, Character :)

Define a new conversion table for the characters.

ocr_change_char is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator `ocr_change_char` establishes a new look-up table for the characters. Hereby the number of strings of `Character` must be the same as of the classifier `OcrHandle`. In order to enlarge the font, the operator `ocr_change_char` may be used as follows: More characters than actually needed will be indicated when creating a network using (`create_ocr_class_box`). The last n characters will not be used so far. If more characters are needed at a later stage, these unused characters will be allocated and then trained with the help of the operator `ocr_change_char`.

Parameters

▷ **OcrHandle** (input_control) . . . . . ocr_box ⇝ *handle*
ID of the OCR-network to be changed.

▷ **Character** (input_control) . . . . . string-array ⇝ *string*
New assign of characters.
Default: ['a','b','c']

Result

If the number of characters in `Character` is identical with the number of the characters of the network, the operator `ocr_change_char` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OcrHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

read_ocr

Possible Successors

do_ocr_multi_class_mlp, do_ocr_single_class_mlp

Module

OCR/OCV

ocr_get_features (Character : : OcrHandle : FeatureVector)

Access the features which correspond to a character.

ocr_get_features is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator ocr_get_features calculates the features for the given Character. The type and number of features is determined by the classifier OcrHandle. FeatureVector contains the same values which are used inside operators like traind_ocr_class_box or trainf_ocr_class_box.

Parameters

▷ **Character** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Characters to be trained.

▷ **OcrHandle** (input_control) . . . . . ocr_box ⇝ *handle*
ID of the desired OCR-classifier.

▷ **FeatureVector** (output_control) . . . . . real-array ⇝ *real*
Feature vector.

Result

If the parameters are correct, the operator ocr_get_features returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_ocr, reduce_domain, threshold, connection

See also

trainf_ocr_class_box, traind_ocr_class_box

Module

OCR/OCV

read_ocr (: : FileName : OcrHandle)

Read an OCR classifier from a file.

read_ocr is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator read_ocr reads an OCR classifier from a file FileName. This file will hereby be searched in the directory ($HALCONROOT/ocr/) as well as in the currently used directory. If too many classifiers have been loaded, an error message will be displayed.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Name of the OCR classifier file.
Default: 'testnet'
File extension: .obc, .fnt

▷ **OcrHandle** (output_control) . . . . . ocr_box ⇝ *handle*
ID of the read OCR classifier.

Result

If the indicated file is available and the format is correct, the operator read_ocr returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

reset_obj_db

Possible Successors

do_ocr_multi, do_ocr_single, traind_ocr_class_box, trainf_ocr_class_box

See also

write_ocr, do_ocr_multi, traind_ocr_class_box, trainf_ocr_class_box

Module

OCR/OCV

serialize_ocr (: : OcrHandle : SerializedItemHandle)

Serialize an OCR classifier.

serialize_ocr is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

serialize_ocr serializes an OCR classifier (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_ocr is converted to a serialized item. The OCR classifier is defined by the handle OcrHandle. The serialized OCR classifier is returned by the handle SerializedItemHandle and can be deserialized by deserialize_ocr.

Parameters

▷ **OcrHandle** (input_control) . . . . . ocr_box ⇝ *handle*
ID of the OCR classifier.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_ocr` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`traind_ocr_class_box`, `trainf_ocr_class_box`

Possible Successors

`do_ocr_multi`, `do_ocr_single`, `fwrite_serialized_item`, `send_serialized_item`, `deserialize_ocr`

See also

`deserialize_ocr`, `do_ocr_multi`, `traind_ocr_class_box`, `trainf_ocr_class_box`

Module

OCR/OCV

testd_ocr_class_box (Character, Image : : OcrHandle, Class : Confidence)

Test an OCR classifier.

testd_ocr_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator `testd_ocr_class_box` tests the confidence with which a character belongs to a given class. Any number of regions of an image can be passed. For each character (region) in `Character` the corresponding name (class) `Class` must be specified. The gray values are passed in `Image`. When the operator has finished the parameter `Confidence` provides information about how sure a character belongs to the (arbitrary chosen) class.

Parameters

▷ **Character** (input_object) . . . . . region(-array) ⇝ *object*
Characters to be tested.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Gray values for the characters.

▷ **OcrHandle** (input_control) . . . . . ocr_box ⇝ *handle*
ID of the desired OCR-classifier.

▷ **Class** (input_control) . . . . . string(-array) ⇝ *string*
Class (name) of the characters.
Default: 'a'

▷ **Confidence** (output_control) . . . . . real(-array) ⇝ *real*
Confidence for the character to belong to the class.

Result

If the parameters are correct, the operator `testd_ocr_class_box` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_ocr, trainf_ocr_class_box, traind_ocr_class_box

Module

OCR/OCV

traind_ocr_class_box (Character, Image : : OcrHandle, Class : AvgConfidence)

Train an OCR classifier by the input of regions.

traind_ocr_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator traind_ocr_class_box trains the classifier directly via the input of regions in an image. Any number of regions of an image can be passed. For each character (region) in Character the corresponding name (class) Class must be specified. The gray values are passed in Image. When the procedure has finished the parameter AvgConfidence provides information about the success of the training: It contains the average confidence of the trained characters measured by a re-classification. The confidence of mismatched characters is set to 0 (thus, the average confidence will be decreased significantly).

Parameters

- ▷ **Character** (input_object) . . . . . region(-array) ⇝ *object*
  Characters to be trained.
- ▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Gray values for the characters.
- ▷ **OcrHandle** (input_control) . . . . . ocr_box ⇝ *handle*
  ID of the desired OCR-classifier.
- ▷ **Class** (input_control) . . . . . string(-array) ⇝ *string*
  Class (name) of the characters.
  Default: 'a'
- ▷ **AvgConfidence** (output_control) . . . . . real ⇝ *real*
  Average confidence during a re-classification of the trained characters.

Result

If the parameters are correct, the operator traind_ocr_class_box returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OcrHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_ocr_class_box, read_ocr

Possible Successors

write_ocr, do_ocr_multi, do_ocr_single

Alternatives

trainf_ocr_class_box

Module

OCR/OCV

trainf_ocr_class_box (: : OcrHandle, TrainingFile : AvgConfidence)

Train an OCR classifier with the help of a training file.

trainf_ocr_class_box is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator trainf_ocr_class_box trains the classifier OcrHandle via the indicated training files. Any number of files can be indicated. The parameter AvgConfidence provides information about the success of the training: It contains the average confidence of the trained characters measured by a re-classification. The confidence of mismatched characters is set to 0 (thus, the average confidence will be decreased significantly). Please, note that training characters that have no corresponding class in the classifier OcrHandle are discarded.

Attention

The names of the characters in the file must fit the network.

Parameters

- ▷ **OcrHandle** (input_control) ..... ocr_box ⇝ *handle*
  ID of the desired OCR-network.
- ▷ **TrainingFile** (input_control) ..... filename.read(-array) ⇝ *string*
  Names of the training files.
  Default: 'train_ocr'
  File extension: .trf, .otr
- ▷ **AvgConfidence** (output_control) ..... real ⇝ *real*
  Average confidence during a re-classification of the trained characters.

Result

If the file name is correct and the data fit the network, the operator trainf_ocr_class_box returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OcrHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_ocr_class_box, read_ocr

Possible Successors

traind_ocr_class_box, write_ocr, do_ocr_multi, do_ocr_single

Alternatives

traind_ocr_class_box

Module

OCR/OCV

write_ocr (: : OcrHandle, FileName :)

Writing an OCR classifier into a file.

write_ocr is obsolete and is only provided for reasons of backward compatibility. New applications should use the MLP, SVM or CNN based operators instead.

The operator write_ocr writes the OCR classifier OcrHandle into the file FileName. Since the data of the classifier will be lost when the program is finished, they have to be stored after the training if the user wants to use them again at a later execution of the program. The data can then be read with the help of the operator read_ocr. The extension will be added automatically to the parameter FileName.

Attention

The output file FileName must be given without extension.

Parameters

▷ **OcrHandle** (input_control) . . . . . ocr_box ⇝ *handle*
ID of the OCR classifier.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the file for the OCR classifier (without extension).
Default: 'my_ocr'
File extension: .obc

Result

If the parameter OcrHandle is valid and the indicated file can be written, the operator write_ocr returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

traind_ocr_class_box, trainf_ocr_class_box

Possible Successors

do_ocr_multi, do_ocr_single

See also

read_ocr, do_ocr_multi, traind_ocr_class_box, trainf_ocr_class_box

Module

OCR/OCV

17.13 Regions

get_region_chain (Region : : : Row, Column, Chain)

Contour of an object as chain code.

get_region_chain is obsolete and is only provided for reasons of backward compatibility.

The operator get_region_chain returns the contour of a region. A contour is a series of pixels describing the outline of the region. The contour "lies on" the region. It starts at the smallest line number; in that line at the pixel with the largest column index. The rotation occurs clockwise. Holes of the region are ignored. The direction code (chain code) is defined as follows:

$$\begin{matrix} 3 & 2 & 1 \\ 4 & * & 0 \\ 5 & 6 & 7 \end{matrix}$$

The operator get_region_chain returns the code in the form of a tuple. In case of an empty region the parameters Row and Column are zero and Chain is the empty tuple.

Attention

Holes of the region are ignored. Only one region may be passed, and it must have exactly one connection component.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
Region to be transformed.

▷ **Row** (output_control) . . . . . chain.begin.y ⇝ *integer*
Line of starting point.

▷ **Column** (output_control) . . . . . chain.begin.x ⇝ *integer*
Column of starting point.

▷ **Chain** (output_control) . . . . . chain.code-array ⇝ *integer*
Direction code of the contour (from starting point).
Value range: 0 ≤ `Chain` ≤ 7

Result

The operator `get_region_chain` normally returns the value `2` (`H_MSG_TRUE`). If more than one connection component is passed an exception is raised. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via the operator `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`sobel_amp`, `threshold`, `skeleton`, `edges_image`, `gen_rectangle1`, `gen_circle`

Possible Successors

`approx_chain`, `approx_chain_simple`

See also

`copy_obj`, `get_region_contour`, `get_region_polygon`

Module

Foundation

`hamming_change_region` `( InputRegion : OutputRegion : Width, Height, Distance : )`

Generate a region having a given Hamming distance.

`hamming_change_region` is obsolete and is only provided for reasons of backward compatibility.

`hamming_change_region` changes the region in the left upper part of the image given by `Width` and `Height` such that the resulting regions have a Hamming distance of `Distance` to the input regions. This is done by adding or removing `Distance` points from the input region.

Attention

If `Width` and `Height` are chosen too large the resulting region requires a lot of memory.

Parameters

▷ **InputRegion** (input_object) . . . . . region(-array) ⇝ *object*
Region to be modified.

▷ **OutputRegion** (output_object) . . . . . region(-array) ⇝ *object*
Regions having the required Hamming distance.

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the region to be changed.
Default: 100
Suggested values: `Width` ∈ {64, 128, 256, 512}
Value range: 1 ≤ `Width` ≤ 512 (lin)
Minimum increment: 1
Recommended increment: 10
Restriction: Width > 0

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the region to be changed.
Default: 100
Suggested values: Height ∈ {64, 128, 256, 512}
Value range: 1 ≤ Height ≤ 512 (lin)
Minimum increment: 1
Recommended increment: 10
Restriction: Height > 0

▷ **Distance** (input_control) . . . . . integer ⇝ *integer*
Hamming distance between the old and new regions.
Default: 1000
Suggested values: Distance ∈ {100, 500, 1000, 5000, 10000}
Value range: 0 ≤ Distance ≤ 10000 (lin)
Minimum increment: 1
Recommended increment: 10
Restriction: Distance >= 0 && Distance < Width * Height

Complexity

Memory requirement of the generated region (worst case): $O(2 * \texttt{Width} * \texttt{Height})$.

Result

hamming_change_region returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

connection, regiongrowing, pouring, class_ndim_norm

Possible Successors

select_shape

See also

hamming_distance

Module

Foundation

interjacent (Region : RegionInterjacent : Mode :)

Partition the image plane using given regions.

interjacent is obsolete and is only provided for reasons of backward compatibility.

interjacent partitions the image plane using the regions given in Region. The result is a region containing the extracted separating lines. The following modes of operation can be used:

'medial_axis' This mode is used for regions that do not touch or overlap. The operator will find separating lines between the regions which partition the background evenly between the input regions. This corresponds to the following calls:

Example:

```
complement('full',Region,Tmp)
skeleton(Tmp,Result)
```

'border' If the input regions do not touch or overlap this mode is equivalent to boundary(Region,Result), i.e., it replaces each region by its boundary. If regions are touching they are aggregated into one region. The corresponding output region then contains the boundary of the aggregated region, as well as the one pixel wide separating line between the original regions. This corresponds to the following calls:

Example:

```
boundary(Region,Tmp1,'inner')
union1(Tmp1,Tmp2)
skeleton(Tmp2,Result)
```

'mixed' In this mode the operator behaves like the mode *'medial_axis'* for non-overlapping regions. If regions touch or overlap, again separating lines between the input regions are generated on output, but this time including the "touching line" between regions, i.e., touching regions are separated by a line in the output region. This corresponds to the following calls:

Example:

```
erosion1(Region,Mask,Tmp1,1)
union1(Tmp1,Tmp2)
complement(full,Tmp2,Tmp3)
skeleton(Tmp3,Result)
```

where Mask denotes the following "cross mask":

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions for which the separating lines are to be determined.

▷ **RegionInterjacent** (output_object) ..... region ⇝ *object*
Output region containing the separating lines.

▷ **Mode** (input_control) ..... string ⇝ *string*
Mode of operation.
Default: 'mixed'
List of values: Mode ∈ {'medial_axis', 'border', 'mixed'}

Example

```
read_image(Image,'forest_air1')
mean_image(Image,Mean,31,31)
dyn_threshold(Image,Mean,Seg,20,'light')
interjacent(Seg,Graph,'medial_axis')
dev_display(Graph)
```

Result

interjacent always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>), the behavior in case of an empty input region via set_system('empty_region_result',<Result>), and the behavior in case of an empty result region via set_system('store_empty_region',<'true'/'false'>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

See also

expand_region, junctions_skeleton, boundary

Module

Foundation

17.14 Segmentation

bin_threshold (Image : Region : :)

Segment an image using an automatically determined threshold.

bin_threshold is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.05. New applications should use the operator binary_threshold instead.

bin_threshold segments a single-channel gray value image using an automatically determined threshold. First, the relative histogram of the gray values is determined. Then, relevant minima are extracted from the histogram, which are used as parameters for a thresholding operation. In order to reduce the number of minima, the histogram is smoothed with a Gaussian, as in auto_threshold. The mask size is enlarged until there is only one minimum in the smoothed histogram. The selected region contains the pixels with gray values from 0 to the minimum or for real images from the smallest value to the respective minimum. This operator is, for example useful for the segmentation of dark characters on a light paper.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **Region** (output_object) ..... region(-array) ⇝ *object*
  Dark regions of the image.

Example

```
read_image (Image, 'letters')
bin_threshold (Image, Seg)
connection (Seg, Connected)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Module

Foundation

class_ndim_box (MultiChannelImage : Regions : ClassifHandle :)

Classify pixels using hyper-cuboids.

class_ndim_box is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11.

class_ndim_box classifies the pixels of the multi-channel image given in MultiChannelImage. To do so, the classificator ClassifHandle created with create_class_box is used. The classificator can be trained using learn_ndim_box or as described with create_class_box. More information on the structure of the classificator can be found also under that operator.

MultiChannelImage is a multi channel image. Its pixel values are used for the classification.

Parameters

- ▷ **MultiChannelImage** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / int4 / real
  Multi channel input image.
- ▷ **Regions** (output_object) ..... region-array ⇝ *object*
  Classification result.
- ▷ **ClassifHandle** (input_control) ..... class_box ⇝ *handle*
  Handle of the classifier.

Example

```
read_image(Image,'montery')
dev_display(Image)
dev_set_color('green')
disp_message (WindowHandle, 'Draw learning region  ',\
              'window', 12, 12, 'black', 'true')
draw_region(Reg1,WindowHandle)
reduce_domain(Image,Reg1,Foreground)
dev_set_color('red')
disp_message (WindowHandle, 'Draw background region',\
              'window', 12, 12, 'black', 'true')
draw_region(Reg2,WindowHandle)
reduce_domain(Image,Reg2,Background)
disp_message (WindowHandle, 'Training...           ',\
              'window', 12, 12, 'black', 'true')
create_class_box(ClassifHandle)
learn_ndim_box(Foreground,Background,Image,ClassifHandle)
disp_message (WindowHandle, 'Classification        ',\
              'window', 12, 12, 'black', 'true')
class_ndim_box(Image,Res,ClassifHandle)
dev_set_draw('fill')
dev_display (Res)
close_class_box(ClassifHandle)
```

Complexity

Let N be the number of hyper-cuboids and A be the area of the input region. Then the runtime complexity is $O(N, A)$.

Result

`class_ndim_box` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_class_box`, `learn_class_box`, `median_image`, `compose2`, `compose3`, `compose4`, `compose5`, `compose6`, `compose7`

Alternatives

`class_ndim_norm`, `class_2dim_sup`, `class_2dim_unsup`

See also

`descript_class_box`

Module

Foundation

`expand_line` (Image : RegionExpand : Coordinate, ExpandType, RowColumn, Threshold :)

Expand a region starting at a given line.

`expand_line` is obsolete and is only provided for reasons of backward compatibility.

expand_line generates a region by expansion, starting at a given line (row or column). The expansion is terminated when the current gray value differs by more than Threshold from the mean gray value along the line (ExpandType = *'mean'*) or from the previously added gray value (ExpandType = *'gradient'*).

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object : byte*
Input image.

▷ **RegionExpand** (output_object) ..... region(-array) ⇝ *object*
Extracted segments.

▷ **Coordinate** (input_control) ..... integer ⇝ *integer*
Row or column coordinate.
Default: 256
Suggested values: Coordinate ∈ {16, 64, 128, 200, 256, 300, 400, 511}
Restriction: Coordinate >= 0

▷ **ExpandType** (input_control) ..... string ⇝ *string*
Stopping criterion.
Default: 'gradient'
List of values: ExpandType ∈ {'gradient', 'mean'}

▷ **RowColumn** (input_control) ..... string ⇝ *string*
Segmentation mode (row or column).
Default: 'row'
List of values: RowColumn ∈ {'row', 'column'}

▷ **Threshold** (input_control) ..... number ⇝ *real / integer*
Threshold for the expansion.
Default: 3.0
Suggested values: Threshold ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 13.0, 17.0, 20.0, 30.0}
Value range: 0 ≤ Threshold ≤ 255 (lin)
Minimum increment: 1.0
Recommended increment: 1.0

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  HImage   image (argv[1]),
           gauss;
  HWindow  win;

  win.SetDraw ("margin");
  win.SetColored (12);

  image.Display (win);

  gauss = image.GaussImage (5);

  HRegionArray reg = gauss.ExpandLine (100, "mean", "row", 5.0);

  reg.Display (win);
  win.Click ();

  return (0);
}
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image, anisotropic_diffusion, median_image, affine_trans_image, rotate_image

Possible Successors

intersection, opening, closing

Alternatives

regiongrowing_mean, expand_gray, expand_gray_ref

Module

Foundation

learn_ndim_box (Foreground, Background, MultiChannelImage : : ClassifHandle :)

Train a classificator using a multi-channel image.

learn_ndim_box is obsolete and is only provided for reasons of backward compatibility. The operator will be removed with HALCON 25.11.

learn_ndim_box trains the classificator ClassifHandle with the gray values of MultiChannelImage using the points in Foreground as training sample. The points in Background are to be rejected by the classificator. The classificator trained thus can be used in class_ndim_box to segment multi-channel images. Foreground are the points that should be found, Background contains the points that should not be found.

Each pixel is trained once during the training process. For points in Foreground the class "0" is used, while for Background "1" is used. Pixels are trained by alternating points from Foreground with points from Background. If one region is smaller than the other, pixels are taken cyclically from the smaller region until the larger region is exhausted. learn_ndim_box later accepts only points that can be classified into class "0".

From a user's point of view the key difference between learn_ndim_norm and learn_ndim_box is that in the latter case the rejection class affects the classification process itself. Here, a hyper plane is generated that separates Foreground and Background classes, so that no points in feature space are classified incorrectly. As for learn_ndim_norm, however, an overlap between Foreground and Background class is allowed. This has its effect on the return value Quality. The larger the overlap, the smaller this value.

Attention

All channels must be of the same type.

Parameters

▷ **Foreground** (input_object) ..... region(-array) ⇝ *object*
Foreground pixels to be trained.

▷ **Background** (input_object) ..... region(-array) ⇝ *object*
Background pixels to be trained (rejection class).

▷ **MultiChannelImage** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / int4 / real
Multi-channel training image.

▷ **ClassifHandle** (input_control) ..... class_box ⇝ *handle*
Handle of the classifier.

Complexity

Let N be the number of generated hyper-cuboids and A be the area of the larger input region. Then the runtime complexity is $O(N * A)$.

Result

learn_ndim_box returns 2 (H_MSG_TRUE) if all parameters are correct and there is an active classificator. The

behavior with respect to the input images can be determined by setting the values of the flags *'no_object_result'* and *'empty_region_result'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ClassifHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_box, draw_region

Possible Successors

class_ndim_box, descript_class_box

Alternatives

learn_class_box, learn_ndim_norm

Module

Foundation

17.15 Tools

```
approx_chain ( : : Row, Column, MinWidthCoord, MaxWidthCoord,
    ThreshStart, ThreshEnd, ThreshStep, MinWidthSmooth,
    MaxWidthSmooth, MinWidthCurve, MaxWidthCurve, Weight1, Weight2,
    Weight3 : ArcCenterRow, ArcCenterCol, ArcAngle, ArcBeginRow,
    ArcBeginCol, LineBeginRow, LineBeginCol, LineEndRow, LineEndCol,
    Order )
```

Approximate a contour by arcs and lines.

approx_chain is obsolete and is only provided for reasons of backward compatibility.

The coordinates of a curve are approximated by a row of lines and arcs. The procedure tries values from a user-definable range for certain parameters. The limits of these ranges are explicitly stated in the parameter list of the function (MinWidthCoord ... MaxWidthCoord, ThreshStart ... ThreshEnd, MinWidthSmooth ... MaxWidthSmooth, MinWidthCurve ... MaxWidthCurve). Additionally, the step width for the parameter area of the threshold value for pointed corners has to be indicated (ThreshStep). By narrowing the covered areas the runtime of the calculation can be shortened, but the result may deteriorate.

The parameters Weight1, Weight2 and Weight3 indicate whether the desired weighting is placed more on precision of the approximation, obtaining as much large segments as possible or as few small segments as possible. Thus, for (Weight1,Weight2,Weight3) (1,0,0) creates a very precise approximation and (0,1,1) an approximation with as few large segments as possible.

The result of the procedure is returned separately as arcs and lines. If one is interested in the sequence of the segments the individual resulting elements can be read successively from the resulting tuples; the sequence can be taken from the return parameter order (0: next element is next line segment, 1: next element is next arc segment).

Attention

Contours which can possibly consist of only one segment should also be examined with a threshold maximum (ThreshEnd) > 1.0, because otherwise at least one “corner point” is determined in any case.

Parameters

▷ **Row** (input_control) . . . . . point.y-array ⇝ *integer*
Row of the contour.
Default: 32

▷ **Column** (input_control) . . . . . point.x-array ⇝ *integer*
Column of the contour.
Default: 32

▷ **MinWidthCoord** (input_control) . . . . . real ⇝ *real*
Minimum width of Gauss operator for coordinate smoothing (> 0.4).
Default: 0.5
Suggested values: MinWidthCoord ∈ {0.5, 0.7, 1.0, 1.2, 1.5, 1.7}
Value range: $0.4 \leq$ MinWidthCoord ≤ 3.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **MaxWidthCoord** (input_control) . . . . . real ⇝ *real*
Maximum width of Gauss operator for coordinate smoothing (> 0.4).
Default: 2.4
Suggested values: MaxWidthCoord ∈ {0.5, 0.7, 1.0, 1.2, 1.5, 1.7}
Value range: $0.4 \leq$ MaxWidthCoord ≤ 3.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **ThreshStart** (input_control) . . . . . real ⇝ *real*
Minimum threshold value of the curvature for accepting a corner (relative to the largest curvature present).
Default: 0.3
Suggested values: ThreshStart ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8}
Value range: $0.1 \leq$ ThreshStart ≤ 0.9 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **ThreshEnd** (input_control) . . . . . real ⇝ *real*
Maximum threshold value of the curvature for accepting a corner (relative to the largest curvature present).
Default: 0.9
Suggested values: ThreshEnd ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8}
Value range: $0.1 \leq$ ThreshEnd ≤ 0.9 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **ThreshStep** (input_control) . . . . . real ⇝ *real*
Step width for threshold increase.
Default: 0.2
Suggested values: ThreshStep ∈ {0.3, 0.4, 0.5}
Value range: $0.1 \leq$ ThreshStep ≤ 0.9 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **MinWidthSmooth** (input_control) . . . . . real ⇝ *real*
Minimum width of Gauss operator for smoothing the curvature function (> 0.4).
Default: 0.5
Suggested values: MinWidthSmooth ∈ {0.5, 0.7, 1.0, 1.2, 1.5, 1.7}
Value range: $0.4 \leq$ MinWidthSmooth ≤ 3.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **MaxWidthSmooth** (input_control) . . . . . real ⇝ *real*
Maximum width of Gauss operator for smoothing the curvature function.
Default: 2.4
Suggested values: MaxWidthSmooth ∈ {0.5, 0.7, 1.0, 1.2, 1.5, 1.7}
Value range: $0.4 \leq$ MaxWidthSmooth ≤ 3.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **MinWidthCurve** (input_control) . . . . . integer ⇝ *integer*
Minimum width of curve area for curvature determination (> 0.4).
Default: 2
Suggested values: MinWidthCurve ∈ {2, 5, 7}
Value range: $1 \leq$ MinWidthCurve ≤ 12 (lin)
Minimum increment: 1
Recommended increment: 2

▷ **MaxWidthCurve** (input_control) . . . . . integer ⇝ *integer*
Maximum width of curve area for curvature determination.
Default: 12
Suggested values: MaxWidthCurve ∈ {2, 5, 7}
Value range: $1 \leq$ MaxWidthCurve ≤ 20 (lin)
Minimum increment: 1
Recommended increment: 2

▷ **Weight1** (input_control) . . . . . real ⇝ *real*
Weighting factor for approximation precision.
Default: 1.0
Suggested values: Weight1 ∈ {0.0, 0.5, 1.0}
Value range: $0.0 \leq$ Weight1 ≤ 1.0 (lin)
Minimum increment: 0.1
Recommended increment: 0.5

▷ **Weight2** (input_control) . . . . . real ⇝ *real*
Weighting factor for large segments.
Default: 1.0
Suggested values: Weight2 ∈ {0.0, 0.5, 1.0}
Value range: $0.0 \leq$ Weight2 ≤ 1.0 (lin)
Minimum increment: 0.1
Recommended increment: 0.5

▷ **Weight3** (input_control) . . . . . real ⇝ *real*
Weighting factor for small segments.
Default: 1.0
Suggested values: Weight3 ∈ {0.0, 0.5, 1.0}
Value range: $0.0 \leq$ Weight3 ≤ 1.0 (lin)
Minimum increment: 0.1
Recommended increment: 0.5

▷ **ArcCenterRow** (output_control) . . . . . arc.center.y-array ⇝ *integer*
Row of the center of an arc.

▷ **ArcCenterCol** (output_control) . . . . . arc.center.x-array ⇝ *integer*
Column of the center of an arc.

▷ **ArcAngle** (output_control) . . . . . arc.angle.rad-array ⇝ *real*
Angle of an arc.

▷ **ArcBeginRow** (output_control) . . . . . arc.begin.y-array ⇝ *integer*
Row of the starting point of an arc.

▷ **ArcBeginCol** (output_control) . . . . . arc.begin.x-array ⇝ *integer*
Column of the starting point of an arc.

▷ **LineBeginRow** (output_control) . . . . . line.begin.y-array ⇝ *integer*
Row of the starting point of a line segment.

▷ **LineBeginCol** (output_control) . . . . . line.begin.x-array ⇝ *integer*
Column of the starting point of a line segment.

▷ **LineEndRow** (output_control) . . . . . line.end.y-array ⇝ *integer*
Row of the ending point of a line segment.

▷ **LineEndCol** (output_control) . . . . . line.end.x-array ⇝ *integer*
Column of the ending point of a line segment.

▷ **Order** (output_control) . . . . . integer-array ⇝ *integer*
Sequence of line (value 0) and arc segments (value 1).

Example

```
/* read edge image  */
read_image(&Image,"fig1_kan");
/* construct edge region  */
hysteresis_threshold(Image,&RK1,64,255,40,1);
connection(RK1,&Rand);
/* fetch chain code  */
T_get_region_contour(Rand,&Rows,&Columns);
firstline = get_i(Tline,0);
firstcol = get_i(Tcol,0);
/* approximation with lines and circular arcs */
set_d(t1,0.4,0);
set_d(t2,2.4,0);

set_d(t3,0.3,0);
set_d(t4,0.9,0);

set_d(t5,0.2,0);

set_d(t6,0.4,0);
set_d(t7,2.4,0);

set_i(t8,2,0);
set_i(t9,12,0);

set_d(t10,1.0,0);
set_d(t11,1.0,0);
set_d(t12,1.0,0);

T_approx_chain(Rows,Columns,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,
                &Bzl,&Bzc,&Br,&Bwl,&Bwc,&Ll0,&Lc0,&Ll1,&Lc1,&order);
nob = length_tuple(Bzl);
nol = length_tuple(Ll0);
/* draw lines and arcs */
set_i(WindowHandleTuple,WindowHandle,0);
set_line_width(WindowHandle,4);
if (nob>0) T_disp_arc(Bzl,Bzc,Br,Bwl,Bwc);
set_line_width(WindowHandle,1);
if (nol>0) T_disp_line(WindowHandleTuple,Ll0,Lc0,Ll1,Lc1);
```

Result

The operator `approx_chain` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

sobel_amp, edges_image, get_region_contour, threshold, hysteresis_threshold

Possible Successors

set_line_width, disp_arc, disp_line

Alternatives

get_region_polygon, approx_chain_simple

See also

get_region_chain, smallest_circle, disp_circle, disp_line

Module

Foundation

approx_chain_simple (: : Row, Column : ArcCenterRow, ArcCenterCol, ArcAngle, ArcBeginRow, ArcBeginCol, LineBeginRow, LineBeginCol, LineEndRow, LineEndCol, Order)

Approximate a contour by arcs and lines.

approx_chain_simple is obsolete and is only provided for reasons of backward compatibility.

The contour of a curve is approximated by a sequence of lines and arcs.

The result of the procedure is returned separately as arcs and lines. If one is interested in the sequence of the segments the individual resulting elements can be read successively from the resulting tuples. The sequence can be taken from the return parameter order (0: next element is next line segment, 1: next element is next arc segment).

The operator approx_chain_simple behaves similarly as approx_chain except that in the case of approx_chain_simple the missing parameters are internally allocated as follows: MinWidthCoord = 1.0, MaxWidthCoord = 3.0, ThreshStart = 0.5, ThreshEnd = 0.9, ThreshStep = 0.3, MinWidthSmooth = 1.0, MaxWidthSmooth = 3.0, MinWidthCurve = 2, MaxWidthCurve = 9, Weight1 = 1.0, Weight2 = 1.0, Weight3 = 1.0.

Parameters

- ▷ **Row** (input_control) ..... point.y-array ⇝ *integer*
  Row of the contour.
  Default: 32
- ▷ **Column** (input_control) ..... point.x-array ⇝ *integer*
  Column of the contour.
  Default: 32
- ▷ **ArcCenterRow** (output_control) ..... arc.center.y-array ⇝ *integer*
  Row of the center of an arc.
- ▷ **ArcCenterCol** (output_control) ..... arc.center.x-array ⇝ *integer*
  Column of the center of an arc.
- ▷ **ArcAngle** (output_control) ..... arc.angle.rad-array ⇝ *real*
  Angle of an arc.
- ▷ **ArcBeginRow** (output_control) ..... arc.begin.y-array ⇝ *integer*
  Row of the starting point of an arc.
- ▷ **ArcBeginCol** (output_control) ..... arc.begin.x-array ⇝ *integer*
  Column of the starting point of an arc.
- ▷ **LineBeginRow** (output_control) ..... line.begin.y-array ⇝ *integer*
  Row of the starting point of a line segment.
- ▷ **LineBeginCol** (output_control) ..... line.begin.x-array ⇝ *integer*
  Column of the starting point of a line segment.
- ▷ **LineEndRow** (output_control) ..... line.end.y-array ⇝ *integer*
  Row of the ending point of a line segment.
- ▷ **LineEndCol** (output_control) ..... line.end.x-array ⇝ *integer*
  Column of the ending point of a line segment.
- ▷ **Order** (output_control) ..... integer-array ⇝ *integer*
  Sequence of line (value 0) and arc segments (value 1).

Example

```
/* read edge image */
read_image(&Image,"fig1_kan");
/* construct edge region */
hysteresis_threshold(Image,&RK1,64,255,40,1);
connection(RK1,&Rand);
/* fetch chain code */
T_get_region_contour(Rand,&Rows,&Columns);
```

```
firstline = get_i(Tline,0);
firstcol = get_i(Tcol,0);
/* approximation with lines and circular arcs */
T_approx_chain_simple(Rows,Columns,
                &Bzl,&Bzc,&Br,&Bwl,&Bwc,&Ll0,&Lc0,&Ll1,&Lc1,&order);
nob = length_tuple(Bzl);
nol = length_tuple(Ll0);
/* draw lines and arcs */
set_i(WindowHandleTuple,WindowHandle,0);
set_line_width(WindowHandle,4);
if (nob>0) T_disp_arc(Bzl,Bzc,Br,Bwl,Bwc);
set_line_width(WindowHandle,1);
if (nol>0) T_disp_line(WindowHandleTuple,Ll0,Lc0,Ll1,Lc1);
```

Result

The operator `approx_chain_simple` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`sobel_amp`, `edges_image`, `get_region_contour`, `threshold`, `hysteresis_threshold`

Possible Successors

`set_line_width`, `disp_arc`, `disp_line`

Alternatives

`get_region_polygon`, `approx_chain`

See also

`get_region_chain`, `smallest_circle`, `disp_circle`, `disp_line`

Module

Foundation

clear_all_bar_code_models (: : :)

This operator is inoperable. It had the following function: Delete all bar code models and free the allocated memory

`clear_all_bar_code_models` is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use `clear_all_bar_code_models`, the operator will be removed in future versions.

.

Attention

`clear_all_bar_code_models` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_bar_code_models` *must not* be used in any application.

Result

The operator `clear_all_bar_code_models` returns the value `2` (`H_MSG_TRUE`) if all bar code models were freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Module

Bar Code

clear_all_barriers (: : :)

This operator is inoperable. It had the following function: Destroy all barrier synchronization objects.

clear_all_barriers is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_barriers, the operator will be removed in future versions.

.

Attention

clear_all_barriers exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_barriers *must not* be used in any application.

Result

clear_all_barriers returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_calib_data (: : :)

Free the memory of all calibration data models.

clear_all_calib_data is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_calib_data, the operator will be removed in future versions.

.

Attention

clear_all_calib_data exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_calib_data *must not* be used in any application.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Calibration

clear_all_camera_setup_models (: : :)

Free the memory of all camera setup models.

clear_all_camera_setup_models is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_camera_setup_models, the operator will be removed in future versions.

.

Attention

clear_all_camera_setup_models exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_camera_setup_models *must not* be used in any application.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Calibration

clear_all_class_gmm (: : :)

This operator is inoperable. It had the following function: Clear all Gaussian Mixture Models.

clear_all_class_gmm is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_class_gmm, the operator will be removed in future versions.

.

Attention

clear_all_class_gmm exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_class_gmm *must not* be used in any application.

Result

clear_all_class_gmm always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_class_knn (: : :)

This operator is inoperable. It had the following function: Clear all k-NN classifiers.

clear_all_class_knn is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_class_knn, the operator will be removed in future versions.

.

Attention

clear_all_class_knn exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_class_knn *must not* be used in any application.

Result

clear_all_class_knn always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

Foundation

clear_all_class_lut (: : :)

This operator is inoperable. It had the following function: Clear all look-up table classifiers.

clear_all_class_lut is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_class_lut, the operator will be removed in future versions.

.

Attention

clear_all_class_lut exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_class_lut *must not* be used in any application.

Result

clear_all_class_lut always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_class_mlp (: : :)

This operator is inoperable. It had the following function: Clear all multilayer perceptrons.

clear_all_class_mlp is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_class_mlp, the operator will be removed in future versions.

.

Attention

clear_all_class_mlp exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_class_mlp *must not* be used in any application.

Result

clear_all_class_mlp always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Module

Foundation

clear_all_class_svm (: : :)

This operator is inoperable. It had the following function: Clear all support vector machines.

clear_all_class_svm is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_class_svm, the operator will be removed in future versions.

.

Attention

clear_all_class_svm exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_class_svm *must not* be used in any application.

Result

clear_all_class_svm always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_class_train_data (: : :)

This operator is inoperable. It had the following function: Clear all training data for classifiers.

clear_all_class_train_data is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_class_train_data, the operator will be removed in future versions.

.

Attention

clear_all_class_train_data exists solely for the purpose of implementing the "reset program" functionality in HDevelop.

Result

clear_all_class_train_data returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_color_trans_luts (: : :)

This operator is inoperable. It had the following function: Delete all look-up-tables of the color space transformation.

clear_all_color_trans_luts is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_color_trans_luts, the operator will be removed in future versions.

.

Attention

After execution of clear_all_color_trans_luts all handles to look-up-tables become invalid. clear_all_color_trans_luts exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_color_trans_luts *must not* be used in any application.

Result

The operator clear_all_color_trans_luts returns the value 2 (H_MSG_TRUE) if all look-up-tables were successfully released.Otherwise, an exception will be raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_conditions (: : :)

This operator is inoperable. It had the following function: Destroy all condition synchronization objects.

clear_all_conditions is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use close_all_framegrabbers, the operator will be removed in future versions.

.

Attention

clear_all_conditions exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_conditions *must not* be used in any application.

Result

clear_all_conditions returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_data_code_2d_models (: : :)

This operator is inoperable. It had the following function: Delete all 2D data code models and free the allocated memory.

clear_all_data_code_2d_models is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_data_code_2d_models, the operator will be removed in future versions.

.

Attention

clear_all_data_code_2d_models exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_data_code_2d_models *must not* be used in any application.

Result

The operator clear_all_data_code_2d_models returns the value 2 (H_MSG_TRUE) if all 2D data code models were freed correctly. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Data Code

clear_all_deformable_models (: : :)

This operator is inoperable. It had the following function: Free the memory of all deformable models.

clear_all_deformable_models is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_deformable_models, the operator will be removed in future versions.

.

Attention

clear_all_deformable_models exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_deformable_models *must not* be used in any application.

Result

clear_all_deformable_models always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

clear_all_descriptor_models (: : :)

This operator is inoperable. It had the following function: Free the memory of all descriptor models in RAM.

clear_all_descriptor_models is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_descriptor_models, the operator will be removed in future versions.

.

Result

If the operator is successful, clear_descriptor_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

clear_all_events (: : :)

This operator is inoperable. It had the following function: Clear all event synchronization objects.

clear_all_events is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_events, the operator will be removed in future versions.

.

Attention

clear_all_events exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_events *must not* be used in any application.

Result

clear_all_events returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_lexica (: : :)

This operator is inoperable. It had the following function: Clear all lexica.

clear_all_lexica is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_lexica, the operator will be removed in future versions.

.

Attention

clear_all_lexica exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_lexica *must not* be used in any application.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

OCR/OCV

clear_all_matrices (: : :)

This operator is inoperable. It had the following function: Clear all matrices from memory.

clear_all_matrices is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_matrices, the operator will be removed in future versions.

.

Attention

clear_all_matrices exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_matrices *must not* be used in any application.

Result

clear_all_matrices always returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_metrology_models (: : :)

This operator is inoperable. It had the following function: Delete all metrology models and free the allocated memory.

clear_all_metrology_models is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_metrology_models, the operator will be removed in future versions.

.

Attention

clear_all_metrology_models exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_metrology_models *must not* be used in any application.

Result

The operator clear_all_metrology_models returns the value 2 (H_MSG_TRUE) if all metrology models were freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

2D Metrology

clear_all_mutexes (: : :)

This operator is inoperable. It had the following function: Clear all mutex synchronization objects.

clear_all_mutexes is obsolete, inoperable. and is only provided for reasons of backward compatibility. New applications should not use clear_all_mutexes, the operator will be removed in future versions.

.

Attention

`clear_all_mutexes` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_mutexes` *must not* be used in any application.

Result

`clear_all_mutexes` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_ncc_models (: : :)

This operator is inoperable. It had the following function: Free the memory of all NCC models.

`clear_all_ncc_models` is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use `clear_all_ncc_models`, the operator will be removed in future versions.

.

Attention

`clear_all_ncc_models` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_ncc_models` *must not* be used in any application.

Result

`clear_all_ncc_models` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

clear_all_object_model_3d (: : :)

This operator is inoperable. It had the following function: Free the memory of all 3D object models.

`clear_all_object_model_3d` is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use `clear_all_object_model_3d`, the operator will be removed in future versions.

.

Attention

`clear_all_object_model_3d` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_object_model_3d` *must not* be used in any application.

Result

`clear_all_object_model_3d` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

3D Metrology

clear_all_ocr_class_knn (: : :)

This operator is inoperable. It had the following function: Clear all OCR classifiers.

clear_all_ocr_class_knn is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_ocr_class_knn, the operator will be removed in future versions.

.

Attention

clear_all_ocr_class_knn exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_ocr_class_knn *must not* be used in any application.

Result

clear_all_ocr_class_knn always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

OCR/OCV

clear_all_ocr_class_mlp (: : :)

This operator is inoperable. It had the following function: Clear all OCR classifiers.

clear_all_ocr_class_mlp is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_ocr_class_mlp, the operator will be removed in future versions.

.

Attention

clear_all_ocr_class_mlp exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_ocr_class_mlp *must not* be used in any application.

Result

clear_all_ocr_class_mlp always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

OCR/OCV

clear_all_ocr_class_svm (: : :)

This operator is inoperable. It had the following function: Clear all SVM based OCR classifiers.

clear_all_ocr_class_svm is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_ocr_class_svm, the operator will be removed in future versions.

.

Attention

clear_all_ocr_class_svm exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_ocr_class_svm *must not* be used in any application.

Result

clear_all_ocr_class_svm always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

OCR/OCV

clear_all_sample_identifiers (: : :)

This operator is inoperable. It had the following function: Free the memory of all sample identifiers.

clear_all_sample_identifiers is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_sample_identifiers, the operator will be removed in future versions.

.

Attention

clear_all_sample_identifiers exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_sample_identifiers *must not* be used in any application.

Result

clear_all_sample_identifiers always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

clear_all_scattered_data_interpolators (: : :)

This operator is inoperable. It had the following function: Clear all scattered data interpolators.

clear_all_scattered_data_interpolators is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_scattered_data_interpolators, the operator will be removed in future versions.

.

Attention

clear_all_scattered_data_interpolators exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_scattered_data_interpolators *must not* be used in any application.

Result

clear_all_scattered_data_interpolators always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_serialized_items (: : :)

This operator is inoperable. It had the following function: Delete all current existing serialized items.

clear_all_serialized_items is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_serialized_items, the operator will be removed in future versions.

.

Attention

clear_all_serialized_items exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_serialized_items *must not* be used in any application.

Result

If the parameters are valid, the operator clear_all_serialized_items returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_shape_model_3d (: : :)

This operator is inoperable. It had the following function: Free the memory of all 3D shape models.

clear_all_shape_model_3d is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_shape_model_3d, the operator will be removed in future versions.

.

Attention

`clear_all_shape_model_3d` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_shape_model_3d` *must not* be used in any application.

Result

`clear_all_shape_model_3d` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

3D Metrology

clear_all_shape_models (: : :)

This operator is inoperable. It had the following function: Free the memory of all shape models.

`clear_all_shape_models` is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use `clear_all_shape_models`, the operator will be removed in future versions.

.

Attention

`clear_all_shape_models` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_shape_models` *must not* be used in any application.

Result

`clear_all_shape_models` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

clear_all_sheet_of_light_models (: : :)

This operator is inoperable. It had the following function: Delete all sheet-of-light models and free the allocated memory.

`clear_all_sheet_of_light_models` is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use `clear_all_sheet_of_light_models`, the operator will be removed in future versions..

.

Attention

`clear_all_sheet_of_light_models` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_sheet_of_light_models` *must not* be used in any application.

Result

The operator `clear_all_sheet_of_light_models` returns the value `2` (`H_MSG_TRUE`) if all sheet-of-light models were freed correctly. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

3D Metrology

clear_all_stereo_models (: : :)

This operator is inoperable. It had the following function: Free the memory of all stereo models.

clear_all_stereo_models is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_stereo_models, the operator will be removed in future versions..

.

Attention

clear_all_stereo_models exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_stereo_models *must not* be used in any application.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

3D Metrology

clear_all_surface_matching_results (: : :)

This operator is inoperable. It had the following function: Free the memory of all surface matching results.

clear_all_surface_matching_results is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_surface_matching_results, the operator will be removed in future versions.

.

Attention

clear_all_surface_matching_results exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_surface_matching_results *must not* be used in any application.

Result

clear_all_surface_matching_results always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

3D Metrology

clear_all_surface_models (: : :)

This operator is inoperable. It had the following function: Free the memory of all surface models.

clear_all_surface_models is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_surface_models, the operator will be removed in future versions.

.

Attention

clear_all_surface_models exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_surface_models *must not* be used in any application.

Result

clear_all_surface_models always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

3D Metrology

clear_all_templates (: : :)

This operator is inoperable. It had the following function: Deallocation of the memory of all templates.

clear_all_templates is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_templates. The operator will be removed with HALCON 25.11.

.

Attention

clear_all_templates exists solely for the purpose of implementing the "reset program" functionality in HDevelop. clear_all_templates *must not* be used in any application.

Result

clear_all_templates always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

clear_all_text_models (: : :)

This operator is inoperable. It had the following function: Clear all text models.

clear_all_text_models is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use clear_all_text_models, the operator will be removed in future versions..

.

Attention

`clear_all_text_models` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_text_models` *must not* be used in any application.

Result

`clear_all_text_models` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

OCR/OCV

clear_all_text_results (: : :)

This operator is inoperable. It had the following function: Clear all text results.

`clear_all_text_results` is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use `clear_all_text_results`, the operator will be removed in future versions.

.

Attention

`clear_all_text_results` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_text_results` *must not* be used in any application.

Result

`clear_all_text_results` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

clear_all_variation_models (: : :)

This operator is inoperable. It had the following function: Free the memory of all variation models.

`clear_all_variation_models` is obsolete, inoperable and is only provided for reasons of backward compatibility. New applications should not use `clear_all_variation_models`, the operator will be removed in future versions.

.

Attention

`clear_all_variation_models` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `clear_all_variation_models` *must not* be used in any application.

Result

`clear_all_variation_models` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Matching

close_all_bg_esti (: : :)

This operator is inoperable. It had the following function: Delete all background estimation data sets.

close_all_bg_esti is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use close_all_bg_esti, the operator will be removed in future versions.

.

Attention

close_all_bg_esti exists solely for the purpose of implementing the "reset program" functionality in HDevelop. close_all_bg_esti *must not* be used in any application.

Result

If it is possible to close the background estimation data sets the operator close_all_bg_esti returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

close_all_class_box (: : :)

This operator is inoperable. It had the following function: Destroy all classifiers.

close_all_class_box is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use close_all_class_box, the operator will be removed with HALCON 25.11.

.

Attention

close_all_class_box exists solely for the purpose of implementing the "reset program" functionality in HDevelop. close_all_class_box *must not* be used in any application.

Result

If it is possible to close the classifiers the operator close_all_class_box returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

close_all_files (: : :)

This operator is inoperable. It had the following function: Close all open files.

close_all_files is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use close_all_files, the operator will be removed in future version.

.

Attention

close_all_files exists solely for the purpose of implementing the "reset program" functionality in HDevelop. close_all_files *must not* be used in any application.

Result

If it is possible to close the files the operator close_all_files returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

close_all_framegrabbers (: : :)

This operator is inoperable. It had the following function: Close all image acquisition devices.

close_all_framegrabbers is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use close_all_framegrabbers, the operator will be removed in future versions.

.

Attention

close_all_framegrabbers exists solely for the purpose of implementing the "reset program" functionality in HDevelop. close_all_framegrabbers *must not* be used in any application.

Result

If it is possible to close all image acquisition devices, the operator close_all_framegrabbers returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

close_all_measures (: : :)

This operator is inoperable. It had the following function: Delete all measure objects.

close_all_measures is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use close_all_measures, the operator will be removed in future versions.

.

Attention

`close_all_measures` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `close_all_measures` *must not* be used in any application.

Result

`close_all_measures` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

1D Metrology

close_all_ocrs (: : :)

This operator is inoperable. It had the following function: Destroy all OCR classifiers.

`close_all_ocrs` is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use `close_all_ocrs`, the operator will be removed in future versions.

.

Attention

`close_all_ocrs` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `close_all_ocrs` *must not* be used in any application.

Result

If it is possible to close the OCR classifiers the operator `close_all_ocrs` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

OCR/OCV

close_all_ocvs (: : :)

This operator is inoperable. It had the following function: Clear all OCV tools.

`close_all_ocvs` is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use `close_all_ocvs`, the operator will be removed in future versions.

.

Attention

`close_all_ocvs` exists solely for the purpose of implementing the "reset program" functionality in HDevelop. `close_all_ocvs` *must not* be used in any application.

Result

`close_all_ocvs` returns always `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

OCR/OCV

close_all_serials (: : :)

This operator is inoperable. It had the following function: Close all serial devices.

close_all_serials is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use close_all_serials, the operator will be removed in future versions.

.

Attention

close_all_serials exists solely for the purpose of implementing the "reset program" functionality in HDevelop. close_all_serials *must not* be used in any application.

Result

close_all_serials returns always 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

close_all_sockets (: : :)

This operator is inoperable. It had the following function: Close all opened sockets.

close_all_sockets is obsolete, inoperable, and is only provided for reasons of backward compatibility. New applications should not use close_all_sockets, the operator will be removed in future versions.

.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

distance_funct_1d (: : Function1, Function2, Mode, Sigma : Distance)

Compute the distance of two functions.

distance_funct_1d is obsolete and is only provided for reasons of backward compatibility.

distance_funct_1d calculates the distance of two functions. The two functions may differ in length.

Parameters

▷ **Function1** (input_control) ..... function_1d ⇝ *real* / integer
Input function 1.

▷ **Function2** (input_control) ..... function_1d ⇝ *real* / integer
Input function 2.

▷ **Mode** (input_control) ..... string(-array) ⇝ *string*
Modes of invariants.
Default: 'length'
List of values: `Mode` ∈ {'length', 'mean'}

▷ **Sigma** (input_control) ..... number(-array) ⇝ *real*
Variance of the optional smoothing with a Gaussian filter.
Default: 0.0
Suggested values: `Sigma` ∈ {0.0, 0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0, 15.0, 20.0, 25.0, 30.0, 40.0, 50.0}

▷ **Distance** (output_control) ..... real-array ⇝ *real* / integer
Distance of the functions.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

filter_kalman (: : Dimension, Model, Measurement, PredictionIn : PredictionOut, Estimate)

Estimate the current state of a system with the help of the Kalman filtering.

filter_kalman is obsolete and is only provided for reasons of backward compatibility.

The operator `filter_kalman` returns an estimate of the current state (or also a prediction of a future state) of a discrete, stochastically disturbed, linear system. In practice, Kalman filters are used successfully in image processing in the analysis of image sequences (background identification, lane tracking with the help of line tracing or region analysis, etc.). A short introduction concerning the theory of the Kalman filters will be followed by a detailed description of the routine `filter_kalman` itself.

KALMAN FILTER: A discrete, stochastically disturbed, linear system is characterized by the following markers:

- State $x(t)$: Describes the current state of the system (speeds, temperatures,...).
- Parameter $u(t)$: Inputs from outside into the system.
- Measurement $y(t)$: Measurements gained by observing the system. They indicate the state of the system (or at least parts of it).
- An output function describing the dependence of the measurements on the state.
- A transition function indicating how the state changes with regard to time, the current value and the parameters.

The output function and the transition function are linear. Their application can therefore be written as a multiplication with a matrix.

The transition function is described with the help of the transition matrix $A(t)$ and the parameter matrix $G(t)$, the initial function is described by the measurement matrix $C(t)$. Hereby $C(t)$ characterizes the dependency of the new state on the old, $G(t)$ indicates the dependency on the parameters. In practice it is rarely possible (or at least too time consuming) to describe a real system and its behavior in a complete and exact way. Normally only a relatively small number of variables will be used to simulate the behavior of the system. This leads to an error, the so called system error (also called system disturbance) $v(t)$.

The output function, too, is usually not exact. Each measurement is faulty. The measurement errors will be called $w(t)$. Therefore the following system equations arise:

$$\begin{aligned} x(t+1) &= A(t)x(t) + G(t)u(t) + v(t) \\ y(t) &= c(t)x(t) + w(t) \end{aligned}$$

The system error $v(t)$ and the measurement error $w(t)$ are not known. As far as systems are concerned which are interpreted with the help of the Kalman filter, these two errors are considered as Gaussian distributed random vectors (therefore the expression "stochastically disturbed systems"). Therefore the system can be calculated, if the corresponding expected values for $v(t)$ and $w(t)$ as well as the covariance matrices are known.

The estimation of the state of the system is carried out in the same way as in the Gaussian-Markov-estimation. However, the Kalman filter is a recursive algorithm which is based only on the current measurements $y(t)$ and the latest state $x(t)$. The latter implicitly also includes the knowledge about earlier measurements.

A suitable estimate value x_0, which is interpreted as the expected value of a random variable for $x(0)$, must be indicated for the initial value $x(0)$. This variable should have an expected error value of 0 and the covariance matrix P_0 which also has to be indicated. At a certain time t the expected values of both disturbances $v(t)$ and $w(t)$ should be 0 and their covariances should be $Q(t)$ and $R(t)$. $x(t)$, $v(t)$ and $w(t)$ will usually be assumed to be not correlated (any kind of noise-process can be modeled - however the development of the necessary matrices by the user will be considerably more demanding). The following conditions must be met by the searched estimate values x^t:

- The estimate values x^t are linearly dependent on the actual value $x(t)$ and on the measurement sequence $y(0), y(1), \cdots, y(t)$.
- x^t being hereby considered to meet its expectations, i.e. $Ex^t = Ex(t)$.
- The grade criterion for x^t is the criterion of minimal variance, i.e. the variance of the estimation error defined as $x(t) - x^t$, being as small as possible.

After the initialization

$$\hat{x}(0) = x_0, \hat{P}(0) = P_0$$

at each point in time t the Kalman filter executes the following calculation steps:

$$\begin{aligned} (K-III) \quad K(t) &= \frac{\hat{P}(t)C'(t)}{C(t)\hat{P}(t)C'(t)+R(t)} \\ (K-IV) \quad x^t &= \hat{x}(t) + K(t)(y(t) - C(t)\hat{x}(t)) \\ (K-V) \quad \tilde{P}(t) &= \hat{P}(t) - K(t)C(t)\hat{P}(t) \\ (K-I) \quad \hat{x}(t+1) &= A(t)x^t + G(t)u(t) \\ (K-II) \quad \hat{P}(t+1) &= A(t)\tilde{P}(t)A'(t) + Q(t) \end{aligned}$$

Hereby $\tilde{P}(t)$ is the covariance matrix of the estimation error, $\hat{x}(t)$ is the extrapolation value respective the prediction value of the state, $\hat{P}(t)$ are the covariances of the prediction error $\hat{x} - x$, K is the amplifier matrix (the so called Kalman gain), and X' is the transposed of a matrix X.

Please note that the prediction of the future state is also possible with the equation (K-I). Sometimes this is very useful in image processing in order to determine "regions of interest" in the next image.

As mentioned above, it is much more demanding to model any kind of noise processes. If for example the system noise and the measurement noise are correlated with the corresponding covariance matrix L, the equations for the Kalman gain and the error covariance matrix have to be modified:

$$\begin{aligned} (K-III) \quad K(t) &= \frac{\hat{P}(t)C'(t)+L(t)}{C(t)\hat{P}(t)+C(t)l(t)+L'C'(t)+R(t)} \\ (K-V) \quad \tilde{P}(t) &= (\hat{P}(t) - K(t)C(t)\hat{P}(t))\hat{P}(t) - K(t)L(t) \end{aligned}$$

This means that the user himself has to establish the linear system equations from (K-I) up to (K-V) with respect to the actual problem. The user must therefore develop a mathematical model upon which the solution to the problem can be based. Statistical characteristics describing the inaccuracies of the system as well as the measurement errors, which are to be expected, thereby have to be estimated if they cannot be calculated exactly. Therefore the following individual steps are necessary:

1. Developing a mathematical model
2. Selecting characteristic state variables
3. Establishing the equations describing the changes of these state variables and their linearization (matrices A and G)
4. Establishing the equations describing the dependency of the measurement values of the system on the state variables and their linearization (matrix C)
5. Developing or estimating of statistical dependencies between the system disturbances (matrix Q)
6. Developing or estimating of statistical dependencies between the measurement errors (matrix R)
7. Initialization of the initial state

As mentioned above, the initialization of the system (point 7) hereby necessitates to indicate an estimate x_0 of the state of the system at the time 0 and the corresponding covariance matrix P_0. If the exact initial state is not known, it is recommendable to set the components of the vector x_0 to the average values of the corresponding range, and to set high values for P_0 (about the size of the squares of the range). After a few iterations (when the number of the accumulated measurement values in total has exceeded the number of the system values), the values which have been determined in this way are also usable.

If on the other hand the initial state is known exactly, all entries for P_0 have to be set to 0, because P_0 describes the covariances of the error between the estimated value x_0 and the actual value $x(0)$.

THE FILTER ROUTINE:

A Kalman filter is dependent on a range of data which can be organized in four groups:

Model parameter: transition matrix A, control matrix G including the parameter u and the measurement matrix C

Model stochastic: system-error covariance matrix Q, system-error - measurement-error covariance matrix L, and measurement-error covariance matrix R

Measurement vector: y

History of the system: extrapolation vector $\hat{x}$ and extrapolation-error covariance matrix $\hat{P}$

Thereby many systems can work without input "from outside", i.e. without G and u. Further, system errors and measurement errors are normally not correlated (L is dropped).

Actually the data necessary for the routine will be set by the following parameters:

`Dimension`: This parameter includes the dimensions of the status vector, the measurement vector and the controller vector. `Dimension` thereby is a vector $[n, m, p]$, whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members. For a system without determining control (i.e. without influence "from outside") therefore $[n, m, 0]$ has to be passed.

`Model`: This parameter includes the lined up matrices (vectors) A, C, Q, G, u and (if necessary) L having been stored in row-major order. `Model` therefore is a vector of the length $n \times n + n \times m + n \times n + n \times p + p[+n \times m]$. The last summand is dropped, in case the system errors and measurement errors are not correlated, i.e. there is no value for L.

`Measurement`: This parameter includes the matrix R which has been stored in row-major order, and the measurement vector y lined up. `Measurement` therefore is a vector of the dimension $m \times m + m$.

`PredictionIn` / `PredictionOut`: These two parameters include the matrix $\hat{P}$ (the extrapolation-error covariance matrix) which has been stored in row-major order and the extrapolation vector $\hat{x}$ lined up. This means, they are vectors of the length $n \times n + n$. `PredictionIn` therefore is an input parameter, which must contain $\hat{P}(t)$ and $\hat{x}(t)$ at the current time t. With `PredictionOut` the routine returns the corresponding predictions $\hat{P}(t+1)$ and $\hat{x}(t+1)$.

`Estimate`: With this parameter the routine returns the matrix $\tilde{P}$ (the estimation-error covariance matrix) which has been stored in row-major order and the estimated state $\tilde{x}$ lined up. `Estimate` therefore is a vector of the length $n \times n + n$.

Please note that the covariance matrices (Q, R, $\hat{P}$, $\tilde{P}$) must of course be symmetric.

Parameters

▷ **Dimension** (input_control) . . . . . integer-array ⇝ *integer*
The dimensions of the state vector, the measurement and the controller vector.
Default: [3,1,0]
Value range: $0 \leq$ `Dimension` ≤ 30

▷ **Model** (input_control) . . . . . real-array ⇝ *real*
The lined up matrices A, C, Q, possibly G and u, and if necessary L which have been stored in row-major order.
Default: [1.0,1.0,0.5,0.0,1.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]
Value range: $0.0 \leq$ `Model` ≤ 10000.0

▷ **Measurement** (input_control) . . . . . real-array ⇝ *real*
The matrix R stored in row-major order and the measurement vector y lined up.
Default: [1.2,1.0]
Value range: $0.0 \leq$ `Measurement` ≤ 10000.0

▷ **PredictionIn** (input_control) . . . . . real-array ⇝ *real*
The matrix $\hat{P}$ (the extrapolation-error covariances) stored in row-major order and the extrapolation vector $\hat{x}$ lined up.
Default: [0.0,0.0,0.0,0.0,180.5,0.0,0.0,0.0,0.0,100.0,0.0,0.0,100.0,0.0]
Value range: $0.0 \leq$ `PredictionIn` ≤ 10000.0

▷ **PredictionOut** (output_control) . . . . . real-array ⇝ *real*
The matrix P$*$ (the extrapolation-error covariances)stored in row-major order and the extrapolation vector $\hat{x}$ lined up.

▷ **Estimate** (output_control) . . . . . real-array ⇝ *real*
The matrix $\tilde{P}$ (the estimation-error covariances) stored in row-major order and the estimated state $\tilde{x}$ lined up.

Example

```
* Typical procedure:
* To initialize the variables, which describe the model, e.g., with
read_kalman('kalman.init',Dim,Mod,Meas,Pred)

* Generation of the first measurements (typical of the first image of an
* image series) with an appropriate problem-specific procedure (there is a
* fictitious procedure extract_features in example):
* extract_features(Image1,Meas,Meas1)

* first Kalman-Filtering:
filter_kalman(Dim,Mod,Meas1,Pred,Pred1,Est1)

* To use the estimate value (if need be the prediction too)
* with a problem-specific procedure (here use_est):
* use_est(Est1)

* To get the next measurements (e.g., from the next image):
* extract_next_features(Image2,Meas1,Meas2)

* if need be Update of the model parameter (a constant model)
* second Kalman-Filtering:
filter_kalman(Dim,Mod,Meas2,Pred1,Pred2,Est2)
* use_est(Est2)
* extract_next_features(Image3,Meas2,Meas3)
* etc.
```

Result

If the parameter values are correct, the operator `filter_kalman` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_kalman

Possible Successors

update_kalman

See also

read_kalman, update_kalman

References

W.Hartinger: "Entwurf eines anwendungsunabhängigen Kalman-Filters mit Untersuchungen im Bereich der Bildfolgenanalyse"; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

R.E.Kalman: "A New Approach to Linear Filtering and Prediction Problems"; Transactions ASME, Ser.D: Journal of Basic Engineering; Vol. 82, S.34-45; 1960.

R.E.Kalman, P.l.Falb, M.A.Arbib: "Topics in Mathematical System Theory"; McGraw-Hill Book Company, New York; 1969.

K-P. Karmann, A.von Brandt: "Moving Object Recognition Using an Adaptive Background Memory"; Time-Varying Image Processing and Moving Object Recognition 2 (ed.: V. Cappellini), Proc. of the 3rd Interantional Workshop, Florence, Italy, May, 29th - 31st, 1989; Elsevier, Amsterdam; 1990.

Module

Foundation

intersection_ll (: : RowA1, ColumnA1, RowA2, ColumnA2, RowB1, ColumnB1, RowB2, ColumnB2 : Row, Column, IsParallel)

Calculate the intersection point of two lines.

intersection_ll is obsolete and is only provided for reasons of backward compatibility. New applications should use the operator intersection_lines of the chapter Tools / Geometry instead.

The operator intersection_ll calculates the intersection point of two lines. As input the two points on each line are expected (RowA1,ColumnA1, RowA2,ColumnA2) and (RowB1,ColumnB1, RowB2,ColumnB2). The parameters Row and Column return the result of the calculation. If the lines are parallel, the values of Row and Column are undefined and IsParallel is 1. Otherwise, IsParallel is 0.

Attention

If the lines are parallel the values of Row and Column are undefined.

Parameters

▷ **RowA1** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinate of the first point of the first line.

▷ **ColumnA1** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the first line.

▷ **RowA2** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the first line.

▷ **ColumnA2** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the first line.

▷ **RowB1** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinate of the first point of the second line.

▷ **ColumnB1** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the second line.

▷ **RowB2** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the second line.

▷ **ColumnB2** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the second line.

▷ **Row** (output_control) . . . . . point.y(-array) ⇝ *real*
Row coordinate of the intersection point.

▷ **Column** (output_control) . . . . . point.x(-array) ⇝ *real*
Column coordinate of the intersection point.

▷ **IsParallel** (output_control) . . . . . number(-array) ⇝ *integer*
Are the two lines parallel?

Example

```
dev_set_color ('black')
RowLine1 := 350
ColLine1 := 250
RowLine2 := 300
ColLine2 := 300
Rows := 300
Columns := 50
disp_line (WindowHandle, RowLine1, ColLine1, RowLine2, ColLine2)
n := 0
for Rows := 40 to 200 by 4
  dev_set_color ('red')
  disp_line (WindowHandle, Rows, Columns, Rows+n, Columns+n)
  intersection_ll (Rows, Columns, Rows+n, Columns+n, RowLine1, ColLine1, \
                   RowLine2, ColLine2, Row, Column, IsParallel)
  dev_set_color ('blue')
  disp_line (WindowHandle, Row, Column-2, Row, Column+2)
  disp_line (WindowHandle, Row-2, Column, Row+2, Column)
  n := n+8
endfor
```

Result

`intersection_ll` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

partition_lines (: : RowBeginIn, ColBeginIn, RowEndIn, ColEndIn, Feature, Operation, Min, Max : RowBeginOut, ColBeginOut, RowEndOut, ColEndOut, FailRowBOut, FailColBOut, FailRowEOut, FailColEOut)

Partition lines according to various criteria.

partition_lines is obsolete and is only provided for reasons of backward compatibility.

The operator `partition_lines` divides lines into two sets according to various criteria. For each input line the indicated features (`Feature`) are calculated. If each (`Operation` = *'and'*) or at least one (`Operation` = *'or'*) of the calculated features is within the given limits (`Min`,`Max`) the line is transferred into the first set (parameters `RowBeginOut` to `ColEndOut`), otherwise into the second set (parameters `FailRowBOut` to `FailColEOut`). The default values *'min'* and *'max'* of the parameters `Min` and `Max` are used to leave bottom and top limit, respectively, open.

Condition: $Min_i \leq Feature_i(Line) \leq Max_i$

Possible values for `Feature`:

'length' (Euclidean) length of the line

'row' Line index of the center

'column' Column index of the center

'phi' Orientation of the line ($-\frac{\pi}{2} < \varphi \leq \frac{\pi}{2}$)

Attention

If only one feature is used the value of `Operation` is meaningless. Several features are processed according to the sequence in which they are passed.

Parameters

- ▷ **RowBeginIn** (input_control) ..... line.begin.y-array ⇝ *integer*
  Row coordinates of the starting points of the input lines.
- ▷ **ColBeginIn** (input_control) ..... line.begin.x-array ⇝ *integer*
  Column coordinates of the starting points of the input lines.
- ▷ **RowEndIn** (input_control) ..... line.end.y-array ⇝ *integer*
  Row coordinates of the ending points of the input lines.
- ▷ **ColEndIn** (input_control) ..... line.end.x-array ⇝ *integer*
  Column coordinates of the ending points of the input lines.
- ▷ **Feature** (input_control) ..... string(-array) ⇝ *string*
  Features to be used for selection.
  List of values: `Feature` ∈ {'length', 'row', 'column', 'phi'}
- ▷ **Operation** (input_control) ..... string ⇝ *string*
  Desired combination of the features.
  List of values: `Operation` ∈ {'and', 'or'}
- ▷ **Min** (input_control) ..... string(-array) ⇝ *string* / integer / real
  Lower limits of the features or 'min'.
  Default: 'min'
- ▷ **Max** (input_control) ..... string(-array) ⇝ *string* / integer / real
  Upper limits of the features or 'max'.
  Default: 'max'
- ▷ **RowBeginOut** (output_control) ..... line.begin.y-array ⇝ *integer*
  Row coordinates of the starting points of the lines fulfilling the conditions.
- ▷ **ColBeginOut** (output_control) ..... line.begin.x-array ⇝ *integer*
  Column coordinates of the starting points of the lines fulfilling the conditions.
- ▷ **RowEndOut** (output_control) ..... line.end.y-array ⇝ *integer*
  Row coordinates of the ending points of the lines fulfilling the conditions.
- ▷ **ColEndOut** (output_control) ..... line.begin.x-array ⇝ *integer*
  Column coordinates of the ending points of the lines fulfilling the conditions.
- ▷ **FailRowBOut** (output_control) ..... line.begin.y-array ⇝ *integer*
  Row coordinates of the starting points of the lines not fulfilling the conditions.
- ▷ **FailColBOut** (output_control) ..... line.begin.x-array ⇝ *integer*
  Column coordinates of the starting points of the lines not fulfilling the conditions.
- ▷ **FailRowEOut** (output_control) ..... line.end.y-array ⇝ *integer*
  Row coordinates of the ending points of the lines not fulfilling the conditions.
- ▷ **FailColEOut** (output_control) ..... line.end.x-array ⇝ *integer*
  Column coordinates of the ending points of the lines not fulfilling the conditions.

Result

The operator `partition_lines` returns the value `2` (`H_MSG_TRUE`) if the parameter values are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

sobel_amp, edges_image, threshold, hysteresis_threshold, split_skeleton_region, split_skeleton_lines

Possible Successors

set_line_width, disp_line

Alternatives

line_orientation, line_position, select_lines, select_lines_longest

See also

select_lines, select_lines_longest, detect_edge_segments, select_shape

Module

Foundation

read_kalman (: : FileName : Dimension, Model, Measurement, Prediction)

Read the description file of a Kalman filter.

read_kalman is obsolete and is only provided for reasons of backward compatibility.

The operator read_kalman reads the description file FileName of a Kalman filter. Kalman filters return an estimate of the current state (or even the prediction of a future state) of a discrete, stochastically disturbed, linear system. They are successfully used in image processing, especially in the analysis of image sequences. A Kalman filtering is based on a mathematical model of the system to be examined which at any point in time has the following characteristics:

Model parameter: transition matrix A, control matrix G including the controller output u and the measurement matrix C

Model stochastic: system-error covariance matrix Q, system-error - measurement-error covariance matrix L and measurement-error covariance matrix R

Estimate of the initial state of the system: state x_0 and corresponding covariance matrix P_0

Many systems do not need entries "from outside", and therefore G and u can be dropped. Further, system errors and measurement errors are normally not correlated (L is dropped). The characteristics mentioned above can be stored in an ASCII-file and then can be read with the help of the operator read_kalman. This ASCII-file must have the following structure:

Dimension row
+ content row
+ matrix A
+ matrix C
+ matrix Q
[+ matrix G + vector u]
[+ matrix L]
+ matrix R
[+ matrix P_0]
[+ vector x_0]

The dimension row thereby is always of the following form:

n = <integer> m = <integer> p = <integer>

whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members (see also Dimension). The maximal dimension will hereby be limited by a system constant (= 30 for the time being).

The content row has the following form:

$$A * C * Q * G * u * L * R * P * x*$$

and describes the following content of the file. Instead of '$*$', '+' (= parameter is available) respectively '-' (= parameter is missing) have to be set. Please note that only the parameters marked by [...] in the above list may be

left out in the description file. If the initial state estimate a_0 is missing (i.e. 'x-'), the components of the vector will supposed to be 0.0. If the covariance matrix P_0 of the initial state estimate is missing (i.e. 'P-'), the error will be supposed to be tremendous. In this case the matrix elements will be set to 10000.0. This value seems to be very high, however, it is only sufficient if the range of components of the state vector x is smaller to the tenth power. $(r \times s)$ matrices will be stored per row in the following form:

$$\begin{array}{cccc} < comment, i.e.\ string > & & & \\ < a_{11} > & < a_{12} > & \cdots & < a_{1s} > \\ \vdots & & \ddots & \vdots \\ < a_{r1} > & < a_{r2} > & \cdots & < a_{rs} > \end{array}$$

(the spaces and line feed characters can be chosen at will),

vectors will be stored correspondingly in the following form:

$$\begin{array}{ccc} < comment, i.e. string > & & \\ < a_1 > & \cdots & < a_k > \end{array}$$

The following parameter values are returned by the operator `read_kalman`:

Dimension: This parameter includes the dimensions of the status vector, the measurement vector and the controller vector. `Dimension` thereby is a vector $[n, m, p]$, whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members. For a system without determining control (i.e. without influence "from outside") therefore `Dimension` = $[n, m, 0]$.

Model: This parameter includes the lined up matrices (vectors) A, C, Q, G, u and (if necessary) L having been stored in row-major order. `Model` therefore is a vector of the length $n \times n + n \times m + n \times n + n \times p + p[+n \times m]$. The last summand is dropped, in case the system errors and measurement errors are not correlated, i.e. there is no value for L.

Measurement: This parameter includes the matrix R which has been stored in row-major order. `Measurement` therefore is vector of the dimension $m \times m$.

Prediction: This parameter includes the matrix P_0 (the error covariance matrix of the initial state estimate) and the initial state estimate x_0 lined up. This means, it is a vector of the length $n \times n + n$.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Description file for a Kalman filter.
Default: 'kalman.init'

▷ **Dimension** (output_control) . . . . . integer-array ⇝ *integer*
The dimensions of the state vector, the measurement vector and the controller vector.

▷ **Model** (output_control) . . . . . real-array ⇝ *real*
The lined up matrices A, C, Q, possibly G and u, and if necessary L stored in row-major order.

▷ **Measurement** (output_control) . . . . . real-array ⇝ *real*
The matrix R stored in row-major order.

▷ **Prediction** (output_control) . . . . . real-array ⇝ *real*
The matrix P_0 (error covariance matrix of the initial state estimate) stored in row-major order and the initial state estimate x_0 lined up.

Example

```
* An example of the description-file:
*
* n=3 m=1 p=0
* A+C+Q+G-u-L-R+P+x+
* transition matrix A:
* 1 1 0.5
* 0 1 1
* 0 0 1
```

```
* measurement matrix C:
* 1 0 0
* system-error covariance matrix Q:
* 54.3 37.9 48.0
* 37.9 34.3 42.5
* 48.0 42.5 43.7
* measurement-error covariance matrix R:
* 1.2
* estimation-error covariance matrix (for the initial estimate) P0: \
* 0 0 0
* 0 180.5 0
* 0 0 100
* initial estimate x0:
* 0 100 0
*
* the result of read_kalman with the upper descriptionfile
* as inputparameter:
*
* Dimension   = [3,1,0]
* Model       = [1.0,1.0,0.5,0.0,1.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,
*                      54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]
* Measurement = [1.2]
* Prediction  = [0.0,0.0,0.0,0.0,180.5,0.0,0.0,0.0,100.0,0.0,100.0,
*                           0.0].
```

Result

If the description file is readable and correct, the operator `read_kalman` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`filter_kalman`

See also

`update_kalman`, `filter_kalman`

Module

Foundation

`select_lines` (: : RowBeginIn, ColBeginIn, RowEndIn, ColEndIn, Feature, Operation, Min, Max : RowBeginOut, ColBeginOut, RowEndOut, ColEndOut)

Select lines according to various criteria.

`select_lines` is obsolete and is only provided for reasons of backward compatibility.

The operator `select_lines` chooses lines according to various criteria. For every input line the indicated features (`Feature`) are calculated. If each (`Operation` = *'and'*) or at least one (`Operation` = *'or'*) of the calculated features is within the given limits (`Min`,`Max`) the line is transferred into the output. The default values *'min'* and *'max'* of the parameters `Min` and `Max` are used to leave bottom and top limits, respectively, open.

Condition:

$$Min_i \leq Feature_i(Line) \leq Max_i$$

Possible values for `Feature`:

'length' (Euclidean) length of the line

'row' Line index of the center

'column' Column index of the center

'phi' Orientation of the line ($-\frac{\pi}{2} < \varphi \leq \frac{\pi}{2}$)

Attention

If only one feature is used the value of `Operation` is meaningless. Several features are processed according to the sequence in which they are passed.

Parameters

- ▷ **`RowBeginIn`** (input_control) . . . . . line.begin.y-array ⇝ *integer*
  Row coordinates of the starting points of the input lines.
- ▷ **`ColBeginIn`** (input_control) . . . . . line.begin.x-array ⇝ *integer*
  Column coordinates of the starting points of the input lines.
- ▷ **`RowEndIn`** (input_control) . . . . . line.end.y-array ⇝ *integer*
  Row coordinates of the ending points of the input lines.
- ▷ **`ColEndIn`** (input_control) . . . . . line.end.x-array ⇝ *integer*
  Column coordinates of the ending points of the input lines.
- ▷ **`Feature`** (input_control) . . . . . string(-array) ⇝ *string*
  Features to be used for selection.
  Default: 'length'
  List of values: `Feature` ∈ {'length', 'row', 'column', 'phi'}
- ▷ **`Operation`** (input_control) . . . . . string ⇝ *string*
  Desired combination of the features.
  Default: 'and'
  List of values: `Operation` ∈ {'and', 'or'}
- ▷ **`Min`** (input_control) . . . . . string(-array) ⇝ *string* / integer / real
  Lower limits of the features or 'min'.
  Default: 'min'
- ▷ **`Max`** (input_control) . . . . . string(-array) ⇝ *string* / integer / real
  Upper limits of the features or 'max'.
  Default: 'max'
- ▷ **`RowBeginOut`** (output_control) . . . . . line.begin.y-array ⇝ *integer*
  Row coordinates of the starting points of the output lines.
- ▷ **`ColBeginOut`** (output_control) . . . . . line.begin.x-array ⇝ *integer*
  Column coordinates of the starting points of the output lines.
- ▷ **`RowEndOut`** (output_control) . . . . . line.end.y-array ⇝ *integer*
  Row coordinates of the ending points of the output lines.
- ▷ **`ColEndOut`** (output_control) . . . . . line.end.x-array ⇝ *integer*
  Column coordinates of the ending points of the output lines.

Result

The operator `select_lines` returns the value `2` (`H_MSG_TRUE`) if the parameter values are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`sobel_amp`, `edges_image`, `threshold`, `hysteresis_threshold`, `split_skeleton_region`, `split_skeleton_lines`

Possible Successors

`set_line_width`, `disp_line`

Alternatives

line_orientation, line_position, partition_lines

See also

partition_lines, select_lines_longest, detect_edge_segments, select_shape

Module

Foundation

select_lines_longest (: : RowBeginIn, ColBeginIn, RowEndIn, ColEndIn, Num : RowBeginOut, ColBeginOut, RowEndOut, ColEndOut)

Select the longest input lines.

select_lines_longest is obsolete and is only provided for reasons of backward compatibility.

The operator select_lines_longest selects the Num longest input lines from the input lines described by the tuples RowBeginIn, ColBeginIn, RowEndIn and ColEndIn.

Parameters

- ▷ **RowBeginIn** (input_control) ..... line.begin.y-array ⇝ *integer*
  Row coordinates of the starting points of the input lines.
- ▷ **ColBeginIn** (input_control) ..... line.begin.x-array ⇝ *integer*
  Column coordinates of the starting points of the input lines.
- ▷ **RowEndIn** (input_control) ..... line.end.y-array ⇝ *integer*
  Row coordinates of the ending points of the input lines.
- ▷ **ColEndIn** (input_control) ..... line.end.x-array ⇝ *integer*
  Column coordinates of the ending points of the input lines.
- ▷ **Num** (input_control) ..... integer ⇝ *integer*
  (Maximum) desired number of output lines.
  Default: 10
- ▷ **RowBeginOut** (output_control) ..... line.begin.y-array ⇝ *integer*
  Row coordinates of the starting points of the output lines.
- ▷ **ColBeginOut** (output_control) ..... line.begin.x-array ⇝ *integer*
  Column coordinates of the starting points of the output lines.
- ▷ **RowEndOut** (output_control) ..... line.end.y-array ⇝ *integer*
  Row coordinates of the ending points of the output lines.
- ▷ **ColEndOut** (output_control) ..... line.end.x-array ⇝ *integer*
  Column coordinates of the ending points of the output lines.

Result

The operator select_lines_longest returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

sobel_amp, edges_image, threshold, hysteresis_threshold, split_skeleton_region, split_skeleton_lines

Possible Successors

set_line_width, disp_line

Alternatives

line_orientation, line_position, select_lines, partition_lines

See also

select_lines, partition_lines, detect_edge_segments, select_shape

Module

Foundation

update_kalman (: : FileName, DimensionIn, ModelIn, MeasurementIn : DimensionOut, ModelOut, MeasurementOut)

Read an update file of a Kalman filter.

update_kalman is obsolete and is only provided for reasons of backward compatibility.

The operator update_kalman reads the update file FileName of a Kalman filter. Kalman filters return an estimate of the current state (or even the prediction of a future state) of a discrete, stochastically disturbed, linear system.

A Kalman filtering is based on a mathematical model of the system to be examined which at any point in time has the following characteristics:

Model parameter: transition matrix A, control matrix G including the controller output u and the measurement matrix C

Model stochastic: system-error covariance matrix Q, system-error - measurement-error covariance matrix L and measurement-error covariance matrix R

Measurement vector: y

History of the system: extrapolation vector $\hat{x}$ and extrapolation-error covariance matrix $\hat{P}$

Many systems do not need entries "from outside" and therefore G and u can be dropped. Further, system errors and measurement errors are normally not correlated (L is dropped). Some of the characteristics mentioned above may change dynamically (from one iteration to the next). The operator update_kalman serves to modify parts of the system according to an update file (ASCII) with the following structure (see also read_kalman):

Dimension row
+ content row
+ matrix A
+ matrix C
+ matrix Q
+ matrix G + vector u
+ matrix L
+ matrix R

The dimension row thereby has the following form:

n = <integer> m = <integer> p = <integer>

whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members (see also DimensionIn / DimensionOut). The maximal dimension will hereby be limited by a system constant (= 30 for the time being). As in this case changes should take effect at a valid model, the dimensions n and m are invariant (and will only be indicated for purposes of control).

The content row has the following form:

$$A * C * Q * G * u * L * R*$$

and describes the further content of the file. Instead of '$*$', '+' (= parameter is available) respectively '-' (= parameter is missing) has to be set. In contrast to description files for read_kalman, the system description needs not be complete in this case. Only those parts of the system which are changed must be indicated. The indication of estimated values is unnecessary, as these values must stem from the latest filtering according to the structure of the filter.

$(r \times s)$ matrices will be stored in row-major order in the following form:

$$\begin{array}{c} < comment, i.e.\ string > \\ < a_{11} > \quad < a_{12} > \quad \cdots \quad < a_{1s} > \\ \vdots \qquad\qquad \ddots \qquad \vdots \\ < a_{r1} > \quad < a_{r2} > \quad \cdots \quad < a_{rs} > \end{array}$$

(the spaces/line feed characters can be chosen at will),

vectors will be stored correspondingly in the following form:

$$< comment, i.e.\ string >$$
$$< a_1 > \quad \cdots \quad < a_k >$$

The following parameter values of the operator `read_kalman` will be changed:

DimensionIn / DimensionOut: These parameters include the dimensions of the state vector, measurement vector and controller vector and therefore are vectors $[n, m, p]$, whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members. n and m are invariant for a given system, i.e. they must not differ from corresponding input values of the update file. For a system without without influence "from outside" $p = 0$.

ModelIn / ModelOut: These parameters include the lined up matrices (vectors) A, C, Q, G, u and if necessary L which have been stored in row-major order. `ModelIn` / `ModelOut` therefore are vectors of the length $n \times n + n \times m + n \times n + n \times p + p[+n \times m]$. The last summand is dropped if system errors and measurement errors are not correlated, i.e. no value has been set for L.

MeasurementIn / MeasurementOut: These parameters include the matrix R stored in row-major order, and therefore are vectors of the dimension $m \times m$.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Update file for a Kalman filter.
Default: 'kalman.updt'

▷ **DimensionIn** (input_control) . . . . . integer-array ⇝ *integer*
The dimensions of the state vector, measurement vector and controller vector.
Default: [3,1,0]
Value range: $0 \leq$ `DimensionIn` ≤ 30

▷ **ModelIn** (input_control) . . . . . real-array ⇝ *real*
The lined up matrices A,C,Q, possibly G and u, and if necessary L which all have been stored in row-major order.
Default: [1.0,1.0,0.5,0.0,1.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]
Value range: $0.0 \leq$ `ModelIn` ≤ 10000.0

▷ **MeasurementIn** (input_control) . . . . . real-array ⇝ *real*
The matrix R stored in row-major order.
Default: [1,2]
Value range: $0.0 \leq$ `MeasurementIn` ≤ 10000.0

▷ **DimensionOut** (output_control) . . . . . integer-array ⇝ *integer*
The dimensions of the state vector, measurement vector and controller vector.

▷ **ModelOut** (output_control) . . . . . real-array ⇝ *real*
The lined up matrices A,C,Q, possibly G and u, and if necessary L which all have been stored in row-major order.

▷ **MeasurementOut** (output_control) . . . . . real-array ⇝ *real*
The matrix R stored in row-major order.

Example

```
* The following values are describing the system
*
* DimensionIn   = [3,1,0]
* ModelIn       = [1.0,1.0,0.5,0.0,1.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,
*                  54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]
* MeasurementIn = [1,2]
*
* An example of the Updatefile:
*
* n=3 m=1 p=0
```

```
* A+C-Q-G-u-L-R-
* transitions at time t=15:
* 2 1 1
* 0 2 2
* 0 0 2
*
* the results of update_kalman:
*
* DimensionOut   = [3,1,0]
* ModelOut       = [2.0,1.0,1.0,0.0,2.0,2.0,0.0,0.0,2.0,1.0,0.0,0.0,
*                         54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]
* MeasurementOut = [1.2]
```

Result

If the update file is readable and correct, the operator `update_kalman` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`filter_kalman`

See also

`read_kalman`, `filter_kalman`

Module

Foundation

17.16 XLD

```
union_straight_contours_histo_xld ( Contours : UnionContours,
    SelectedContours : RefLineStartRow, RefLineStartColumn,
    RefLineEndRow, RefLineEndColumn, Width, MaxWidth,
    FilterSize : HistoValues )
```

Compute the union of neighboring straight contours that have a similar distance from a given line.

`union_straight_contours_histo_xld` is obsolete and is only provided for reasons of backward compatibility. New applications should use the operators of the chapter 1D Measuring instead.

`union_straight_contours_histo_xld` merges neighboring XLD contours `Contours` if certain criteria are fulfilled.

The maximum and minimum distances of the contours to a given reference line are calculated. From this distances a histogram is created. If the histogram should be smoothed, `FilterSize` must be greater than one. Afterwards, the resulting histogram is divided into ranges (from minima to minima). Contours which lie in the same range are concatenated to a new contour. If the width of the range is greater than `MaxWidth`, all contours in this range are ignored (removed). If a contour lies in two ranges or more it is ignored, too. If there are parallel contours, there is a risk of merging neighboring contours.

The parameters of the regression lines are calculated newly for merged contours.

The resulting contours cannot be displayed.

Attention

Before contours can be united by `union_straight_contours_histo_xld`, the parameters of the regression lines to the contours must be calculated by calling `regress_contours_xld`. Note further that already closed contours are not considered for a union anymore.

Parameters

▷ **Contours** (input_object) . . . . . xld_cont-array ⇝ *object*
Input XLD contours.

▷ **UnionContours** (output_object) . . . . . xld_cont-array ⇝ *object*
Output XLD contours.

▷ **SelectedContours** (output_object) . . . . . xld_cont-array ⇝ *object*
Output XLD contours.

▷ **RefLineStartRow** (input_control) . . . . . line.begin.y ⇝ *integer*
y coordinate of the starting point of the reference line.
Default: 0

▷ **RefLineStartColumn** (input_control) . . . . . line.begin.x ⇝ *integer*
x coordinate of the starting point of the reference line.
Default: 0

▷ **RefLineEndRow** (input_control) . . . . . line.end.y ⇝ *integer*
y coordinate of the endpoint of the reference line.
Default: 0

▷ **RefLineEndColumn** (input_control) . . . . . line.end.x ⇝ *integer*
x coordinate of the endpoint of the reference line.
Default: 0

▷ **Width** (input_control) . . . . . integer ⇝ *integer*
Maximum distance.
Default: 1

▷ **MaxWidth** (input_control) . . . . . integer ⇝ *integer*
Maximum width between two minima.
Default: 1

▷ **FilterSize** (input_control) . . . . . integer ⇝ *integer*
Size of smoothing filter
Default: 1
Value range: $1 \leq$ `FilterSize` ≤ 63

▷ **HistoValues** (output_control) . . . . . integer-array ⇝ *integer*
Output values of histogram.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

Chapter 18

Matching

18.1 Correlation-Based

clear_ncc_model (: : ModelID :)

Free the memory of an NCC model.

The operator clear_ncc_model frees the memory of an NCC model that was created by create_ncc_model. After calling clear_ncc_model, the model can no longer be used. The handle ModelID becomes invalid.

Parameters

▷ **ModelID** (input_control) ..... ncc_model ⇝ *handle*
Handle of the model.

Result

If the handle of the model is valid, the operator clear_ncc_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_ncc_model, read_ncc_model, write_ncc_model, find_ncc_model, find_ncc_models

Module

Matching

create_ncc_model (Template : : NumLevels, AngleStart, AngleExtent, AngleStep, Metric : ModelID)

Prepare an NCC model for matching.

The operator create_ncc_model prepares a template, which is passed in the image Template, as an NCC model used for matching using the normalized cross correlation (NCC). The ROI of the model is passed as the domain of Template.

The model is generated using multiple image pyramid levels at multiple rotations on each level and is stored in memory. The output parameter ModelID is a handle for this model, which is used in subsequent calls to find_ncc_model.

The number of pyramid levels is determined with the parameter NumLevels. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, NumLevels must be chosen such that the model is still recognizable and contains a sufficient number of points (at least eight) on the highest pyramid level. This can be checked using the domains of the output images of gen_gauss_pyramid. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, create_ncc_model returns an error message. If NumLevels is set to *'auto'* or *0*, create_ncc_model determines the number of pyramid levels automatically. The automatically computed number of pyramid levels can be queried using get_ncc_model_params. In rare cases, it might happen that create_ncc_model determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for MinScore in find_ncc_model in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in find_ncc_model may increase. In these cases, the number of pyramid levels should be selected by inspecting the output of gen_gauss_pyramid. Here, Mode = *'constant'* and Scale = *0.5* should be used.

The parameters AngleStart and AngleExtent determine the range of possible rotations, in which the model can occur in the image. Note that the model can only be found in this range of angles by find_ncc_model. The parameter AngleStep determines the step length within the selected range of angles. Hence, if subpixel accuracy is not specified in find_ncc_model, this parameter specifies the accuracy that is achievable for the angles in find_ncc_model. AngleStep should be chosen based on the size of the object. Smaller models do not possess many different discrete rotations in the image, and hence AngleStep should be chosen larger for smaller models. If AngleExtent is not an integer multiple of AngleStep, AngleStep is modified accordingly. To ensure a sampling of the range of possible rotations that is independent of the given AngleStart, the range of possible rotations is modified as follows: If there is no positive integer value n such that AngleStart plus n times AngleStep is exactly 0.0, AngleStart is decreased by up to AngleStep and AngleExtent is increased by AngleStep.

The model is pre-generated for the selected angle range and stored in memory. The memory required to store the model is proportional to the number of angle steps and the number of points in the model. Hence, if AngleStep is too small or AngleExtent too big, it may happen that the model no longer fits into the (virtual) memory. In this case, either AngleStep must be enlarged or AngleExtent must be reduced. In any case, it is desirable that the model completely fits into the main memory, because this avoids paging by the operating system, and hence the time to find the object will be much smaller. Since angles can be determined with subpixel resolution by find_ncc_model, AngleStep $\geq$ 1 can be selected for models of a diameter smaller than about 200 pixels. If AngleStep = *'auto'* or *0* is selected, create_ncc_model automatically determines a suitable angle step length based on the size of the model. The automatically computed angle step length can be queried using get_ncc_model_params.

The parameter Metric determines the conditions under which the model is recognized in the image. If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background. If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_ncc_model will increase slightly in this case.

The center of gravity of the domain (region) of the model image Template is used as the origin (reference point) of the model. A different origin can be set with set_ncc_model_origin.

Parameters

▷ **Template** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: NumLevels ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the pattern.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **AngleStep** (input_control) . . . . . angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: AngleStep ∈ {'auto', 0.0, 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep >= 0 && AngleStep <= pi / 16

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity'}

▷ **ModelID** (output_control) . . . . . ncc_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_ncc_model returns the value 2 (H_MSG_TRUE). If the parameter NumLevels are chosen such that the model contains too few points, the error 8506 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

draw_region, reduce_domain, threshold

Possible Successors

find_ncc_model, get_ncc_model_params, clear_ncc_model, write_ncc_model, set_ncc_model_origin, set_ncc_model_param, find_ncc_models

Alternatives

create_generic_shape_model

Module

Matching

deserialize_ncc_model (: : SerializedItemHandle : ModelID)

Deserialize an NCC model.

deserialize_ncc_model deserializes an NCC model, that was serialized by serialize_ncc_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized NCC model model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created NCC model with the handle ModelID.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **ModelID** (output_control) . . . . . ncc_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator `deserialize_ncc_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_ncc_model`

Possible Successors

`find_ncc_model`

See also

`create_ncc_model`, `clear_ncc_model`

Module

Matching

determine_ncc_model_params (Template : : NumLevels, AngleStart, AngleExtent, Metric, Parameters : ParameterName, ParameterValue)

Determine the parameters of an NCC model.

`determine_ncc_model_params` determines certain parameters of an NCC model automatically from the model image `Template`. The parameters to be determined can be specified with `Parameters`. `determine_ncc_model_params` can be used to determine the same parameters that are determined automatically when the respective parameter in `create_ncc_model` is set to *'auto'*: the number of pyramid levels (`Parameters` = *'num_levels'*) and/or the angle step length (`Parameters` = *'angle_step'*). By passing a tuple of the above values in `Parameters`, an arbitrary combination of these parameters can be determined. If all of the above parameters should be determined, the value *'all'* can be passed.

`determine_ncc_model_params` is mainly useful to determine the above parameters before creating the model, e.g., in an interactive system, which makes suggestions for these parameters to the user, but enables the user to modify the suggested values.

The automatically determined parameters are returned as a name-value pair in `ParameterName` and `ParameterValue`. The parameter names in `ParameterName` are identical to the names in `Parameters`, where, of course, the value *'all'* is replaced by the actual parameter names.

The input parameters (`NumLevels`, `AngleStart`, `AngleExtent`, and `Metric`) have the same meaning as in `create_ncc_model`. The description of these parameters can be looked up with this operator. These parameters are used by `determine_ncc_model_params` to calculate the parameters to be determined in the same manner as in `create_ncc_model`.

Note that in `determine_ncc_model_params` the input parameter `NumLevels` can also be determined automatically. If this parameter should not be determined automatically, i.e., the name is not passed in `Parameters`, the parameter must contain a valid value and must not be set to *'auto'*. In contrast, if the maximum number of pyramid levels is to be determined automatically, i.e., `Parameters` contains the value *'num_levels'*, you can let HALCON determine a suitable value and at the same time specify an upper boundary:

If the maximum number of pyramid levels should be specified in advance, the input parameter `NumLevels` can be set to the particular value. If in this case `Parameters` contains the value *'num_levels'*, the computed number of pyramid levels is at most `NumLevels`. If `NumLevels` is set to *'auto'* or *0*, the number of pyramid levels is determined without restrictions as large as possible.

Parameters

- ▷ **Template** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image whose domain will be used to create the model.
- ▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer* / string
  Maximum number of pyramid levels.
  Default: 'auto'
  List of values: NumLevels ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
- ▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
  Smallest rotation of the model.
  Default: -0.39
  Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}
- ▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
  Extent of the rotation angles.
  Default: 0.79
  Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39}
  Restriction: AngleExtent >= 0
- ▷ **Metric** (input_control) . . . . . string ⇝ *string*
  Match metric.
  Default: 'use_polarity'
  List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity'}
- ▷ **Parameters** (input_control) . . . . . string(-array) ⇝ *string*
  Parameters to be determined automatically.
  Default: 'all'
  List of values: Parameters ∈ {'all', 'num_levels', 'angle_step'}
- ▷ **ParameterName** (output_control) . . . . . string-array ⇝ *string*
  Name of the automatically determined parameter.
- ▷ **ParameterValue** (output_control) . . . . . number-array ⇝ *real* / integer
  Value of the automatically determined parameter.

Result

If the parameters are valid, the operator determine_ncc_model_params returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

create_ncc_model

See also

find_ncc_model

Module

Matching

find_ncc_model (Image : : ModelID, AngleStart, AngleExtent, MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels : Row, Column, Angle, Score)

Find the best matches of an NCC model in an image.

The operator find_ncc_model finds the best NumMatches instances of the NCC model ModelID in the input image Image. The model must have been created previously by calling create_ncc_model or read_ncc_model.

The position and rotation of the found instances of the model is returned in Row, Column, and Angle. Additionally, the score of each found instance is returned in Score.

If NCC finds no suitable match or the matching scores are to low, the search should be performed using a different matching method (see, e.g., "Solution Guide II-B - Matching").

Input parameters in detail

Image and its domain: The domain of the image Image determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the NCC model with create_ncc_model. A different origin set with set_ncc_model_origin is not taken into account here. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than MinScore (see below). Note that rounding effects can cause matches to lie up to $0.5 * 2^{\text{NumLevels}-1}$ pixels outside the image.

AngleStart and AngleExtent: The parameters AngleStart and AngleExtent determine the range of rotations for which the model is searched. If necessary, the range of rotations is clipped to the range given when the model was created with create_ncc_model.

MinScore: The parameter MinScore determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger MinScore is chosen, the faster the search is.

NumMatches: The maximum number of instances to be found can be determined with NumMatches. If more than NumMatches instances with a score greater than MinScore are found in the image, only the best NumMatches instances are returned. If fewer than NumMatches are found, only that number is returned, i.e., the parameter MinScore takes precedence over NumMatches. If all model instances exceeding MinScore in the image should be found, NumMatches must be set to *0*.

MaxOverlap: If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. If the model has repeating structures it may happen that multiple instances with identical rotations are found at similar positions in the image. The parameter MaxOverlap determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than MaxOverlap only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see smallest_rectangle2) of the found instances. If MaxOverlap $= 0$, the found instances may not overlap at all, while for MaxOverlap $= 1$ all instances are returned.

SubPixel: The parameter SubPixel determines whether the instances should be extracted with subpixel accuracy. If SubPixel is set to *'false'*, the model's pose is only determined with pixel accuracy and the angle resolution that was specified with create_ncc_model. If SubPixel is set to *'true'*, the position as well as the rotation are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves a high accuracy. Hence, SubPixel should usually be set to *'true'*. Note that the subpixel accurate determination of the model's pose is only possible if the found instance lies at least 2 pixels away from the image border of the lowest used pyramid level. If the instance lies closer to the image border, its pose is only determined with pixel accuracy and the angle resolution that was specified with create_ncc_model, even if SubPixel is set to *'true'*.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the NCC model was created with create_ncc_model. If NumLevels is set to *0*, the number of pyramid levels specified in create_ncc_model is used.

In certain cases, the number of pyramid levels that was determined automatically with, for example, create_ncc_model may be too high. The consequence may be that some matches that may have a high final score are rejected on the highest pyramid level and thus are not found. Instead of setting MinScore to a very low value to find all matches, it may be better to query the value of NumLevels with get_ncc_model_params and then use a slightly lower value in find_ncc_model. This approach is often better regarding the speed and robustness of the matching.

Optionally, NumLevels can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of *[4,2]* for NumLevels means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a

reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

Output parameters in detail

Row, Column and Angle: The position and rotation of the found instances of the model is returned in Row, Column, and Angle. The coordinates Row and Column are related to the position of the origin of the model in the search image. However, Row and Column do not exactly correspond to this position. Instead, find_ncc_model returns slightly modified values that are optimized for creating a transformation matrix, that can be used for alignment or visualization of the model. (This has to do with the way HALCON transforms iconic objects, see affine_trans_pixel). The example below shows how to create the transformation matrix for alignment and how to calculate the exact coordinates of the found matches.

By default, the origin is the center of gravity of the domain (region) of the image that was used to create the NCC model with create_ncc_model. A different origin can be set with set_ncc_model_origin.

Score: Additionally, the score of each found instance is returned in Score. The score is the normalized cross correlation of the template $t(r,c)$ and the image $i(r,c)$:

$$\mathrm{ncc}(r,c) = \frac{1}{n} \sum_{(u,v) \in R} \frac{t(u,v) - m_t}{\sqrt{s_t^2}} \cdot \frac{i(r+u, c+v) - m_i(r,c)}{\sqrt{s_i^2(r,c)}}$$

Here, n denotes the number of points in the template, R denotes the domain (ROI) of the template, m_t is the mean gray value of the template

$$m_t = \frac{1}{n} \sum_{(u,v) \in R} t(u,v)$$

s_t^2 is the variance of the gray values of the template

$$s_t^2 = \frac{1}{n} \sum_{(u,v) \in R} (t(u,v) - m_t)^2$$

$m_i(r,c)$ is the mean gray value of the image at position (r,c) over all points of the template (i.e., the template points are shifted by (r,c))

$$m_i(r,c) = \frac{1}{n} \sum_{(u,v) \in R} i(r+u, c+v)$$

and $s_i^2(r,c)$ is the variance of the gray values of the image at position (r,c) over all points of the template

$$s_i^2(r,c) = \frac{1}{n} \sum_{(u,v) \in R} (i(r+u, c+v) - m_i(r,c))^2$$

The NCC measures how well the template and image correspond at a particular point (r,c). It assumes values between -1 and 1. The larger the absolute value of the correlation, the larger the degree of correspondence between the template and image. A value of 1 means that the gray values in the image are a linear transformation of the gray values in the template:

$$i(r+u, c+v) = at(u,v) + b$$

where $a > 0$. Similarly, a value of -1 means that the gray values in the image are a linear transformation of the gray values in the template with $a < 0$. Hence, in this case the template occurs with a reversed polarity in the image. Because of the above property, the NCC is invariant to linear illumination changes.

The NCC as defined above is used if the NCC model has been created with Metric = *'use_polarity'*. If the model has been created with Metric = *'ignore_global_polarity'*, the absolute value of $\mathrm{ncc}(r,c)$ is used as the score.

Specifying a timeout

Using the operator set_ncc_model_param you can specify a *'timeout'* for find_ncc_model. If find_ncc_model reaches this *'timeout'*, it terminates without results and returns the error code 9400 (H_ERR_TIMEOUT).

Visualization of the results

To display the results found by correlation-based matching, we highly recommend the usage of the procedure dev_display_ncc_matching_results.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Attention

find_ncc_model can be partially executed on OpenCL devices that support the cl_khr_global_int32_base_atomics OpenCL extension. Only the initial search for the model in the topmost pyramid level is done on the OpenCL device, while the tracking of matches is done on the host CPU. If the domain of the image to search in is substantially smaller than the size of the image, use crop_domain to reduce the amount of data that needs to be copied from the OpenCL device to the host CPU. Note that find_ncc_model using OpenCL may run either substantially faster or slower depending on a wide number of factors, so the only way to tell if using OpenCL is beneficial is by testing with images from the actual application.

Furthermore, note that the internally used memory increases with the number of used threads.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image in which the model should be found.

▷ **ModelID** (input_control) . . . . . ncc_model ⇝ *handle*
Handle of the model.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **MinScore** (input_control) . . . . . real ⇝ *real*
Minimum score of the instances of the model to be found.
Default: 0.8
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer ⇝ *integer*
Number of instances of the model to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real ⇝ *real*
Maximum overlap of the instances of the model to be found.
Default: 0.5
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **SubPixel** (input_control) . . . . . string ⇝ *string*
Subpixel accuracy.
Default: 'true'
List of values: SubPixel ∈ {'false', 'true'}

▷ **NumLevels** (input_control) ..... integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Row** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the found instances of the model.

▷ **Column** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the found instances of the model.

▷ **Angle** (output_control) ..... angle.rad-array ⇝ *real*
Rotation angle of the found instances of the model.

▷ **Score** (output_control) ..... real-array ⇝ *real*
Score of the found instances of the model.

Example

```
create_ncc_model (TemplateImage, 'auto', rad(-45), rad(90), 'auto', \
                  'use_polarity', ModelID)
find_ncc_model (SearchImage, ModelID, rad(-45), rad(90), 0.7, 1, \
                0.5, 'true', 0, Row, Column, Angle, Score)
* Create transformation matrix
vector_angle_to_rigid (0, 0, 0, Row, Column, Angle, HomMat2D)
* Calculate true position of the model origin in the search image
affine_trans_pixel (HomMat2D, 0, 0, RowObject, ColumnObject)
* display the results
dev_display_ncc_matching_results (ModelID, 'red', Row, Column, \
                                  Angle, 0)
```

Result

If the parameter values are correct, the operator find_ncc_model returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

create_ncc_model, read_ncc_model, set_ncc_model_origin

Possible Successors

clear_ncc_model

Alternatives

find_generic_shape_model

Module

Matching

find_ncc_models (Image : : ModelIDs, AngleStart, AngleExtent, MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels : Row, Column, Angle, Score, Model)

Find the best matches of multiple NCC models.

The operator find_ncc_models finds the best NumMatches instances of the NCC models that are passed in the tuple ModelIDs in the input Image. The models must have been created previously by calling

create_ncc_model or read_ncc_model. In contrast to find_ncc_model, multiple models can be searched in the same image in one call.

If NCC finds no suitable match or the matching scores are to low, the search should be performed using a different matching method (see, e.g., "Solution Guide II-B - Matching").

Characteristics of the parameter semantics

Compared to find_ncc_model, the semantics of all input parameters have changed to some extent. All input parameters must either contain one element, in which case the parameter is used for all models, or they must contain the same number of elements as ModelIDs, in which case each parameter element refers to the corresponding element in ModelIDs. (NumLevels may also contain either two or twice the number of elements as ModelIDs). More details can be found below in the sections containing information for the respective parameters. Note that a call to find_ncc_models with multiple values for ModelIDs, NumMatches and MaxOverlap has the same effect as multiple independent calls to find_ncc_model with the respective parameters. However, a single call to find_ncc_models is considerably more efficient.

Input parameters in detail

Image and its domain: The domain of the image Image determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the NCC model with create_ncc_model. A different origin set with set_ncc_model_origin is not taken into account. The model has to lie completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than MinScore (see below). Note that rounding effects can cause matches to lie up to $0.5 * 2^{\text{NumLevels}-1}$ pixels outside the image.

The input Image can contain a single image object or an image object array containing multiple image objects. If Image contains a single image object, its domain is used as the region of interest for all models in ModelIDs. If Image contains multiple image objects, each domain is used as the region of interest for the corresponding model in ModelIDs. In this case, the images have to be identical except for their domains, i.e., Image cannot be constructed in an arbitrary manner using concat_obj, but must be created from the same image using add_channels or equivalent calls. If this is not the case, an error message is returned.

AngleStart and AngleExtent: The parameters AngleStart and AngleExtent determine the range of rotations for which the model is searched. If necessary, the range of rotations is clipped to the range given when the model was created with create_ncc_model.

Furthermore, it should be noted that in some cases instances with a rotation that is slightly outside the specified range of rotations are found. This may happen if the specified range of rotations is smaller than the range given when the model was created.

MinScore: The parameter MinScore determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger MinScore is chosen, the faster the search is. If the model can be expected never to be occluded in the images, MinScore may be set as high as 0.8 or even 0.9. If the matches are not tracked to the lowest pyramid level (see NumLevels), it might happen that instances with a score slightly below MinScore are found.

NumMatches: The maximum number of instances to be found can be determined with NumMatches. If more than NumMatches instances with a score greater than MinScore are found in the image, only the best NumMatches instances are returned. If fewer than NumMatches are found, only that number is returned, i.e., the parameter MinScore takes precedence over NumMatches. If all model instances exceeding MinScore in the image should be found, NumMatches must be set to *0*.

If NumMatches contains one element, find_ncc_models returns the best NumMatches instances of the model irrespective of the type of the model. If, for example, two models are passed in ModelIDs and NumMatches $= 2$ is selected, it can happen that two instances of the first model and no instances of the second model, one instance of the first model and one instance of the second model, or no instances of the first model and two instances of the second model are returned. If, on the other hand, NumMatches contains multiple values, the number of instances returned of the different models corresponds to the number specified in the respective entry in NumMatches. If, for example, NumMatches $= [1, 1]$ is selected, one instance of the first model and one instance of the second model is returned.

MaxOverlap: If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter MaxOverlap determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than MaxOverlap, only

the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see smallest_rectangle2) of the found instances. If MaxOverlap = 0, the found instances may not overlap at all, while for MaxOverlap = 1 all instances are returned.

If a single value is passed in MaxOverlap, the overlap is computed for all found instances of the different models, irrespective of the model type, i.e., instances of the same or of different models that overlap too much are eliminated. If, on the other hand, multiple values are passed in MaxOverlap, the overlap is only computed for found instances of the model that have the same model type, i.e., only instances of the same model that overlap too much are eliminated. In this mode, models of different types may overlap completely.

SubPixel: The parameter SubPixel determines whether the instances should be extracted with subpixel accuracy. If SubPixel is set to *'false'* the model's pose is only determined with pixel accuracy and the angle resolution that was specified with create_ncc_model. If SubPixel is set to *'true'*, the position as well as the rotation are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the NCC model was created with create_ncc_model. If NumLevels is set to *0*, the number of pyramid levels specified in create_ncc_model is used.

In certain cases, the number of pyramid levels that was determined automatically with, for example, create_ncc_model may be too high. The consequence may be that some matches that may have a high final score are rejected on the highest pyramid level and thus are not found. Instead of setting MinScore to a very low value to find all matches, it may be better to query the value of NumLevels with get_ncc_model_params and then use a slightly lower value in find_ncc_models. This approach is often better regarding the speed and robustness of the matching.

Optionally, NumLevels can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of *[4,2]* for NumLevels means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

If the lowest pyramid level is specified separately for each model, NumLevels must contain twice the number of elements as ModelIDs. In this case, the number of pyramid levels and the lowest pyramid level must be specified interleaved in NumLevels. If, for example, two models are specified in ModelIDs, the number of pyramid levels is 5 for the first model and 4 for the second model, and the lowest pyramid level is 2 for the first model and 1 for the second model, NumLevels = $[5, 2, 4, 1]$ must be selected. If exactly two models are specified in ModelIDs, a special case occurs. If in this case the lowest pyramid level is to be specified, the number of pyramid levels and the lowest pyramid level must be specified explicitly for both models, even if they are identical, because specifying two values in NumLevels is interpreted as the explicit specification of the number of pyramid levels for the two models.

Output parameters in detail

Row, Column and Angle: The position and rotation of the found instances of the model is returned in Row, Column, and Angle. The coordinates Row and Column are the coordinates of the origin of the NCC model in the search image. By default, the origin is the center of gravity of the domain (region) of the image that was used to create the NCC model with create_ncc_model. A different origin can be set with set_ncc_model_origin.

Note that the coordinates Row and Column do not exactly correspond to the position of the model in the search image. Thus, you cannot directly use them. Instead, the values are optimized for creating the transformation matrix with which you can use the results of the matching for various tasks, e.g., to align ROIs for other processing steps. The example given for find_ncc_model shows how to create this matrix and use it to display the model at the found position in the search image and to calculate the exact coordinates.

Note also that for visualizing the model at the found position, also the procedure dev_display_ncc_matching_results can be used.

Score: Additionally, the score of each found instance is returned in Score.

Model: The type of the found instances of the models is returned in Model. The elements of Model are indices into the tuple ModelIDs, i.e., they can contain values from 0 to |ModelIDs| − 1. Hence, a value of 0 in an element of Model corresponds to an instance of the first model in ModelIDs.

Specifying a timeout

Using the operator set_ncc_model_param you can specify a *'timeout'* for find_ncc_models. If the NCC models referenced by ModelIDs hold different values for *'timeout'*, find_ncc_models uses the smallest one. If find_ncc_models reaches this *'timeout'*, it terminates without results and returns the error code 9400 (H_ERR_TIMEOUT).

Visualization of the results

To display the results found by correlation-based matching, we highly recommend the usage of the procedure dev_display_ncc_matching_results.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Attention

Note that the internally used memory increases with the number of used threads.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Input image in which the model should be found.

▷ **ModelIDs** (input_control) . . . . . ncc_model(-array) ⇝ *handle*
Handle of the models.

▷ **AngleStart** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Smallest rotation of the models.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the models to be found.
Default: 0.8
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of instances of the models to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real(-array) ⇝ *real*
Maximum overlap of the instances of the models to be found.
Default: 0.5
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **SubPixel** (input_control) . . . . . string(-array) ⇝ *string*
Subpixel accuracy if not equal to *'none'*.
Default: 'true'
List of values: SubPixel ∈ {'false', 'true'}

▷ **NumLevels** (input_control) ..... integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Row** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the found instances of the models.

▷ **Column** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the found instances of the models.

▷ **Angle** (output_control) ..... angle.rad-array ⇝ *real*
Rotation angle of the found instances of the models.

▷ **Score** (output_control) ..... real-array ⇝ *real*
Score of the found instances of the models.

▷ **Model** (output_control) ..... integer-array ⇝ *integer*
Index of the found instances of the models.

Example

```
read_image (Image, 'pcb_focus/pcb_focus_telecentric_061')
gen_rectangle1 (ROI_0, 236, 241, 313, 321)
gen_circle (ROI_1, 281, 653, 41)
reduce_domain (Image, ROI_0, ImageReduced1)
reduce_domain (Image, ROI_1, ImageReduced2)
create_ncc_model (ImageReduced1, 'auto', rad(-45), rad(90), 'auto', \
                  'use_polarity', ModelID1)
create_ncc_model (ImageReduced2, 'auto', rad(-45), rad(90), 'auto', \
                  'use_polarity', ModelID2)
ModelIDs:=[ModelID1, ModelID2]
find_ncc_models (Image, ModelIDs, rad(-45), rad(90), 0.7, [1,1], 0.5, \
                 'true', 0, Row, Column, Angle, Score, Model)
dev_display_ncc_matching_results (ModelIDs, 'red', Row, Column, \
                                  Angle, Model)
```

Result

If the parameter values are correct, the operator find_ncc_models returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

create_ncc_model, read_ncc_model, set_ncc_model_origin

Possible Successors

clear_ncc_model

Alternatives

find_generic_shape_model

Module

Matching

get_ncc_model_origin (: : ModelID : Row, Column)

Return the origin (reference point) of an NCC model.

The operator get_ncc_model_origin returns the origin (reference point) of the NCC model ModelID. The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the NCC model with create_ncc_model. Hence, an origin of (0,0) means that the center of gravity of the domain of the model image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity.

Parameters

- ▷ **ModelID** (input_control) . . . . . ncc_model ⇝ *handle*
  Handle of the model.
- ▷ **Row** (output_control) . . . . . point.y ⇝ *real*
  Row coordinate of the origin of the NCC model.
- ▷ **Column** (output_control) . . . . . point.x ⇝ *real*
  Column coordinate of the origin of the NCC model.

Result

If the handle of the model is valid, the operator get_ncc_model_origin returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_ncc_model, read_ncc_model, set_ncc_model_origin

Possible Successors

find_ncc_model

See also

area_center

Module

Matching

get_ncc_model_params (: : ModelID : NumLevels, AngleStart, AngleExtent, AngleStep, Metric)

Return the parameters of an NCC model.

The operator get_ncc_model_params returns the parameters of the NCC model ModelID that were used to create it using create_ncc_model. In particular, this output can be used to check the parameters NumLevels and AngleStep if they were determined automatically during the model creation with create_ncc_model. Furthermore, this output can be used to check the parameters if the model was read with read_ncc_model.

Parameters

- ▷ **ModelID** (input_control) . . . . . ncc_model ⇝ *handle*
  Handle of the model.
- ▷ **NumLevels** (output_control) . . . . . integer ⇝ *integer*
  Number of pyramid levels.
- ▷ **AngleStart** (output_control) . . . . . angle.rad ⇝ *real*
  Smallest rotation of the pattern.
- ▷ **AngleExtent** (output_control) . . . . . angle.rad ⇝ *real*
  Extent of the rotation angles.
  Assertion: AngleExtent >= 0
- ▷ **AngleStep** (output_control) . . . . . angle.rad ⇝ *real*
  Step length of the angles (resolution).
  Assertion: AngleStep >= 0

▷ **Metric** (output_control) . . . . . string ⇝ *string*
Match metric.

Result

If the handle of the model is valid, the operator `get_ncc_model_params` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_ncc_model`, `read_ncc_model`

See also

`find_ncc_model`

Module

Matching

get_ncc_model_region (: ModelRegion : ModelID :)

Return the region used to create an NCC model.

The operator `get_ncc_model_region` returns the region used to create the NCC model `ModelID` in `ModelRegion`. This region can be used, for example, to visualize the found instances of the model in an image. It should be noted that the position of `ModelRegion` is normalized such that the reference point of the model (see `set_ncc_model_origin`) lies at the pixel position $(0, 0)$. Hence, the region simply needs to be translated to the found position in the image (and possibly rotated around this point). `get_ncc_model_region` ignores the value of the system parameter *'clip_region'*.

Parameters

▷ **ModelRegion** (output_object) . . . . . region ⇝ *object*
Model region of the NCC model.

▷ **ModelID** (input_control) . . . . . ncc_model ⇝ *handle*
Handle of the model.

Result

If the handle of the NCC model is valid, the operator `get_ncc_model_region` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_ncc_model`

Module

Matching

read_ncc_model (: : FileName : ModelID)

Read an NCC model from a file.

The operator read_ncc_model reads an NCC model, which has been written with write_ncc_model, from the file FileName. The default HALCON file extension for the NCC model is 'ncm'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .ncm

▷ **ModelID** (output_control) . . . . . ncc_model ⇝ *handle*
Handle of the model.

Result

If the file name is valid, the operator read_ncc_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

find_ncc_model, find_ncc_models

See also

create_ncc_model, clear_ncc_model

Module

Matching

serialize_ncc_model (: : ModelID : SerializedItemHandle)

Serialize an NCC model.

serialize_ncc_model serializes the data of an NCC model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_ncc_model is converted to a serialized item. The NCC model is defined by the handle ModelID. The serialized NCC model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_ncc_model.

Parameters

▷ **ModelID** (input_control) . . . . . ncc_model ⇝ *handle*
Handle of the model.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_ncc_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_ncc_model

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_ncc_model

Module

Matching

set_ncc_model_origin (: : ModelID, Row, Column :)

Set the origin (reference point) of an NCC model.

The operator set_ncc_model_origin sets the origin (reference point) of the NCC model ModelID to a new value. The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the NCC model with create_ncc_model. Hence, an origin of (0,0) means that the center of gravity of the domain of the model image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity.

Parameters

- ▷ **ModelID** (input_control) . . . . . ncc_model ⇝ *handle*
  Handle of the model.
- ▷ **Row** (input_control) . . . . . point.y ⇝ *real*
  Row coordinate of the origin of the NCC model.
- ▷ **Column** (input_control) . . . . . point.x ⇝ *real*
  Column coordinate of the origin of the NCC model.

Result

If the handle of the model is valid, the operator set_ncc_model_origin returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_ncc_model, read_ncc_model

Possible Successors

find_ncc_model, get_ncc_model_origin, find_ncc_models

See also

area_center

Module

Matching

set_ncc_model_param (: : ModelID, GenParamName, GenParamValue :)

Set selected parameters of the NCC model.

The operator set_ncc_model_param sets the selected parameters GenParamName in the NCC model ModelID. The following parameters can be modified:

'timeout'

Sets the maximum runtime of the operators used to find the NCC model ModelID (using find_ncc_model). This is especially useful in cases where a maximum cycle time has to be ensured. The *'timeout'* must be given in

milliseconds. The temporal accuracy depends on several factors including the size of the model, the speed of your computer, and the *'timer_mode'* set via set_system. Be aware that the runtime of the search increases by up to 10 percent with activated timeout. To disable the timeout you can either use a negative value or *'false'*.

Parameters

▷ **ModelID** (input_control) . . . . . ncc_model ⇝ *handle*
Handle of the model.

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Parameter names.
List of values: GenParamName ∈ {'timeout'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Parameter values.

Result

If the parameters are valid, the operator set_ncc_model_param returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_ncc_model

Possible Successors

find_ncc_model

Module

Matching

write_ncc_model (: : ModelID, FileName :)

Write an NCC model to a file.

The operator write_ncc_model writes an NCC model to the file FileName. The model can be read again with read_ncc_model. The default HALCON file extension for the NCC model is 'ncm'.

Parameters

▷ **ModelID** (input_control) . . . . . ncc_model ⇝ *handle*
Handle of the model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: .ncm

Result

If the file name is valid (write permission), the operator write_ncc_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

create_ncc_model

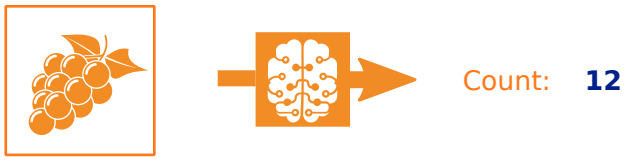
Module

Matching

18.2 Deep Counting

This chapter explains how to use Deep Counting.

Deep Counting is used to find objects in images and returns the number of objects.



A possible example for a Deep Counting application: Objects in an image are counted and the object quantity is returned.

Deep Counting uses user-defined templates to prepare a Deep Counting model. This model can be used to count objects that resemble the templates.

The general workflow is described in the following section.

General Workflow

This paragraph describes how to obtain the number from objects of a kind in images using a Deep Counting model. An application scenario can be seen in the HDevelop example `deep_counting_workflow.hdev`

1. Create a Deep Counting model by using
   - create_deep_counting_model.
2. Query the available DL devices for inference by using
   - query_available_dl_devices.
3. Set the model parameters, the used devices or augmentation parameters for the templates using
   - set_deep_counting_model_param.
4. Draw templates for the preparation of the model. This can be done using the procedure
   - draw_deep_counting_templates.

   The drawn templates are needed for the operator prepare_deep_counting_model.
5. Prepare the model for the inference using
   - prepare_deep_counting_model.

   Note that when changing parameters that influence the template creation, or when other templates should be used, prepare_deep_counting_model must be called again, before the model can be applied with apply_deep_counting_model.
6. Apply the model using the operator
   - apply_deep_counting_model.

The result will be saved in DeepCountingResult.

7. Visualize the Deep Counting results from DeepCountingResult using the procedure
   - dev_display_deep_counting_results.

apply_deep_counting_model (Image : : DeepCountingHandle : Count, DeepCountingResult)

Apply a Deep Counting model on a set of images for inference.

apply_deep_counting_model applies the Deep Counting model given by DeepCountingHandle on the tuple of input images Image. A tuple with the number of found instances per input image is returned in Count. Additional information about the counted instances is returned in DeepCountingResult, which contains a tuple of dictionaries, again one per input image. Note that templates of the objects to be counted must be set beforehand using prepare_deep_counting_model. Note that the inference time of apply_deep_counting_model depends on the number of defined templates in prepare_deep_counting_model.

Please see the chapter Matching / Deep Counting for further information.

The procedure dev_display_deep_count_results can be used to visualize the detected instances.

The settings *'min_score'* and *'max_overlap'* can be used to set the minimum similarity of the instances to the templates as well as the allowed overlap of instances. Details can be found in get_deep_counting_model_param.

The dictionary returned in DeepCountingResult contains additional information about the detected and counted object instances. It contains the following keys, where each key contains a tuple of values, one for each detection. If no instances were detected, i.e. the count is *0*, those tuples are empty.

'area':
Approximate area of the detected templates in pixels.

'row', *'column'*:
Approximate location in row and column coordinates of the instances in the input image.

'score':
Similarity score of the detected instances, i.e., their approximate similarity with the most similar template provided to prepare_deep_counting_model.

'template_index':
Index of the template with the highest similarity to the detected instance. This can be used to find out which templates were counted.

'angle', *'scale'*:
The angle and scale of the template with the highest similarity to the detected instance. If rotational or scale augmentations were enabled during the call of prepare_deep_counting_model, these values can be used to find out which rotation angle and scale creates a template that is most similar to the detected instance.

Attention

Deep Counting does not take into account whether an image has a reduced domain.

System requirements: To run this operator on GPU (see get_deep_counting_model_param), cuDNN and cuBLAS are required. For further details, please refer to the "Installation Guide", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods". Alternatively, this operator can also be run on CPU.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / real
Input image.

▷ **DeepCountingHandle** (input_control) ..... deep_counting ⇝ *handle*
Handle of the Deep Counting model.

▷ **Count** (output_control) ..... integer(-array) ⇝ *integer*
Number of counted objects.

▷ **DeepCountingResult** (output_control) ..... dict(-array) ⇝ *handle*
Tuple of result dictionaries.

Result

If the parameters are valid, the operator `apply_deep_counting_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`create_deep_counting_model`, `set_deep_counting_model_param`, `get_deep_counting_model_param`, `prepare_deep_counting_model`

See also

`prepare_deep_counting_model`

Module

Matching

`create_deep_counting_model` (: : `GenParamName`, `GenParamValue` : `DeepCountingHandle`)

Create a Deep Counting model for counting objects.

`create_deep_counting_model` creates a Deep Counting model and returns its handle in `DeepCountingHandle`.

Internally `create_deep_counting_model` builds a backbone based on the pretrained network *'pretrained_dl_classifier_resnet50.hdl'*, which must be available. See `read_dl_model` for details on where the file is searched.

Additional parameters of the Deep Counting model can be set using the generic parameters `GenParamName` and `GenParamValue`. See `get_deep_counting_model_param` for a list of parameters and their possible values. The parameters can either be set during model creation, or later using `set_deep_counting_model_param`.

Note that before using the returned Deep Counting model with `apply_deep_counting_model`, templates must be stored in the model using `prepare_deep_counting_model`.

Parameters

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Parameter names.
Default: []
List of values: `GenParamName` ∈ {'angle_start', 'angle_end', 'angle_step', 'device', 'max_overlap', 'min_score', 'scale_min', 'scale_max', 'scale_step'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
Parameter values.
Default: []

▷ **`DeepCountingHandle`** (output_control) . . . . . deep_counting ⇝ *handle*
Deep Counting model for counting objects.

Result

If the parameters are valid, the operator `create_deep_counting_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_deep_counting_model_param, get_deep_counting_model_param, prepare_deep_counting_model, apply_deep_counting_model

Alternatives

read_deep_counting_model

See also

get_deep_counting_model_param

Module

Matching

get_deep_counting_model_param (: : DeepCountingHandle, GenParamName : GenParamValue)

Return the parameters of a Deep Counting model.

The operator get_deep_counting_model_param returns the parameter values of GenParamName for the Deep Counting model DeepCountingHandle in GenParamValue.

Note that when changing parameters that influence the template creation, prepare_deep_counting_model must be called again before the model can be applied with apply_deep_counting_model. The following table gives an overview, which parameters can be set using set_deep_counting_model_param or create_deep_counting_model (set), which can be retrieved using get_deep_counting_model_param (get), and which require re-running prepare_deep_counting_model after changing them (prepare).

GenParamName	set	get	Requires prepare
'angle_start'	x	x	x
'angle_step'	x	x	x
'angle_end'	x	x	x
'backbone_model'	x	x	x
'device'	x	x	x
'max_overlap'	x	x	
'min_score'	x	x	
'scale_max'	x	x	x
'scale_min'	x	x	x
'scale_step'	x	x	x

In the following the parameters are described:

'angle_start', *'angle_step'*, *'angle_end'*:

Control the rotational augmentation. Templates passed to prepare_deep_counting_model are rotated from *'angle_start'* to *'angle_end'* in steps of *'angle_step'*. This allows apply_deep_counting_model to better find rotated instances of the templates.

The angles must be passed in radians.

Suggested values: *0*, *-6.28*, *-3.14*, *3.14*, *6.28*

Default: *'angle_start'* = *0*, *'angle_end'* = *0*, *'angle_step'* = *'rad(30)'*

Restriction:

-2π <= *'angle_start'* <= *'angle_end'* <= 2π, *'angle_step'* > *0*

'backbone_model':

The backbone used for the detection of the templates. The backbone is automatically created by `create_deep_counting_model`. It can be obtained and written back in order to, for example, optimize it using `optimize_dl_model_for_inference`.

Note that the Deep Counting model will automatically set the input size of the backbone according to the template and image sizes. It has therefore no effect to change the backbone's input size, and it is not recommended to do so. Also note that the backbone can not be used for any other deep learning methods besides Deep Counting.

'device':

Handle of the device on which the backbone will be executed.

If the backbone was already optimized for a device, setting *'device'* might not be necessary anymore, see `optimize_dl_model_for_inference` for details.

To get a tuple of handles of all available potentially deep-learning capable hardware devices use `query_available_dl_devices`.

Default: Handle of the default device, thus the GPU with index *0*. If not available, this is an empty tuple.

'max_overlap':

The maximum allowed intersection over union (IoU) for two detected templates during counting. When two templates have an IoU higher than *'max_overlap'*, the one with lower confidence value gets suppressed. When set to *0*, no overlap at all is allowed. We refer to the chapter Deep Learning / Object Detection and Instance Segmentation for further explanations of the IoU.

Suggested values: *0.3*, *0.5*, *0.7*, *1.0*

Default: *'max_overlap' = 0.5*

Restriction: *0 <= 'max_overlap' <= 1*

'min_score':

This parameter determines the minimum similarity of detected instances to the original templates. In other words, `apply_deep_counting_model` ignores all detected instances with a similarity smaller than this value. The similarity computed by the Deep Counting model lies between *0* and *1*, where *0* means no similarity and *1* is a very high similarity.

Suggested values: *0.2*, *0.3*, *0.4*, *0.5*

Default: *'min_score' = 0.4*

Restriction: *0 < 'min_score' <= 1*

'scale_min', *'scale_step'*, *'scale_max'*:

Control the scale augmentation. Templates passed to `prepare_deep_counting_model` are scaled from *'scale_min'* to *'scale_max'* in steps of *'scale_step'*. This allows `apply_deep_counting_model` to better find scaled instances of the templates.

Suggested values: *0.9*, *1.0*, *1.1*

Default: *'scale_min' = 1.0*, *'scale_max' = 1.0*, *'scale_step' = 0.1*

Restriction: *0 < 'scale_min' <= 'scale_max'*, *'scale_step' > 0*

Parameters

▷ **`DeepCountingHandle`** (input_control) ..... deep_counting ⇝ *handle*
Handle of the Deep Counting model.

▷ **`GenParamName`** (input_control) ..... attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'angle_start'
List of values: `GenParamName` ∈ {'angle_start', 'angle_end', 'angle_step', 'backbone_model', 'device', 'max_overlap', 'min_score', 'scale_min', 'scale_max', 'scale_step'}

▷ **`GenParamValue`** (output_control) ..... attribute.name ⇝ *real* / string / integer
Value of the generic parameter.

Result

If the handle of the model is valid, the operator `get_deep_counting_model_param` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_deep_counting_model, set_deep_counting_model_param, read_deep_counting_model

Possible Successors

apply_deep_counting_model

See also

set_deep_counting_model_param

Module

Matching

prepare_deep_counting_model (Templates : : DeepCountingHandle :)

Set templates of a Deep Counting model.

prepare_deep_counting_model sets the templates of the objects that should be counted by the Deep Counting model DeepCountingHandle. When applying the Deep Counting model using apply_deep_counting_model, objects in the search images that are similar to the provided templates are detected and counted. Note that this operator overwrites any previously set templates.

To also count scaled and rotated variants of the provided templates, an automatic augmentation of the templates can be enabled by setting the parameters *'angle_start'*, *'angle_end'*, *'angle_step'*, *'scale_min'*, *'scale_max'*, and *'scale_step'* using create_deep_counting_model or set_deep_counting_model_param before calling apply_deep_counting_model.

When changing parameters of the Deep Counting model that influence the template creation, prepare_deep_counting_model must be re-run before apply_deep_counting_model. The list of such parameters is provided in get_deep_counting_model_param.

Attention

System requirements: To run this operator on GPU (see get_deep_counting_model_param), cuDNN and cuBLAS are required. For further details, please refer to the "Installation Guide", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods". Alternatively, this operator can also be run on CPU.

Parameters

▷ **Templates** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / real
Template image(s) with regions.

▷ **DeepCountingHandle** (input_control) ..... deep_counting ⇝ *handle*
Handle of the Deep Counting model.

Result

If the handle of the model is valid, the operator prepare_deep_counting_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- DeepCountingHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_deep_counting_model, set_deep_counting_model_param, get_deep_counting_model_param, read_deep_counting_model

Possible Successors

apply_deep_counting_model

Alternatives

read_deep_counting_model

See also

apply_deep_counting_model

Module

Matching

read_deep_counting_model (: : FileName : DeepCountingHandle)

Read a Deep Counting model from a file.

The operator read_deep_counting_model reads a Deep Counting model. Such models have to be in the HALCON format. As a result, the handle DeepCountingHandle is returned.

The model is loaded from the file FileName. This file is thereby searched in the directory $HALCONROOT/dl/ as well as in the currently used directory. The default HALCON file extension for Deep Counting models is '.hdc'.

Please note that the values of runtime specific parameters are not written to file, see write_deep_counting_model. As a consequence when reading a model these parameters are initialized with their default value, see get_deep_counting_model_param.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Filename
File extension: .hdc

▷ **DeepCountingHandle** (output_control) ..... deep_counting ⇝ *handle*
Handle of the Deep Counting model.

Result

If the parameters are valid, the operator read_deep_counting_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

create_deep_counting_model

Possible Successors

set_deep_counting_model_param, get_deep_counting_model_param, prepare_deep_counting_model, apply_deep_counting_model

Alternatives

create_deep_counting_model

Module

Matching

set_deep_counting_model_param (: : DeepCountingHandle, GenParamName, GenParamValue :)

Set selected parameters of the Deep Counting model.

The operator set_deep_counting_model_param sets the selected parameters GenParamName in the Deep Counting handle DeepCountingHandle to the values passed in GenParamValue.

The possible parameters are listed and described in get_deep_counting_model_param.

Parameters

▷ **DeepCountingHandle** (input_control) . . . . . deep_counting ⇝ *handle*
Handle of the Deep Counting model.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Parameter names.
Default: 'min_score'
List of values: GenParamName ∈ {'angle_start', 'angle_end', 'angle_step', 'backbone_model', 'device', 'max_overlap', 'min_score', 'scale_min', 'scale_max', 'scale_step'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
Parameter values.
Default: 0.5

Result

If the parameters are valid, the operator set_deep_counting_model_param returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DeepCountingHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_deep_counting_model, get_deep_counting_model_param

Possible Successors

prepare_deep_counting_model, apply_deep_counting_model

See also

get_deep_counting_model_param

Module

Matching

write_deep_counting_model (: : DeepCountingHandle, FileName :)

Write a Deep Counting model in a file.

write_deep_counting_model writes the Deep Counting model DeepCountingHandle to the file given by FileName. Please note that the runtime specific parameters *'gpu'*, *'runtime'*, and *'runtime_init'* are not written.

The default HALCON file extension for Deep Counting models is '.hdc'.

The Deep Counting model can be read with read_deep_counting_model.

Parameters

▷ **DeepCountingHandle** (input_control) . . . . . deep_counting ⇝ *handle*
Handle of the Deep Counting model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Filename
File extension: .hdc

Result

If the parameters are valid, the operator `write_deep_counting_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_deep_counting_model`, `set_deep_counting_model_param`

Possible Successors

`clear_handle`

Module

Matching

18.3 Deformable

clear_deformable_model (: : ModelID :)

Free the memory of a deformable model.

The operator `clear_deformable_model` frees the memory of a deformable model that was created by `create_planar_uncalib_deformable_model` or `create_planar_calib_deformable_model`. After calling `clear_deformable_model`, the deformable model can no longer be used. The handle `ModelID` becomes invalid.

Parameters

▷ **ModelID** (input_control) . . . . . deformable_model(-array) ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator `clear_deformable_model` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

`create_planar_uncalib_deformable_model`, `create_planar_calib_deformable_model`, `create_local_deformable_model`

Module

Matching

create_local_deformable_model (Template : : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleRMin, ScaleRMax, ScaleRStep, ScaleCMin, ScaleCMax, ScaleCStep, Optimization, Metric, Contrast, MinContrast, GenParamName, GenParamValue : ModelID)

Creates a deformable model for local, deformable matching.

The operator create_local_deformable_model prepares a template, which is passed in the image Template, as a deformable model used for local deformable matching. The ROI of the model is passed as the domain of Template.

The local deformable matching can be used to detect an object that is distorted by a local deformation, estimate the deformation, and rectify the image area where the object is found. For a distinction from further matching methods we refer to the "Solution Guide I" chapter '2D Matching'.

As described in create_planar_uncalib_deformable_model, the origin (reference point) of the model is defined as the center of gravity of the domain (region) of the model image Template. Further, the axis-aligned enclosing rectangle of the domain of the Template is used to determine the area of the search image that is rectified in calls of find_local_deformable_model.

For further explanation on the parameters used for the creation of a local deformable model we refer to the description of create_planar_uncalib_deformable_model. Note, that the parameters of the actual deformation are set with the call of find_local_deformable_model.

Parameters

▷ **Template** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: NumLevels ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStart** (input_control) . . . . . angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **AngleExtent** (input_control) . . . . . angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **AngleStep** (input_control) . . . . . angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: AngleStep ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep > 0 && AngleStep <= pi / 16

▷ **ScaleRMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in row direction.
Default: 1.0
Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **ScaleRStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in row direction.
Default: 'auto'
Suggested values: ScaleRStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleRStep >= 0

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in column direction.
Default: 1.0
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **ScaleCStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in column direction.
Default: 'auto'
Suggested values: ScaleCStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleCStep >= 0

▷ **Optimization** (input_control) . . . . . string(-array) ⇝ *string*
Kind of optimization used for generating the model.
Default: 'none'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_part_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **Contrast** (input_control) . . . . . number-array ⇝ *integer* / string
Thresholds or hysteresis thresholds for the contrast of the object in the template image.
Default: 'auto'
Suggested values: Contrast ∈ {'auto', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer* / string
Minimum contrast of the objects in the search images.
Default: 'auto'
Suggested values: MinContrast ∈ {'auto', 1, 2, 3, 5, 7, 10, 20, 30, 40}
Restriction: MinContrast < Contrast

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The generic parameter names.
Default: []
List of values: GenParamName ∈ {[], 'min_size', 'part_size'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the generic parameters.
Default: []
List of values: GenParamValue ∈ {[], 'small', 'medium', 'big'}

▷ **ModelID** (output_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_local_deformable_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

determine_deformable_model_params

Possible Successors

set_deformable_model_origin, set_deformable_model_param, get_deformable_model_contours, find_local_deformable_model, get_deformable_model_params, write_deformable_model, clear_deformable_model

Alternatives

read_deformable_model

See also

create_planar_uncalib_deformable_model, create_planar_calib_deformable_model

Module

Matching

create_local_deformable_model_xld (Contours : : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleRMin, ScaleRMax, ScaleRStep, ScaleCMin, ScaleCMax, ScaleCStep, Optimization, Metric, MinContrast, GenParamName, GenParamValue : ModelID)

Prepare a deformable model for local deformable matching from XLD contours.

The operator create_local_deformable_model_xld creates a deformable model used for local deformable matching from the XLD contours passed in Contours. The XLD contours represent the gray-value edges of the object to be searched for. In contrast to the operator create_local_deformable_model, which creates a deformable model from a template image, the operator create_local_deformable_model_xld creates the deformable model from XLD contours.

The center of gravity of the smallest surrounding rectangle of the Contours that is parallel to the coordinate axes is used as the origin (reference point) of the model. A different origin can be set with set_deformable_model_origin.

For further explanation on the deformable model and its parameters we refer to the description of create_planar_uncalib_deformable_model and create_local_deformable_model. Note that set_local_deformable_model_metric can be used to change the metric of the model.

Attention

Note that, in contrast to the operator create_local_deformable_model, it is not possible to specify a minimum size of the model components. To avoid small model components in the model, short contours can be eliminated with the operator select_contours_xld before calling create_local_deformable_model_xld.

Parameters

- ▷ **Contours** (input_object) ..... xld_cont(-array) ⇝ *object*
  Input contours that will be used to create the model.
- ▷ **NumLevels** (input_control) ..... integer ⇝ *integer* / string
  Maximum number of pyramid levels.
  Default: 'auto'
  List of values: NumLevels ∈ {'auto', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
- ▷ **AngleStart** (input_control) ..... angle.rad-array ⇝ *real*
  This parameter is not used.
  Default: []
- ▷ **AngleExtent** (input_control) ..... angle.rad-array ⇝ *real*
  This parameter is not used.
  Default: []
- ▷ **AngleStep** (input_control) ..... angle.rad ⇝ *real* / string
  Step length of the angles (resolution).
  Default: 'auto'
  Suggested values: AngleStep ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
  Restriction: AngleStep > 0 && AngleStep <= pi / 16
- ▷ **ScaleRMin** (input_control) ..... number ⇝ *real*
  Minimum scale of the pattern in row direction.
  Default: 1.0
  Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
  Restriction: ScaleRMin > 0
- ▷ **ScaleRMax** (input_control) ..... number-array ⇝ *real*
  This parameter is not used.
  Default: []

▷ **ScaleRStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in row direction.
Default: 'auto'
Suggested values: ScaleRStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleRStep > 0

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in column direction.
Default: 1.0
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **ScaleCStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in column direction.
Default: 'auto'
Suggested values: ScaleCStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleCStep > 0

▷ **Optimization** (input_control) . . . . . string(-array) ⇝ *string*
Kind of optimization used for generating the model.
Default: 'auto'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'ignore_local_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_part_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer*
Minimum contrast of the objects in the search images.
Default: 5
Suggested values: MinContrast ∈ {1, 2, 3, 5, 7, 10, 20, 30, 40}

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The generic parameter names.
Default: []
List of values: GenParamName ∈ {[], 'part_size'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the generic parameters.
Default: []
List of values: GenParamValue ∈ {[], 'small', 'medium', 'big'}

▷ **ModelID** (output_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_local_deformable_model_xld returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised. If the parameter NumLevels is chosen such that the model contains too few points, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_contour_xld_dxf, edges_sub_pix, select_contours_xld

Possible Successors

find_local_deformable_model

See also

create_local_deformable_model

Module

Matching

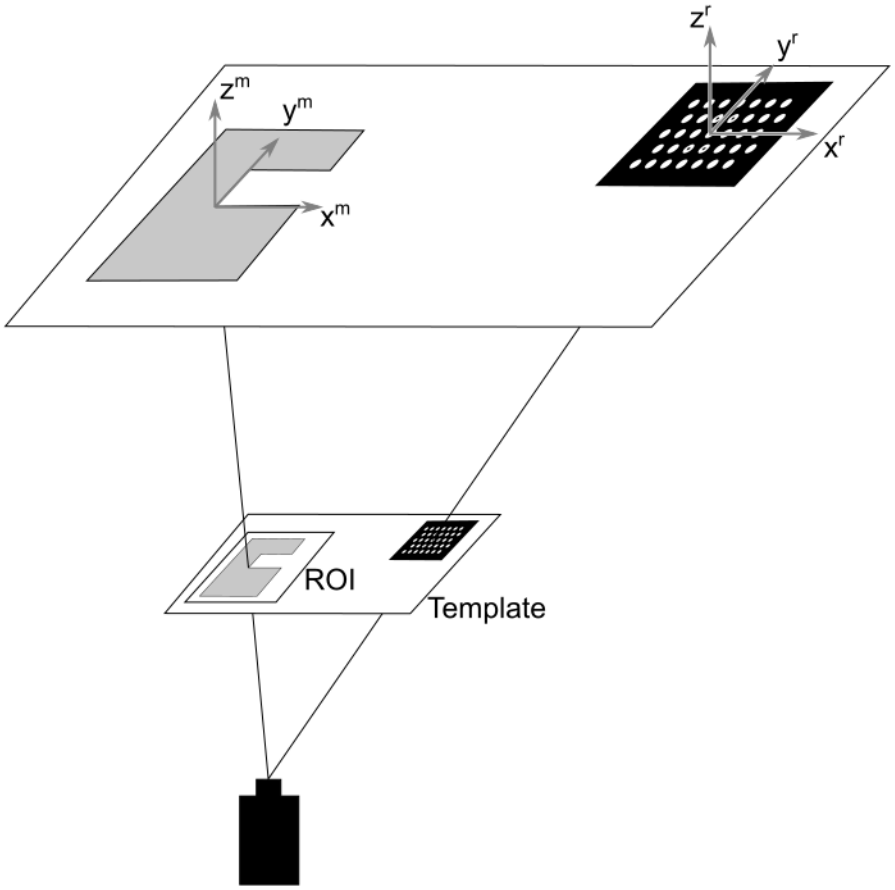
create_planar_calib_deformable_model (Template : : CamParam, ReferencePose, NumLevels, AngleStart, AngleExtent, AngleStep, ScaleRMin, ScaleRMax, ScaleRStep, ScaleCMin, ScaleCMax, ScaleCStep, Optimization, Metric, Contrast, MinContrast, GenParamName, GenParamValue : ModelID)

Create a deformable model for calibrated perspective matching.

The operator create_planar_calib_deformable_model creates a deformable model for planar calibrated matching based on the input image Template.

The model ROI corresponds to the domain of Template. The model is generated for multiple image pyramid levels and is stored in memory. The output parameter ModelID is a handle for this model and is used in subsequent calls to find_planar_calib_deformable_model.

The internal camera parameters are provided in CamParam. The internal parameters can typically be determined using calibrate_cameras. By providing CamParam it is possible to retrieve a 3D pose of the model as a result of the planar calibrated matching. See create_planar_uncalib_deformable_model on how to progress without using calibration data. See "Solution Guide II-B - Matching" for more information on these approaches.



Scheme of the reference pose ReferencePose in relation to the absolute pose of the model as projected from the image into the model plane.

The x- and y-axis of the reference pose `ReferencePose` (x^r, y^r, z^r) determine the 3D model plane, thus the plane in the world coordinate system, the model is located in (see scheme above). The 3D model plane can be obtained by using `calibrate_cameras` or `vector_to_pose`, e.g., by placing and localizing a calibration plate in the same plane as the model is located in.

The model's origin is located in the center of gravity of the model region in `Template`. The projection of this image point onto the 3D model plane determined by `ReferencePose` corresponds to the absolute 3D pose of the model (x^m, y^m, z^m). The projection is depicted in the scheme above. After the model creation you can query the absolute pose of the model in world coordinates using `get_deformable_model_params`. The 3D pose of the model differs from the reference pose by a 2D translation in the model plane.

Further input parameters that specify the model are described in `create_planar_uncalib_deformable_model`.

Parameters

▷ **`Template`** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **`CamParam`** (input_control) . . . . . campar ⇝ *real* / integer / string
The parameters of the internal orientation of the camera.

▷ **`ReferencePose`** (input_control) . . . . . pose ⇝ *real* / integer
The reference pose of the object in the reference image.

▷ **`NumLevels`** (input_control) . . . . . integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: `NumLevels` ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **`AngleStart`** (input_control) . . . . . angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **`AngleExtent`** (input_control) . . . . . angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **`AngleStep`** (input_control) . . . . . angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: `AngleStep` ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep > 0 && AngleStep <= pi / 16

▷ **`ScaleRMin`** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in row direction.
Default: 1.0
Suggested values: `ScaleRMin` ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **`ScaleRMax`** (input_control) . . . . . number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **`ScaleRStep`** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in row direction.
Default: 'auto'
Suggested values: `ScaleRStep` ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleRStep >= 0

▷ **`ScaleCMin`** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in column direction.
Default: 1.0
Suggested values: `ScaleCMin` ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **`ScaleCMax`** (input_control) . . . . . number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **ScaleCStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in the column direction.
Default: 'auto'
Suggested values: ScaleCStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleCStep >= 0

▷ **Optimization** (input_control) . . . . . string(-array) ⇝ *string*
Kind of optimization used for generating the model.
Default: 'none'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_part_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **Contrast** (input_control) . . . . . number-array ⇝ *integer* / string
Thresholds or hysteresis thresholds for the contrast of the object in the template image.
Default: 'auto'
Suggested values: Contrast ∈ {'auto', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer* / string
Minimum contrast of the objects in the search images.
Default: 'auto'
Suggested values: MinContrast ∈ {'auto', 1, 2, 3, 5, 7, 10, 20, 30, 40}
Restriction: MinContrast < Contrast

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The parameter names.
Default: []
List of values: GenParamName ∈ {[], 'part_size'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the parameters.
Default: []
List of values: GenParamValue ∈ {[], 'small', 'medium', 'big'}

▷ **ModelID** (output_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_planar_calib_deformable_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

determine_deformable_model_params

Possible Successors

set_deformable_model_origin, set_deformable_model_param, get_deformable_model_contours, find_planar_calib_deformable_model, get_deformable_model_params, write_deformable_model, clear_deformable_model

Alternatives

read_deformable_model

See also

create_planar_uncalib_deformable_model

Module

Matching

create_planar_calib_deformable_model_xld (
Contours : : CamParam, ReferencePose, NumLevels, AngleStart, AngleExtent, AngleStep, ScaleRMin, ScaleRMax, ScaleRStep, ScaleCMin, ScaleCMax, ScaleCStep, Optimization, Metric, MinContrast, GenParamName, GenParamValue : ModelID)

Prepare a deformable model for planar calibrated matching from XLD contours.

The operator create_planar_calib_deformable_model_xld creates a deformable model used for planar calibrated matching from the XLD contours passed in Contours. The XLD contours represent the gray-value edges of the object to be searched for. In contrast to the operator create_planar_calib_deformable_model, which creates a deformable model from a template image, the operator create_planar_calib_deformable_model_xld creates the deformable model from XLD contours. Note that the operator expects Contours to be in a metric world coordinate system. This is in contrast to create_planar_uncalib_deformable_model_xld, where the Contours are provided in image coordinates. To check if shape and size of the model correspond with the objects in the world coordinate system you can project the model into the image using project_3d_point.

Further, the origin (reference point) of the model is taken directly from the provided Contours. A different origin in the world coordinate system can be set with set_deformable_model_origin.

The parameters ModelID, CamParam and ReferencePose are described in create_planar_calib_deformable_model. Further input parameters that specify the model are described in create_planar_uncalib_deformable_model.

Attention

Note that, in contrast to the operator create_planar_calib_deformable_model, it is not possible to specify a minimum size of the model components. To avoid small model components in the model, short contours can be eliminated before calling create_planar_calib_deformable_model_xld with the operator select_contours_xld.

Parameters

▷ **Contours** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contours that will be used to create the model.

▷ **CamParam** (input_control) ..... campar ⇝ *real* / integer / string
The parameters of the internal orientation of the camera.

▷ **ReferencePose** (input_control) ..... pose ⇝ *real* / integer
The reference pose of the object.

▷ **NumLevels** (input_control) ..... integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: NumLevels ∈ {'auto', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStart** (input_control) ..... angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **AngleExtent** (input_control) ..... angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **AngleStep** (input_control) ..... angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: AngleStep ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep > 0 && AngleStep <= pi / 16

▷ **ScaleRMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in row direction.
Default: 1.0
Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **ScaleRStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in row direction.
Default: 'auto'
Suggested values: ScaleRStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleRStep > 0

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in column direction.
Default: 1.0
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **ScaleCStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in the column direction.
Default: 'auto'
Suggested values: ScaleCStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleCStep > 0

▷ **Optimization** (input_control) . . . . . string(-array) ⇝ *string*
Kind of optimization used for generating the model.
Default: 'auto'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'ignore_local_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_part_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer*
Minimum contrast of the objects in the search images.
Default: 5
Suggested values: MinContrast ∈ {1, 2, 3, 5, 7, 10, 20, 30, 40}

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The generic parameter names.
Default: []
List of values: GenParamName ∈ {[], 'part_size'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the generic parameter.
Default: []
List of values: GenParamValue ∈ {[], 'small', 'medium', 'big'}

▷ **ModelID** (output_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_planar_calib_deformable_model_xld returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised. If the parameter NumLevels is chosen such that the model contains too few points, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_contour_xld_dxf, edges_sub_pix, select_contours_xld

Possible Successors

find_planar_calib_deformable_model

See also

create_planar_calib_deformable_model

Module

Matching

create_planar_uncalib_deformable_model (Template : : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleRMin, ScaleRMax, ScaleRStep, ScaleCMin, ScaleCMax, ScaleCStep, Optimization, Metric, Contrast, MinContrast, GenParamName, GenParamValue : ModelID)

Creates a deformable model for uncalibrated, perspective matching.

The operator create_planar_uncalib_deformable_model prepares a template, which is passed in the image Template, as a deformable model used for uncalibrated perspective matching. The ROI of the model is passed as the domain of Template.

The planar uncalibrated matching can be used to detect planar objects or planar subparts of non-planar objects that are distorted by a projective view.

This is an alternative to create_generic_shape_model, where typically the search image must be rectified by calibrate_cameras and gen_image_to_world_plane_map beforehand.

In contrast to create_shape_model_3d there is no need to pre-generate different views of an object, resulting into a smaller memory consumption. Hence, in case of a planar perspectively distorted object, create_planar_uncalib_deformable_model should be preferred.

The model is generated using multiple image pyramid levels and is stored in memory. The output parameter ModelID is a handle for this model, which is used in subsequent calls to find_planar_uncalib_deformable_model.

Input parameters in detail

NumLevels: The number of pyramid levels is determined with the parameter NumLevels. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, NumLevels must be chosen such that the model is still recognizable and contains a sufficient number of points on the highest pyramid level. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, create_planar_uncalib_deformable_model returns with an error message. If NumLevels is set to *'auto'*, create_planar_uncalib_deformable_model determines the number of pyramid levels automatically. The automatically computed number of pyramid levels can be queried using get_deformable_model_params. In rare cases, it might happen that create_planar_uncalib_deformable_model determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for MinScore or Greediness in find_planar_uncalib_deformable_model in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in find_planar_uncalib_deformable_model increases. In these cases, the number of pyramid levels should be selected using the output of inspect_shape_model.

Angle and Scale parameters: Note that the parameters AngleStart, AngleExtent, ScaleRMax, and ScaleCMax are not used by this operator. Instead, they have to be specified within the operator find_planar_uncalib_deformable_model. ScaleRMin and ScaleCMin can be relevant if you expect matches that are smaller than in the original image Template. Then, if NumLevels is set to *'auto'*, the number of used pyramid levels may decrease.

Generally, the parameters AngleStep, ScaleRStep and ScaleCStep can be determined automatically. The automatically computed angle and scale step lengths can be queried using get_deformable_model_params. For more information about how these parameters work, please refer to the operator find_planar_uncalib_deformable_model.

Optimization: For particularly large models, it may be useful to reduce the number of model points by setting Optimization to a value different from *'none'*. If Optimization = *'none'*, all model points are stored. In all other cases, the number of points is reduced according to the value of Optimization. If the number of points is reduced, it may be necessary in find_planar_uncalib_deformable_model to set the parameter Score and Greediness to a smaller value as, e.g., *0.7* or *0.8*. For small models, the reduction of the number of model points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined. If Optimization is set to *'auto'*, create_planar_uncalib_deformable_model automatically determines the reduction of the number of model points.

Contrast: The parameter Contrast determines the contrast the model points (edges) must have. The contrast is a measure for local gray value differences between the object and the background and between different parts of the object. Contrast should be chosen such that only the significant features of the template are used for the model.

The following values can be set for Contrast:

- A single contrast value. You can set:
  - an integer value: the manually determined value is used.
  - *'auto_contrast'*: the contrast value is computed automatically.
- Hysteresis thresholds. In this case, the model is segmented using a method similar to the hysteresis threshold method used in edges_image. For more information about the hysteresis threshold method, see hysteresis_threshold. You can set:
  - A tuple of two values: The first value determines the lower threshold, the second value determines the upper threshold.
  - *'auto'* (default value) or *'auto_contrast_hyst'*: The hysteresis thresholds are computed automatically.

In certain cases, it might happen that the automatic determination of the contrast thresholds is not satisfying. For example, a manual setting of these parameters should be preferred if certain model components should be included or suppressed because of application-specific reasons or if the object contains several different contrasts. Therefore, the contrast thresholds should be automatically determined with determine_deformable_model_params and subsequently verified using inspect_shape_model before calling create_planar_uncalib_deformable_model. Note that MinContrast influences the automatic contrast estimation, and hence also the estimation of the minimum size.

MinContrast: With MinContrast, it can be determined which minimal contrast the model must have in the recognition performed by find_planar_uncalib_deformable_model. In other words, this parameter separates the model from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, MinContrast should be set to 10. Obviously, MinContrast must be smaller than Contrast. If the model should be recognized in very low contrast images, MinContrast must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, MinContrast should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the position and rotation of the model are extracted robustly and accurately by find_planar_uncalib_deformable_model. If MinContrast is set to *'auto'*, the minimum contrast is determined automatically based on the noise in the model image. Consequently, an automatic determination only makes sense if the image noise during the recognition is similar to the noise in the model image. Furthermore, in some cases it is advisable to increase the automatically determined value in order to increase the robustness against occlusions (see above). The automatically computed minimum contrast can be queried using get_deformable_model_params. The MinContrast of a deformable model can later be changed with the help of set_deformable_model_param.

Metric: The parameter Metric determines the conditions under which the model is recognized in the image. If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background. If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_planar_uncalib_deformable_model will increase slightly in this case. If Metric = *'ignore_part_polarity'*, the the contrast polarity is allowed to change only between different parts of the model, whereas the polarity of model points that are within the same model part must not change. Please note that the term *'ignore_part_polarity'* is capable of being misunderstood. It means that polarity changes between neighboring model parts do not influence the score, and hence are ignored. If Metric = *'ignore_local_polarity'*, the model is found even if the contrast changes for each individual model point. This mode can, for example, be useful if the object consists of a part with medium gray value, within which either darker or brighter sub-objects lie. In this case the runtime of find_planar_uncalib_deformable_model increases significantly. The above four metrics can only be applied to single-channel images. If a multi-channel image is used as the model image or as the search image only the first channel will be used (and no error message will be returned). If Metric = *'ignore_color_polarity'*, the model is found even if the color contrast changes locally. This is, for example, the case if parts of the object can change their color, e.g., from red to green. In particular, this mode is useful if it is not known in advance in which channels the object is visible. In this mode, the runtime of find_planar_uncalib_deformable_model can also increase significantly. The metric *'ignore_color_polarity'* can be used for images with an arbitrary number of channels. If it is used for single-channel images it has the same effect as *'ignore_local_polarity'*. It should be noted that for Metric = *'ignore_color_polarity'* the number of channels in the model creation with create_planar_uncalib_deformable_model and in the search with find_planar_uncalib_deformable_model can be different. This can, for example, be used to create a model from a synthetically generated single-channel image. Furthermore, it should be noted that the channels do not need to contain a spectral subdivision of the light (like in an RGB image). The channels can, for example, also contain images of the same object that were obtained by illuminating the object from different directions.

GenParamName, GenParamValue: With the help of the generic parameters GenParamName and GenParamValue, the user can set parameters for the deformable model generation.

The following parameters can be set for GenParamName.

'part_size': Adapts the average size of the sub-parts that the deformable model should consist of. For objects that consist of many small contours, *'small'* should be selected. For objects that consist of only big contours, *'big'* should be set.
List of values: *'small'*, *'medium'*, *'big'*
Default: *'big'*.

'min_size': Determines a threshold for the selection of significant model components based on the size of the components, i.e., components that have fewer points than the minimum size specified in the corresponding GenParamValue are suppressed. This threshold for the minimum size is divided by two for each successive pyramid level. *'min_size'* can be set to an arbitrary integer value greater or equal *0*. It can also be set to *'auto'* which means that the minimum size will be determined automatically.
Default: *'auto'*

The center of gravity of the domain (region) of the model image Template is used as the origin (reference point) of the model. A different origin for the model can be set with set_deformable_model_origin.

Parameters

▷ **Template** (input_object) ..... (multichannel-)image ⇝ *object* : byte / uint2
Input image whose domain will be used to create the model.

▷ **NumLevels** (input_control) ..... integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: NumLevels ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStart** (input_control) ..... angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **AngleExtent** (input_control) ..... angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **AngleStep** (input_control) . . . . . angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: AngleStep ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep > 0 && AngleStep <= pi / 16

▷ **ScaleRMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in row direction.
Default: 1.0
Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **ScaleRStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in row direction.
Default: 'auto'
Suggested values: ScaleRStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleRStep >= 0

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in column direction.
Default: 1.0
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **ScaleCStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in column direction.
Default: 'auto'
Suggested values: ScaleCStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleCStep >= 0

▷ **Optimization** (input_control) . . . . . string(-array) ⇝ *string*
Kind of optimization used for generating the model.
Default: 'none'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_part_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **Contrast** (input_control) . . . . . number-array ⇝ *integer* / string
Thresholds or hysteresis thresholds for the contrast of the object in the template image.
Default: 'auto'
Suggested values: Contrast ∈ {'auto', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer* / string
Minimum contrast of the objects in the search images.
Default: 'auto'
Suggested values: MinContrast ∈ {'auto', 1, 2, 3, 5, 7, 10, 20, 30, 40}
Restriction: MinContrast < Contrast

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The generic parameter names.
Default: []
List of values: GenParamName ∈ {[], 'min_size', 'part_size'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the generic parameter.
Default: []
List of values: GenParamValue ∈ {[], 'small', 'medium', 'big'}

▷ **ModelID** (output_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator `create_planar_uncalib_deformable_model` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`determine_deformable_model_params`

Possible Successors

`set_deformable_model_origin`, `set_deformable_model_param`, `get_deformable_model_contours`, `find_planar_uncalib_deformable_model`, `get_deformable_model_params`, `write_deformable_model`, `clear_deformable_model`

Alternatives

`read_deformable_model`

Module

Matching

create_planar_uncalib_deformable_model_xld (
Contours : : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleRMin, ScaleRMax, ScaleRStep, ScaleCMin, ScaleCMax, ScaleCStep, Optimization, Metric, MinContrast, GenParamName, GenParamValue : ModelID)

Prepare a deformable model for planar uncalibrated matching from XLD contours.

The operator `create_planar_uncalib_deformable_model_xld` creates a deformable model used for planar uncalibrated matching from the XLD contours passed in `Contours`. The XLD contours represent the gray-value edges of the object to be searched for. In contrast to the operator `create_planar_uncalib_deformable_model`, which creates a deformable model from a template image, the operator `create_planar_uncalib_deformable_model_xld` creates the deformable model from XLD contours.

The model is generated for multiple image pyramid levels and is stored in memory. The center of gravity of the smallest surrounding rectangle of the `Contours` that is parallel to the coordinate axes is used as the origin (reference point) of the model. A different origin can be set with `set_deformable_model_origin`. The output parameter `ModelID` is a handle for this model, which is used in subsequent calls to `find_planar_uncalib_deformable_model`.

Some Input parameters in detail

NumLevels: The number of pyramid levels is determined with the parameter `NumLevels`. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, `NumLevels` must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure leads to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, `create_planar_uncalib_deformable_model_xld` returns with an error message. If `NumLevels` is set to *'auto'*, `create_planar_uncalib_deformable_model_xld` determines the number of pyramid levels automatically. The computed number of pyramid levels can be queried using `get_deformable_model_params`. In rare cases, it might happen that

create_planar_uncalib_deformable_model_xld determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the search image or it may be necessary to select very low parameters for MinScore or Greediness in find_planar_uncalib_deformable_model to find the object. If the number of pyramid levels is chosen too small, the time required to find the object with find_planar_uncalib_deformable_model may increase. In both cases, the number of pyramid levels should be selected manually.

Angle and Scale parameters: Note that the parameters AngleStart, AngleExtent, ScaleRMax, and ScaleCMax are not used by this operator. Instead, they have to be specified within the operator find_planar_uncalib_deformable_model. ScaleRMin and ScaleCMin can be relevant if you expect matches that are smaller than the given XLD Contours. Then, if NumLevels is set to *'auto'*, the number of used pyramid levels may decrease.

Generally, the parameters AngleStep, ScaleRStep and ScaleCStep can be determined automatically. The automatically computed angle and scale step lengths can be queried using get_deformable_model_params. For more information about how these parameters work, please refer to the operator find_planar_uncalib_deformable_model.

Optimization: For particularly large models, it may be useful to reduce the number of model points by setting Optimization to a value different from *'none'*. If Optimization = *'none'* is set, all model points are stored. In all other cases, the number of points is reduced according to the value of Optimization. If the number of points is reduced, it may be necessary in find_planar_uncalib_deformable_model to set the parameter Score and Greediness to a smaller value as, e.g., *0.7* or *0.8*. For small models, the reduction of the number of model points does not result in a speed-up of the search, because in this case usually significantly more potential instances of the model must be examined. If Optimization is set to *'auto'*, create_planar_uncalib_deformable_model_xld automatically determines the reduction of the number of model points.

MinContrast: With MinContrast, it can be determined which contrast the object contours must at least have in the recognition performed by find_planar_uncalib_deformable_model. In other words, this parameter separates the object from the noise in the image. Therefore, a good choice is the range of gray-value changes caused by the noise in the image. If, e.g., the gray values fluctuate within a range of 10 gray levels, MinContrast should be set to 10. If multi-channel images are used for the model and the search images, and if the parameter Metric is set to *'ignore_color_polarity'* (see below) the noise in one channel must be multiplied by the square root of the number of channels to determine MinContrast. If, e.g., the gray values fluctuate within a range of 10 gray levels in a single channel and the image is a three-channel image, MinContrast should be set to 17. If the model should be recognized in very low contrast images, MinContrast must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, MinContrast should be slightly larger than the range of gray value fluctuations created by noise to ensure that the position and rotation of the model are extracted robustly and accurately by find_planar_uncalib_deformable_model.

Metric: The parameter Metric determines the conditions under which the model is recognized in the image. If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, e.g., the model is a bright object on a dark background, the object is found only if it is also brighter than the background. If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_planar_uncalib_deformable_model will increase slightly in this case. If Metric = *'ignore_part_polarity'* or Metric = *'ignore_local_polarity'*, the model is found even if the contrast changes locally. These modes can, e.g., be useful if the object consists of a part with a medium gray value, within which either darker or brighter sub-objects lie. The above mentioned four metrics can only be applied to single-channel images. If a multi-channel image is used as the search image, only the first channel will be used (and no error message will be returned). If Metric = *'ignore_color_polarity'*, the model is found even if the color contrast changes locally. This is, e.g., the case if parts of the object can change their color, e.g., from red to green. In particular, this mode is useful if it is not known in advance in which channels the object is visible. In this mode, the runtime of find_planar_calib_deformable_model can also increase significantly. The metric *'ignore_color_polarity'* can be used for images with an arbitrary number of channels. If it is used for single-channel images, it has the same effect as *'ignore_local_polarity'*. It should be noted that for Metric = *'ignore_color_polarity'* the channels do not need to contain a spectral subdivision of the light (like in an RGB image). The channels can, e.g., also contain images of the same object that were obtained by illuminating the object from different directions. Note that the first two metrics (*'use_polarity'* and *'ignore_global_polarity'*) can only be selected if all Contours provide the attribute

'edge_direction', which defines the polarity of the contained contours. For more information about contour attributes like *'edge_direction'* see `get_contour_attrib_xld`. Otherwise, these two metrics can be selected with the operator `set_planar_uncalib_deformable_model_metric`, which determines the polarity of the model contours from a representative image.

Attention

Note that, in contrast to the operator `create_planar_uncalib_deformable_model`, it is not possible to specify a minimum size of the model components. To avoid small model components in the model, short contours can be eliminated before calling `create_planar_uncalib_deformable_model_xld` with the operator `select_contours_xld`.

Parameters

▷ **`Contours`** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contours that will be used to create the model.

▷ **`NumLevels`** (input_control) ..... integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
Suggested values: `NumLevels` ∈ {'auto', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **`AngleStart`** (input_control) ..... angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **`AngleExtent`** (input_control) ..... angle.rad-array ⇝ *real*
This parameter is not used.
Default: []

▷ **`AngleStep`** (input_control) ..... angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: `AngleStep` ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep > 0 && AngleStep <= pi / 16

▷ **`ScaleRMin`** (input_control) ..... number ⇝ *real*
Minimum scale of the pattern in row direction.
Default: 1.0
Suggested values: `ScaleRMin` ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **`ScaleRMax`** (input_control) ..... number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **`ScaleRStep`** (input_control) ..... number ⇝ *real* / string
Scale step length (resolution) in row direction.
Default: 'auto'
Suggested values: `ScaleRStep` ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleRStep > 0

▷ **`ScaleCMin`** (input_control) ..... number ⇝ *real*
Minimum scale of the pattern in column direction.
Default: 1.0
Suggested values: `ScaleCMin` ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **`ScaleCMax`** (input_control) ..... number-array ⇝ *real*
This parameter is not used.
Default: []

▷ **`ScaleCStep`** (input_control) ..... number ⇝ *real* / string
Scale step length (resolution) in the column direction.
Default: 'auto'
Suggested values: `ScaleCStep` ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleCStep > 0

▷ **`Optimization`** (input_control) ..... string(-array) ⇝ *string*
Kind of optimization used for generating the model.
Default: 'auto'
List of values: `Optimization` ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium',

'point_reduction_high'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'ignore_local_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_part_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer*
Minimum contrast of the objects in the search images.
Default: 5
Suggested values: MinContrast ∈ {1, 2, 3, 5, 7, 10, 20, 30, 40}

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The generic parameter names.
Default: []
List of values: GenParamName ∈ {[], 'part_size'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the generic parameters.
Default: []
List of values: GenParamValue ∈ {[], 'small', 'medium', 'big'}

▷ **ModelID** (output_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_planar_uncalib_deformable_model_xld returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised. If the parameter NumLevels is chosen such that the model contains too few points, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_contour_xld_dxf, edges_sub_pix, select_contours_xld

Possible Successors

find_planar_uncalib_deformable_model

See also

create_planar_uncalib_deformable_model

Module

Matching

deserialize_deformable_model (
: : SerializedItemHandle : ModelID)

Deserialize a deformable model.

deserialize_deformable_model deserializes a deformable model, that was serialized by serialize_deformable_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized deformable model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created deformable model with the handle ModelID.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **ModelID** (output_control) ..... deformable_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator `deserialize_deformable_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_deformable_model`

Possible Successors

`find_planar_uncalib_deformable_model`, `find_planar_calib_deformable_model`, `find_local_deformable_model`

See also

`create_planar_uncalib_deformable_model`, `create_planar_calib_deformable_model`, `create_local_deformable_model`

Module

Matching

determine_deformable_model_params (Template : : NumLevels, AngleStart, AngleExtent, ScaleMin, ScaleMax, Optimization, Metric, Contrast, MinContrast, GenParamName, GenParamValue, Parameters : ParameterName, ParameterValue)

Determine the parameters of a deformable model.

`determine_deformable_model_params` determines certain parameters of a deformable model automatically from the model image `Template`. The parameters to be determined can be specified with `Parameters`. `determine_deformable_model_params` can be used to determine the same parameters that are determined automatically when the respective parameter in `create_planar_uncalib_deformable_model` is set to *'auto'*: the number of pyramid levels (`Parameters` = *'num_levels'*), the angle step length (`Parameters` = *'angle_step'*), the scale step length (`Parameters` = *'scale_step'* for isotropic scaling and *'scale_r_step'* and/or *'scale_c_step'* for anisotropic scaling), the kind of optimization (`Parameters` = *'optimization'*), the threshold (`Parameters` = *'contrast'*) or the hysteresis thresholds (`Parameters` = *'contrast_hyst'*) for the contrast, the minimum size of the object parts (`Parameters` = *'min_size'*), and the minimum contrast (`Parameters` = *'min_contrast'*). By passing a tuple of the above values in `Parameters`, an arbitrary combination of these parameters can be determined. If all of the above parameters should be determined, the value *'all'* can be passed. In this case both hysteresis thresholds are determined, i.e., the operator behaves like passing *'contrast_hyst'* instead of *'contrast'*.

`determine_deformable_model_params` is mainly useful to determine the above parameters before creating the model, e.g., in an interactive system, which makes suggestions for these parameters to the user, but enables the user to modify the suggested values.

The automatically determined parameters are returned as a name-value pair in `ParameterName` and `ParameterValue`. The parameter names in `ParameterName` are identical to the names in `Parameters`, where, of course, the value *'all'* is replaced by the actual parameter names. An exception is the parameter *'contrast_hyst'*, for which the two values *'contrast_low'* and *'contrast_high'* are returned.

The remaining parameters (`NumLevels`, `AngleStart`, `AngleExtent`, `ScaleMin`, `ScaleMax`, `Optimization`, `Metric`, `Contrast`, and `MinContrast`) have

the same meaning as in create_planar_uncalib_deformable_model and find_planar_uncalib_deformable_model. The description of these parameters can be looked up with these operators. These parameters are used by determine_deformable_model_params to calculate the parameters to be determined in the same manner as in create_planar_uncalib_deformable_model. It should be noted that if the parameters of a deformable model with isotropic scaling should be determined, i.e., if Parameters contains *'scale_step'* either explicitly or implicitly via *'all'*, the parameters ScaleMin and ScaleMax must contain one value each. If the parameters of a deformable model with anisotropic scaling should be determined, i.e., if Parameters contains *'scale_r_step'* or *'scale_c_step'* either explicitly or implicitly, the parameters ScaleMin and ScaleMax must contain two values each. In this case, the first value of the respective parameter refers to the scaling in row direction, while the second value refers to scaling in the column direction.

Note that in determine_deformable_model_params some parameters appear that can also be determined automatically (NumLevels, Optimization, Contrast, MinContrast). If these parameters should not be determined automatically, i.e., their name is not passed in ParameterName, the corresponding parameters must contain valid values and must not be set to *'auto'*. In contrast, if these parameters are to be determined automatically, their values are treated in the following way: If the optimization or the (hysteresis) contrast is to be determined automatically, i.e., ParameterName contains the value *'optimization'* or *'contrast'* (*'contrast_hyst'*), the values passed in Optimization and Contrast are ignored. In contrast, if the maximum number of pyramid levels or the minimum contrast is to be determined automatically, i.e., ParameterName contains the value *'num_levels'* or *'min_contrast'*, you can let HALCON determine suitable values and at the same time specify an upper or lower boundary, respectively:

If the maximum number of pyramid levels should be specified in advance, NumLevels can be set to the particular value. If in this case Parameters contains the value *'num_levels'*, the computed number of pyramid levels is at most NumLevels. If NumLevels is set to *'auto'* (or *0* for backwards compatibility), the number of pyramid levels is determined without restrictions as large as possible.

If the minimum contrast should be specified in advance, MinContrast can be set to the particular value. If in this case Parameters contains the value *'min_contrast'*, the computed minimum contrast is at least MinContrast. If MinContrast is set to *'auto'*, the minimum contrast is determined without restrictions.

If the user wishes to create a calibrated, deformable model with create_planar_calib_deformable_model, the internal camera parameters are needed for the correct estimation of the parameters in ParameterValue. The camera parameters can be provided by setting GenParamName to *'cam_param'* and providing the tuple containing the camera parameters in GenParamValue.

Parameters

▷ **Template** (input_object) ..... (multichannel-)image ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **NumLevels** (input_control) ..... integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: NumLevels ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStart** (input_control) ..... angle.rad ⇝ *real*
Smallest rotation of the model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) ..... angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.28, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **ScaleMin** (input_control) ..... number(-array) ⇝ *real*
Minimum scale of the model.
Default: 0.9
Suggested values: ScaleMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleMin > 0

▷ **ScaleMax** (input_control) ..... number(-array) ⇝ *real*
Maximum scale of the model.
Default: 1.1
Suggested values: ScaleMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleMax >= ScaleMin

▷ **Optimization** (input_control) . . . . . string ⇝ *string*
Kind of optimization.
Default: 'auto'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **Contrast** (input_control) . . . . . number(-array) ⇝ *integer* / string
Threshold or hysteresis thresholds for the contrast of the object in the template image and optionally minimum size of the object parts.
Default: 'auto'
Suggested values: Contrast ∈ {'auto', 'auto_contrast', 'auto_contrast_hyst', 'auto_min_size', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer* / string
Minimum contrast of the objects in the search images.
Default: 'auto'
Suggested values: MinContrast ∈ {'auto', 1, 2, 3, 5, 7, 10, 20, 30, 40}
Restriction: MinContrast < Contrast

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The general parameter names.
Default: []
List of values: GenParamName ∈ {'cam_param'}

▷ **GenParamValue** (input_control) . . . . . number-array ⇝ *real* / integer / string
Values of the general parameter.
Default: []
List of values: GenParamValue ∈ {[]}

▷ **Parameters** (input_control) . . . . . string(-array) ⇝ *string*
Parameters to be determined automatically.
Default: 'all'
List of values: Parameters ∈ {'all', 'num_levels', 'angle_step', 'scale_step', 'scale_r_step', 'scale_c_step', 'optimization', 'contrast', 'contrast_hyst', 'min_size', 'min_contrast'}

▷ **ParameterName** (output_control) . . . . . string-array ⇝ *string*
Name of the automatically determined parameter.

▷ **ParameterValue** (output_control) . . . . . number-array ⇝ *real* / integer
Value of the automatically determined parameter.

Result

If the parameters are valid, the operator determine_deformable_model_params returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameters NumLevels and Contrast are chosen such that the model contains too few points, or the input image does not contain a sufficient number of significant features, the error message 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

create_planar_uncalib_deformable_model, create_planar_calib_deformable_model, create_local_deformable_model

See also

find_planar_uncalib_deformable_model, find_planar_calib_deformable_model

Module

Matching

find_local_deformable_model (Image : ImageRectified, VectorField, DeformedContours : ModelID, AngleStart, AngleExtent, ScaleRMin, ScaleRMax, ScaleCMin, ScaleCMax, MinScore, NumMatches, MaxOverlap, NumLevels, Greediness, ResultType, GenParamName, GenParamValue : Score, Row, Column)

Find the best matches of a local deformable model in an image.

The operator find_local_deformable_model finds the best NumMatches instances of the locally deformable model ModelID in the input Image. The model must have been created previously by calling create_local_deformable_model or read_deformable_model.

The model is searched within those points of the domain of the image in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than MinScore. Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed with set_system('border_shape_models','true'), which will cause models that extend beyond the image border to be found if they achieve a score greater than MinScore. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the search will increase in this mode. Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels them model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., dilation_circle.

The operator find_local_deformable_model returns the Row and Column coordinates of the found instances. Additionally, a rectified part of the image, the respective vector field, and the contours of the found deformed model instance can be returned in ImageRectified, VectorField, and DeformedContours. By default, these iconic objects are not returned. In case they are needed the parameter ResultType should be set to *'image_rectified'*, *'vector_field'* or *'deformed_contours'*. The size of ImageRectified and VectorField is the smallest axis-aligned rectangle of the domain of the image that was used to create the local deformable model with create_local_deformable_model. The size of the rectified image and the vector field can be expanded in all directions by setting GenParamName to *'expand_border'* and the respective GenParamValue to the number of pixels. Optionally, a specific direction can be selected with *'expand_border_top'*, *'expand_border_bottom'*, *'expand_border_left'*, or *'expand_border_right'*. Please note that the returned VectorField is in absolute coordinates and can be used for convert_map_type.

The smoothness of the estimated deformation can be selected with GenParamName set to *'deformation_smoothness'*. The value for *'deformation_smoothness'* can be imagined as the size of a smoothing filter to the estimated deformation field. A too small value expects a strong deformation a too big value expects a rigid object. The minimal value for *'deformation_smoothness'* is *3*, a typical value is *11*, and the value can be increased further for only small global deformations.

Finally, the score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image.

For further explanation on the parameters used for the search of a local deformable model we refer to the description of find_planar_uncalib_deformable_model.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image in which the model should be found.

▷ **ImageRectified** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Rectified image of the found model.

▷ **VectorField** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* vector_field
Vector field of the rectification transformation.

▷ **DeformedContours** (output_object) . . . . . xld_cont-array ⇝ *object*
Contours of the found instances of the model.

▷ **ModelID** (input_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **ScaleRMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the model in row direction.
Default: 1.0
Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the model in row direction.
Default: 1.0
Suggested values: ScaleRMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleRMax >= ScaleRMin

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the model in column direction.
Default: 1.0
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the model in column direction.
Default: 1.0
Suggested values: ScaleCMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleCMax >= ScaleCMin

▷ **MinScore** (input_control) . . . . . real ⇝ *real*
Minimum score of the instances of the model to be found.
Default: 0.5
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: $0 \leq$ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer ⇝ *integer*
Number of instances of the model to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real ⇝ *real*
Maximum overlap of the instances of the model to be found.
Default: 1.0
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: $0 \leq$ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer*
Number of pyramid levels used in the matching.
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Greediness** (input_control) . . . . . real ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: $0 \leq$ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **ResultType** (input_control) . . . . . string-array ⇝ *string*
Switch for requested iconic result.
Default: []
List of values: ResultType ∈ {[], 'image_rectified', 'vector_field', 'deformed_contours'}

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The general parameter names.
Default: []
List of values: GenParamName ∈ {[], 'subpixel', 'angle_step', 'scale_r_step', 'scale_c_step', 'deformation_smoothness', 'expand_border', 'expand_border_top', 'expand_border_bottom', 'expand_border_left', 'expand_border_right'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the general parameters.
Default: []
List of values: GenParamValue ∈ {[], 'none', 'least_squares', 'least_squares_high', 'least_squares_very_high'}

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Scores of the found instances of the model.

▷ **Row** (output_control) . . . . . real-array ⇝ *real*
Row coordinates of the found instances of the model.

▷ **Column** (output_control) . . . . . real-array ⇝ *real*
Column coordinates of the found instances of the model.

Result

If the parameters are valid, the operator find_local_deformable_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

create_local_deformable_model, create_local_deformable_model_xld, read_deformable_model

Module

Matching

find_planar_calib_deformable_model (Image : : ModelID, AngleStart, AngleExtent, ScaleRMin, ScaleRMax, ScaleCMin, ScaleCMax, MinScore, NumMatches, MaxOverlap, NumLevels, Greediness, GenParamName, GenParamValue : Pose, CovPose, Score)

Find the best matches of a calibrated deformable model in an image and return their 3D pose.

The operator find_planar_calib_deformable_model finds the best NumMatches instances of the calibrated deformable model ModelID in the input image Image. The model must have been created previously by calling create_planar_calib_deformable_model or read_deformable_model. There is no need to correct any distortions in Image as the calibration data has already been provided during the model creation. The operator find_planar_calib_deformable_model returns the 3D Pose of the found instances in the camera coordinate system.

Additionally, the accuracy of the six pose parameters are returned in CovPose. By default, CovPose contains the 6 standard deviations of the pose parameters for each match. If the generic parameter *'cov_pose_mode'* was set to *'covariances'*, CovPose contains the 36 values of the complete 6×6 covariance matrix of the 6 pose parameters. Note that this reflects only an inner accuracy from which the real accuracy of the pose may differ. Finally, the score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image.

For further explanation on the planar deformable model and its parameters we refer to the description of find_planar_uncalib_deformable_model.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image in which the model should be found.

▷ **ModelID** (input_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.78, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.78
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.78, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **ScaleRMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the model in row direction.
Default: 1.0
Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the model in row direction.
Default: 1.0
Suggested values: ScaleRMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleRMax >= ScaleRMin

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the model in column direction.
Default: 1.0
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the model in column direction.
Default: 1.0
Suggested values: ScaleCMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleCMax >= ScaleCMin

▷ **MinScore** (input_control) . . . . . real ⇝ *real*
Minimum score of the instances of the model to be found.
Default: 0.5
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer ⇝ *integer*
Number of instances of the model to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real ⇝ *real*
Maximum overlap of the instances of the model to be found.
Default: 1.0
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumLevels** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Greediness** (input_control) . . . . . real ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The general parameter names.
Default: []
List of values: GenParamName ∈ {[], 'subpixel', 'angle_change_restriction', 'aniso_scale_change_restriction', 'angle_step', 'scale_r_step', 'scale_c_step', 'cov_pose_mode'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the general parameters.
Default: []
List of values: GenParamValue ∈ {[], 'least_squares', 'least_squares_high', 'least_squares_very_high', 'standard_deviations', 'covariances'}

▷ **Pose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
Pose of the object.

▷ **CovPose** (output_control) . . . . . real-array ⇝ *real*
6 standard deviations or 36 covariances of the pose parameters.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the model.

Result

If the parameters are valid, the operator find_planar_calib_deformable_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

create_planar_calib_deformable_model, read_deformable_model

Module

Matching

find_planar_uncalib_deformable_model (Image : : ModelID, AngleStart, AngleExtent, ScaleRMin, ScaleRMax, ScaleCMin, ScaleCMax, MinScore, NumMatches, MaxOverlap, NumLevels, Greediness, GenParamName, GenParamValue : HomMat2D, Score)

Find the best matches of a planar projective invariant deformable model in an image.

The operator find_planar_uncalib_deformable_model finds the best NumMatches instances of the perspectively distorted deformable model ModelID in the input image Image. The model must have been created previously by calling create_planar_uncalib_deformable_model or read_deformable_model.

HomMat2D determines the projective transformation (homography), which describes the position of the found matches. In case several objects are found, the different homographies are concatenated. Then, a single homography can be extracted using tuple_select_range(HomMat2D,Index*9,(Index+1)). The different results are sorted according to their Score in descending order.

The row and column coordinates of the origin of the deformable model within the search image can be determined calling projective_trans_pixel(HomMat2D,0,0,Row,Column). Usually the origin of the model is the center of gravity of the image region used to create the deformable

model calling create_planar_uncalib_deformable_model. The origin can be modified using set_deformable_model_origin.

The model contours of found instances can be visualized using projective_trans_contour_xld with the HomMat2D and the original model contour, which has been extracted previously using get_deformable_model_contours.

The Score is a number between 0 and 1 and may indicate how much of the model is visible within the image.

Example: Half of the model is occluded in the search image. As a result, the Score of this match can not exceed *0.5*.

Input parameters in detail

Image and its domain: The domain of the image Image determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the deformable model with create_planar_uncalib_deformable_model. A different origin set with set_deformable_model_origin is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than MinScore (see below). Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed with set_system('border_shape_models','true'), which will cause models that extend beyond the image border to be found if they achieve a score greater than MinScore. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the search will increase in this mode. Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels the model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., dilation_circle.

Angle and Scale parameters: The parameters AngleStart, AngleExtent, ScaleRMin, ScaleRMax, ScaleCMin and ScaleCMax are used to specify a basic range of up to an anisotropic transformation that is exhaustively searched on the top level of the image pyramid. The parameters AngleStart and AngleExtent determine the range of possible rotations in which the model is exhaustively searched. ScaleRMin, ScaleRMax, ScaleCMin, and ScaleCMax determine the range of possible anisotropic scales that are exhaustively searched in the image. A scale of 1 in both scale factors corresponds to the original size of the model.

The operator find_planar_uncalib_deformable_model may find objects outside this range, e.g., when the object is perspectively distorted. Hence, the range parameters are a kind of suggestion for the search algorithm. Starting from this, certain models in a wider range of transformations can be detected, depending on the used pyramid levels, but also on the model/image content. It is important to note that, e.g., small scale changes can be tolerated without the need to specify a scale range, leading to faster execution times.

Often, it is not necessary to use an anisotropic scaling to find the object on the top level of the pyramid. In these cases, ScaleCMin and ScaleCMax should be set to *1.0*. The search is then performed with isotropic scaling only, which is much faster. If the object should be detected despite severe perspective distortions anisotropic scaling is required. Here, ScaleRMin and ScaleRMax specify the anisotropic scaling in row, ScaleCMin and ScaleCMax in column direction.

Note that the transformations are treated internally such that the scalings are applied first, followed by the rotation. Therefore, the model should usually be aligned such that it appears horizontally or vertically in the model image.

Additionally, the operator find_planar_uncalib_deformable_model processes the parameters *'angle_step'*, *'scale_r_step'* and *'scale_c_step'* which are set by the operator create_planar_uncalib_deformable_model during the model creation, or, as described below, can be set with the generic parameters GenParamName and GenParamValue.

The parameter *'angle_step'* determines the step size within the selected range of angles. *'angle_step'* should be chosen based on the size of the object. Smaller models do not have many different discrete rotations in the image, and hence *'angle_step'* should be chosen larger for smaller models. If AngleExtent is not an integer multiple of *'angle_step'*, *'angle_step'* is modified accordingly. The parameters *'scale_r_step'* and *'scale_c_step'* determine the step size within the selected range of scales. Like *'angle_step'*, *'scale_r_step'* and *'scale_c_step'* should be chosen based on the size of the object. If the respective range of scales is not an integer multiple of *'scale_r_step'* and *'scale_c_step'*, *'scale_r_step'* and *'scale_c_step'* are modified accordingly.

MinScore: The parameter MinScore determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger MinScore is chosen, the faster the search is. If the model can be expected never to be occluded in the images, MinScore may be set as high as 0.8 or even 0.9.

NumMatches: The maximum number of instances to be found can be determined with NumMatches. If more than NumMatches instances with a score greater than MinScore are found in the image, only the best NumMatches instances are returned. If fewer than NumMatches are found, only that number is returned, i.e., the parameter MinScore takes precedence over NumMatches. If all model instances exceeding MinScore in the image should be found, NumMatches must be set to *0*. In rare cases, NumMatches must be set to a higher value than the required number of matches. This is the case if, for instance, a small MinScore is set.

When tracking the matches through the image pyramid, on each level, some less promising matches are rejected based on NumMatches. Thus, it is possible that some matches are rejected that would have had a higher score on the lowest pyramid level. Due to this, for example, the found match for NumMatches set to *1* might be different from the match with the highest score returned when setting NumMatches to *0* or > *1*.

If multiple objects with a similar score are expected, but only the one with the highest score should be returned, it might be preferable to raise NumMatches, and then select the match with the highest score.

MaxOverlap: If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter MaxOverlap determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than MaxOverlap only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see smallest_rectangle2) of the found instances. If MaxOverlap=*0*, the found instances may not overlap at all, while for MaxOverlap=*1* all instances are returned.

GenParamName, GenParamValue: With the generic parameters GenParamName and GenParamValue it is possible to adjust parameters that typically do not have to be set by the user. By default the pose is extracted with high subpixel accuracy (*'least_squares_very_high'*) through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. However, if no high accuracy is required by an application, the subpixel precise extraction can be reduced or switched off as it increases the processing time. Here, *'subpixel'* must be passed in GenParamName and *'none'*, *'least_squares'*, *'least_squares_high'* for GenParamValue. A further use of GenParamName and GenParamValue is to override the discretization steps of the parameter space *'angle_step'*, *'scale_r_step'* and *'scale_c_step'* that have been defined when the model was created in create_planar_uncalib_deformable_model.

As described in create_planar_uncalib_deformable_model the deformable matching algorithm searches exhaustively a basic set of parameters that are specified with AngleStart,AngleExtent, ScaleRMin,ScaleRMax,ScaleCMin and ScaleCMax. However, to allow a detection even when the object is imaged under perspective distortion, an additional transformation is estimated. This additional transformation transforms the model from the original search range to a bigger perspectively distorted one. By allowing perspective distortions, the risk of false positives is also increased. One possible use of the parameter GenParamName is to help discarding false positives that occur, if for instance a small score was specified in MinScore and the image contains significant clutter with similar shape as the model.

To restrict arbitrary perspective matches from occurring, the values *'angle_change_restriction'* and *'aniso_scale_change_restriction'* can be used in GenParamName. With *'angle_change_restriction'* the maximal tolerated angular distortion can be restricted. As default value $\frac{\pi}{2}$ is set, which allows arbitrary distortion. By setting *'angle_change_restriction'* to *0*, no distortion is allowed at all. The set value should be within the interval [0, $\frac{\pi}{2}$]. This parameter tests, if the angle of 90 degree at the corners of the axis-aligned rectangle around the model points is changed by more than the corresponding GenParamValue for the found instance of the model. Note that this parameter helps to restrict both affine (a shear mapping) and perspective parts of the transformation. As an example, with *'angle_change_restriction'* a square-like model can be prevented to match with a parallelogram or an arbitrary trapezium.

With the parameter *'aniso_scale_change_restriction'* the ratio of anisotropic scaling can be restricted (the smaller scale factor divided by the bigger scale factor). The value of this parameter ranges from the default value *0.0*, where arbitrary distortion is allowed, to *1.0*, where no distortion is allowed. One typical use for this parameter is to restrict for instance a square-like model to deform to a rectangular model.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the deformable model was created with create_planar_uncalib_deformable_model. If NumLevels is set to *0*, the number of pyramid levels specified in create_planar_uncalib_deformable_model is used.

Greediness: The parameter Greediness determines how "greedily" the search should be carried out. If Greediness = *0*, a safe search heuristic is used, which finds the model if it is visible in the image and the other parameters are set appropriately. However, the search will be relatively time consuming in this case. If Greediness=*1*, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For Greediness=*1*, the maximum search speed is achieved. In almost all cases, the deformable model will be found for Greediness=*0.9*.

Output parameters in detail

HomMat2D: The projective transformation (homographies) that encode the position of the found instances of the model are returned in HomMat2D. In case that multiple objects are found, the different homographies are concatenated. A single homography can easily be extracted by tuple_select_range(HomMat2D,Index*9,(Index+1)). The different detection results are sorted in decreasing order of Score. The row and column coordinates are the coordinates of the origin of the deformable model in the search image, which can be found by calling projective_trans_pixel(HomMat2D,0,0,Row,Column). By default, the origin is the center of gravity of the domain (region) of the image that was used to create the deformable model with create_planar_uncalib_deformable_model. A different origin can be set with set_deformable_model_origin. For visualization purposes, the model contours that are extracted by get_deformable_model_contours can be projected to the found location given HomMat2D with projective_trans_contour_xld.

Score: Additionally, the score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image in which the model should be found.

▷ **ModelID** (input_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.78
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **ScaleRMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the model in row direction.
Default: 1.0
Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the model in row direction.
Default: 1.0
Suggested values: ScaleRMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleRMax >= ScaleRMin

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the model in column direction.
Default: 1.0
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the model in column direction.
Default: 1.0
Suggested values: ScaleCMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleCMax >= ScaleCMin

▷ **MinScore** (input_control) . . . . . real ⇝ *real*
Minimum score of the instances of the model to be found.
Default: 0.5
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer ⇝ *integer*
Number of instances of the model to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real ⇝ *real*
Maximum overlap of the instances of the model to be found.
Default: 1.0
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumLevels** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Greediness** (input_control) . . . . . real ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
The general parameter names.
Default: []
List of values: GenParamName ∈ {[], 'subpixel', 'angle_change_restriction', 'aniso_scale_change_restriction', 'angle_step', 'scale_r_step', 'scale_c_step'}

▷ **GenParamValue** (input_control) . . . . . integer-array ⇝ *integer* / real / string
Values of the general parameters.
Default: []
List of values: GenParamValue ∈ {[], 'none', 'least_squares', 'least_squares_high', 'least_squares_very_high'}

▷ **HomMat2D** (output_control) . . . . . hom_mat2d(-array) ⇝ *real*
Homographies between model and found instances.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the model.

Result

If the parameters are valid, the operator find_planar_uncalib_deformable_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

create_planar_uncalib_deformable_model, read_deformable_model

Alternatives

find_planar_calib_deformable_model

Module

Matching

get_deformable_model_contours (: ModelContours : ModelID, Level :)

Return the contour representation of a deformable model.

The operator get_deformable_model_contours returns a representation of the deformable model ModelID as XLD contours in ModelContours. The parameter Level determines for which pyramid level of the model the contour representation should be returned. The contours can be used, for example, to visualize the found instances of the model in an image.

In case that the model was generated by create_planar_calib_deformable_model_xld, the contours by default are returned in the world coordinate system in metric units. Here, the contours must be transformed by the returned pose for visualizing a match.

In all other cases, the contours of the model by default are returned in the image coordinate system in pixel units. In the calibrated case this system corresponds to the rectified image coordinate system. The rectified image coordinate system is the coordinate system of an image one would obtain by change_radial_distortion_image when using the rectified camera parameters. The rectified camera parameters can be queried by get_deformable_model_params. It should be noted that the position of ModelContours is normalized such that the reference point of the model (see set_deformable_model_origin) lies at the pixel position $(0, 0)$. Hence, the contours simply need to be transformed by the found homography in the image.

The default behavior for the calibrated case can be changed with the generic parameter *'get_deformable_model_contours_coord_system'* of the operator set_deformable_model_param.

Parameters

▷ **ModelContours** (output_object) ..... xld_cont-array ⇝ *object*
Contour representation of the deformable model.

▷ **ModelID** (input_control) ..... deformable_model ⇝ *handle*
Handle of the model.

▷ **Level** (input_control) ..... integer ⇝ *integer*
Pyramid level for which the contour representation should be returned.
Default: 1
Suggested values: Level ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Restriction: Level >= 1

Result

If the handle of the model is valid, the operator get_deformable_model_contours returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_planar_uncalib_deformable_model, create_planar_uncalib_deformable_model_xld, read_deformable_model, create_planar_calib_deformable_model, create_planar_calib_deformable_model_xld, create_local_deformable_model, create_local_deformable_model_xld

See also

set_deformable_model_param, find_planar_uncalib_deformable_model, find_planar_calib_deformable_model

Module

Matching

get_deformable_model_origin (: : ModelID : Row, Column)

Return the origin (reference point) of a deformable model.

The operator get_deformable_model_origin returns the origin (reference point) of the deformable model ModelID. The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the deformable model with create_planar_uncalib_deformable_model or create_planar_calib_deformable_model. Hence, an origin of (0,0) means that the center of gravity of the domain of the model image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity.

Parameters

▷ **ModelID** (input_control) ..... deformable_model ⇝ *handle*
Handle of the model.

▷ **Row** (output_control) ..... point.y ⇝ *real*
Row coordinate of the origin of the deformable model.

▷ **Column** (output_control) ..... point.x ⇝ *real*
Column coordinate of the origin of the deformable model.

Result

If the handle of the model is valid, the operator get_deformable_model_origin returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_planar_uncalib_deformable_model, read_deformable_model, set_deformable_model_origin

Possible Successors

find_planar_uncalib_deformable_model

See also

area_center

Module

Matching

get_deformable_model_params (: : ModelID, GenParamName : GenParamValue)

Return the parameters of a deformable model.

The operator get_deformable_model_params allows to query parameters of the deformable model. The names of the desired parameters are passed in the generic parameter GenParamName, the corresponding values are returned in GenParamValue.

The following parameters can be queried:

'cam_param': Internal parameters of the camera that is used for the calibrated case.

'cam_param_rect': Rectified internal parameters of the camera that is used for the calibrated case.

'reference_pose': Reference position and orientation of the calibrated deformable model. The returned pose describes the model plane that the user provided.

'model_pose': Reference position and orientation of the deformable model. The returned pose describes the pose of the internally used 3D metric model that was used for model generation. The pose refers to the model's center of gravity if not explicitly set with set_deformable_model_origin.

'angle_start': The smallest rotation angle of the model.

'angle_extent': The extent of the rotation angle of the model.

'angle_step': The angle step length of the model.

'scale_r_min': The minimum scale of the pattern in row direction.

'scale_r_max': The maximum scale of the pattern in row direction.

'scale_r_step': The scale step of the pattern in row direction.

'scale_c_min': The minimum scale of the pattern in column direction.

'scale_c_max': The maximum scale of the pattern in column direction.

'scale_c_step': The scale step of the pattern in column direction.

'num_levels': User-specified number of pyramid levels.

'optimization': Kind of optimization by reducing the number of model points.

'min_size': Minimum size of the deformable model edge.

'min_contrast': Minimum contrast of the objects in the search images.

'metric': Match metric.

'model_row': Row coordinate of origin in the radial undistorted model image.

'model_col': Column coordinate of origin in the radial undistorted model image.

'model_type': The type of the model: *'planar_uncalib'*, *'planar_calib'* or *'local'*.

'created_from_xld': Was the model created from XLD or from an image: *'created_from_xld'* or *'created_from_image'*.

'get_deformable_model_contours_coord_system': Coordinate system (*'image'* or *'world'*) in which contours are returned when calling the operator `get_deformable_model_contours`.

A detailed description of the parameters can be looked up with the operator `create_planar_uncalib_deformable_model` and `create_planar_calib_deformable_model` or with the operator `get_deformable_model_contours`.

Note that although the parameter `Contrast` can be determined automatically during the model creation, its value cannot be queried using `get_deformable_model_params`. Instead, the operator `determine_deformable_model_params` should be used to retrieve its value.

It is possible to query the values of several parameters with a single operator call by passing a tuple containing the names of all desired parameters to `GenParamName`. As a result a tuple of the same length with the corresponding values is returned in `GenParamValue`. Note that this is solely possible for parameters that return only a single value.

Parameters

▷ **`ModelID`** (input_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that are to be queried for the deformable model.
Default: 'angle_start'
List of values: `GenParamName` ∈ {'cam_param', 'cam_param_rect', 'angle_start', 'angle_extent', 'angle_step', 'scale_r_min', 'scale_r_max', 'scale_r_step', 'scale_c_min', 'scale_c_max', 'scale_c_step', 'optimization', 'metric', 'min_contrast', 'num_levels', 'min_size', 'reference_pose', 'model_pose', 'model_row', 'model_col', 'model_type', 'created_from_xld', 'get_deformable_model_contours_coord_system'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
Values of the generic parameters.

Result

If the parameters are valid, the operator `get_deformable_model_params` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_planar_uncalib_deformable_model, create_planar_calib_deformable_model, create_local_deformable_model

Possible Successors

find_planar_uncalib_deformable_model, find_planar_calib_deformable_model

See also

set_deformable_model_param

Module

Matching

read_deformable_model (: : FileName : ModelID)

Read a deformable model from a file.

The operator read_deformable_model reads a deformable model from the file FileName, which has been written with write_deformable_model. The default HALCON file extension for the deformable model is 'dfm'.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
File name.
File extension: .dfm

▷ **ModelID** (output_control) ..... deformable_model ⇝ *handle*
Handle of the model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

find_planar_uncalib_deformable_model, find_planar_calib_deformable_model, find_local_deformable_model

See also

create_planar_uncalib_deformable_model, create_planar_calib_deformable_model, create_local_deformable_model

Module

Matching

serialize_deformable_model (: : ModelID : SerializedItemHandle)

Serialize a deformable model.

serialize_deformable_model serializes the data of a deformable model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_deformable_model is converted to a serialized item. The deformable model is defined by the handle ModelID. The serialized deformable model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_deformable_model.

Parameters

▷ **ModelID** (input_control) . . . . . deformable_model ⇝ *handle*
Handle of a model to be saved.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_deformable_model` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_planar_uncalib_deformable_model`, `create_planar_calib_deformable_model`, `create_local_deformable_model`

Possible Successors

`fwrite_serialized_item`, `send_serialized_item`, `deserialize_deformable_model`

Module

Matching

set_deformable_model_origin (: : ModelID, Row, Column :)

Set the origin (reference point) of a deformable model.

The operator `set_deformable_model_origin` sets the origin (reference point) of the deformable model `ModelID` to a new value. The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the deformable model with `create_planar_uncalib_deformable_model`. Hence, an origin of (0,0) means that the center of gravity of the domain of the deformable image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity. If a deformable model was created by `create_planar_calib_deformable_model` the 3D pose of the origin changes by the respective translation. Hence, further calls of `find_planar_calib_deformable_model` will include the pose offset. If a deformable model was created by `create_planar_calib_deformable_model_xld` the 3D pose of the origin changes directly by the offset. This means that the offsets are interpreted in world coordinates. In this case, the row coordinate corresponds to the y world coordinate (from top to down) and the column coordinate corresponds to the x world coordinate (from left to right).

Parameters

▷ **ModelID** (input_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

▷ **Row** (input_control) . . . . . point.y ⇝ *real*
Row coordinate of the origin of the deformable model.

▷ **Column** (input_control) . . . . . point.x ⇝ *real*
Column coordinate of the origin of the deformable model.

Result

If the handle of the model is valid, the operator `set_deformable_model_origin` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_planar_uncalib_deformable_model`, `create_planar_calib_deformable_model`, `create_local_deformable_model`, `read_deformable_model`

Possible Successors

`find_planar_uncalib_deformable_model`, `find_planar_calib_deformable_model`, `get_deformable_model_origin`

See also

`area_center`

Module

Matching

set_deformable_model_param (: : ModelID, GenParamName, GenParamValue :)

Set selected parameters of the deformable model.

The operator `set_deformable_model_param` sets the selected parameters `GenParamName` in the deformable model `ModelID`. The following parameters can be modified:

'min_contrast'

Sets the minimum contrast of the object in the search images for the deformable model `ModelID`. Thereby, the value of *'min_contrast'* that was originally set, e.g., with `create_planar_uncalib_deformable_model`, is overwritten for the deformable model `ModelID`. Note that if the deformable model `ModelID` was read from file and if this file should be changed as well, the deformable model `ModelID` must again be written to file after the execution of the operator `set_deformable_model_param`.

'get_deformable_model_contours_coord_system'

Sets the coordinate system in which the contours are returned when calling the operator `get_deformable_model_contours`. If *'world'* is passed, the contours are returned in the world coordinate system in metric units. Note that this is only possible for the calibrated case. If *'image'* is passed, the contours are returned in the image coordinate system in pixel units. In the calibrated case this system corresponds to the rectified image coordinate system. The rectified image coordinate system is the coordinate system of an image one would obtain by `change_radial_distortion_image` when using the rectified camera parameters. The rectified camera parameters can be queried by `get_deformable_model_params`. If the model was created by `create_planar_calib_deformable_model_xld` the value of *'get_deformable_model_contours_coord_system'* is *'world'* by default. In all other cases it is *'image'* by default.

Parameters

▷ **ModelID** (input_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Parameter names.
List of values: `GenParamName` ∈ {'min_contrast', 'get_deformable_model_contours_coord_system'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Parameter values.
Suggested values: `GenParamValue` ∈ {5, 6, 7, 8, 9, 10, 'image', 'world'}

Result

If the parameters are valid, the operator `set_deformable_model_param` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_planar_uncalib_deformable_model, create_planar_calib_deformable_model, create_local_deformable_model, read_deformable_model

Possible Successors

find_planar_uncalib_deformable_model, find_planar_calib_deformable_model

See also

get_deformable_model_contours, get_deformable_model_params

Module

Matching

set_local_deformable_model_metric (Image, VectorField : : ModelID, Metric :)

Set the metric of a local deformable model that was created from XLD contours.

set_local_deformable_model_metric changes the value of the parameter Metric that was selected during the creation of the deformable model ModelID from XLD contours. Because no information about the polarity of the model contours is available for XLD contours, the polarity is determined based on a representative Image. For this, the model contours are mapped to the respective position where the object appears in the Image, which is done with the VectorField which must be of the semantic type 'vector_field_absolute' and which can be obtained by a previous call to find_local_deformable_model. Here, the returned VectorField is used for the deformable transformation. Note, that the *'expand_border'* option should not be used in find_local_deformable_model, because then the returned VectorField has the wrong image size.

The parameter Metric then determines the conditions under which the model is recognized later in the search image. If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, e.g., the model is a bright object on a dark background, the object is found only if it is also brighter than the background. If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_local_deformable_model will increase slightly in this case.

It must be ensured that the object contours in the Image have the same (or inverse) polarity as the object contours in the image in which the object must be searched later. Especially, the object must not be occluded in the Image and the background must be either brighter than the object or darker. Otherwise, the determined polarity of the model contour will not represent the polarity of the object contour during the search. Note that only the polarity of the contours is determined, not their contrast. Note also that the polarity is determined from a single-channel image, only. If a multi-channel image is passed in Image, only the first channel will be used (and no error message will be returned).

A typical proceeding is to read the XLD contours from file. Since these XLD contours do not provide polarity information, the model must be created from the XLD contours by setting the parameter Metric to *'ignore_local_polarity'*. Then, in a first search image the model is recognized. The transformation that maps the model contours to the position of the object in the search image is provided in VectorField, which is of the semantic type 'vector_field_absolute'. If the matching result is correct, the value of the parameter Metric can be changed, e.g., to *'use_polarity'*. This leads to a faster and more robust recognition in the following images.

Attention

set_local_deformable_model_metric can only be used with deformable models that were created from XLD contours.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image used for the determination of the polarity.

▷ **VectorField** (input_object) . . . . . singlechannelimage ⇝ *object :* vector_field
Vector field of the local deformation.

▷ **ModelID** (input_control) . . . . . deformable_model ⇝ *handle*
Handle of the model.

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity'}

Result

If the parameters are valid, the operator set_local_deformable_model_metric returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_local_deformable_model_xld, find_local_deformable_model

Possible Successors

find_local_deformable_model

See also

create_local_deformable_model_xld

Module

Matching

set_planar_calib_deformable_model_metric (Image : : ModelID, Pose, Metric :)

Set the metric of a planar calibrated deformable model that was created from XLD contours.

set_planar_calib_deformable_model_metric changes the value of the parameter Metric that was selected during the creation of the deformable model ModelID from XLD contours. Because no information about the polarity of the model contours is available for XLD contours, the polarity is determined based on a representative Image. For this, the model contours are mapped to the respective position where the object appears in the Image using Pose, which can be obtained by a previous call to find_planar_calib_deformable_model.

The parameter Metric then determines the conditions under which the model is recognized later in the search image. If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, e.g., the model is a bright object on a dark background, the object is found only if it is also brighter than the background. If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_planar_calib_deformable_model will increase slightly in this case.

It must be ensured that the object contours in the Image have the same (or inverse) polarity as the object contours in the image in which the object must be searched later. Especially, the object must not be occluded in the Image

and the background must be either brighter than the object or darker. Otherwise, the determined polarity of the model contour will not represent the polarity of the object contour during the search. Note that only the polarity of the contours is determined, not their contrast. Note also that the polarity is determined from a single-channel image, only. If a multi-channel image is passed in `Image`, only the first channel will be used (and no error message will be returned).

A typical proceeding is to read the XLD contours from file. Since these XLD contours do not provide polarity information, the model must be created from the XLD contours by setting the parameter `Metric` to *'ignore_local_polarity'*. Then, in a first search image the model is recognized. The transformation that maps the model contours to the position of the object in the search image `Pose` can be determined from the matching result. To verify the match interactively, the model contours can be mapped to this position. If the matching result is correct, the value of the parameter `Metric` can be changed, e.g., to *'use_polarity'*. This leads to a faster and more robust recognition in the following images.

Attention

`set_planar_calib_deformable_model_metric` can only be used with deformable models that were created from XLD contours.

Parameters

- ▷ **`Image`** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image used for the determination of the polarity.
- ▷ **`ModelID`** (input_control) . . . . . deformable_model ⇝ *handle*
  Handle of the model.
- ▷ **`Pose`** (input_control) . . . . . pose ⇝ *real* / integer
  Pose of the model in the image.
  Number of elements: 7
- ▷ **`Metric`** (input_control) . . . . . string ⇝ *string*
  Match metric.
  Default: 'use_polarity'
  List of values: `Metric` ∈ {'use_polarity', 'ignore_global_polarity'}

Result

If the parameters are valid, the operator `set_planar_calib_deformable_model_metric` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_planar_calib_deformable_model_xld`, `find_planar_calib_deformable_model`

Possible Successors

`find_planar_calib_deformable_model`

See also

`create_planar_calib_deformable_model_xld`

Module

Matching

`set_planar_uncalib_deformable_model_metric` (`Image` : : `ModelID`, `HomMat2D`, `Metric` :)

Set the metric of a planar uncalibrated deformable model that was created from XLD contours.

set_planar_uncalib_deformable_model_metric changes the value of the parameter Metric that was selected during the creation of the deformable model ModelID from XLD contours. Because no information about the polarity of the model contours is available for XLD contours, the polarity is determined based on a representative Image. For this, the model contours are mapped to the respective position where the object appears in the Image using HomMat2D, which can be obtained by a previous call to find_planar_uncalib_deformable_model.

The parameter Metric then determines the conditions under which the model is recognized later in the search image. If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, e.g., the model is a bright object on a dark background, the object is found only if it is also brighter than the background. If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_planar_uncalib_deformable_model will increase slightly in this case.

It must be ensured that the object contours in the Image have the same (or inverse) polarity as the object contours in the image in which the object must be searched later. Especially, the object must not be occluded in the Image and the background must be either brighter than the object or darker. Otherwise, the determined polarity of the model contour will not represent the polarity of the object contour during the search. Note that only the polarity of the contours is determined, not their contrast. Note also that the polarity is determined from a single-channel image, only. If a multi-channel image is passed in Image, only the first channel will be used (and no error message will be returned).

A typical proceeding is to read the XLD contours from file. Since these XLD contours do not provide polarity information, the model must be created from the XLD contours by setting the parameter Metric to *'ignore_local_polarity'*. Then, in a first search image the model is recognized. The transformation that maps the model contours to the position of the object in the search image (HomMat2D) can be determined from the matching result. To verify the match interactively, the model contours can be mapped to this position. If the matching result is correct, the value of the parameter Metric can be changed, e.g., to *'use_polarity'*. This leads to a faster and more robust recognition in the following images.

Attention

set_planar_uncalib_deformable_model_metric can only be used with deformable models that were created from XLD contours.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
  Input image used for the determination of the polarity.
- ▷ **ModelID** (input_control) ..... deformable_model ⇝ *handle*
  Handle of the model.
- ▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
  Transformation matrix.
- ▷ **Metric** (input_control) ..... string ⇝ *string*
  Match metric.
  Default: 'use_polarity'
  List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity'}

Result

If the parameters are valid, the operator set_planar_uncalib_deformable_model_metric returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_planar_uncalib_deformable_model_xld,
find_planar_uncalib_deformable_model

Possible Successors

find_planar_uncalib_deformable_model

See also

create_planar_uncalib_deformable_model_xld

Module

Matching

write_deformable_model (: : ModelID, FileName :)

Write a deformable model to a file.

The operator write_deformable_model writes a deformable model to the file FileName. The model can be read again with read_deformable_model. The default HALCON file extension for the deformable model is 'dfm'.

Parameters

▷ **ModelID** (input_control) . . . . . deformable_model ⇝ *handle*
Handle of a model to be saved.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
The path and filename of the model to be saved.
File extension: .dfm

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_planar_uncalib_deformable_model, create_planar_calib_deformable_model,
create_local_deformable_model

Module

Matching

18.4 Descriptor-Based

clear_descriptor_model (: : ModelID :)

Free the memory of a descriptor model.

The operator clear_descriptor_model frees the memory of a descriptor model that was created by create_calib_descriptor_model or create_uncalib_descriptor_model. After calling clear_descriptor_model, the model can no longer be used. The handle ModelID becomes invalid.

Parameters

▷ **ModelID** (input_control) . . . . . descriptor_model(-array) ⇝ *handle*
Handle of the descriptor model.

Result

If the handle of the model is valid, the operator clear_descriptor_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

create_uncalib_descriptor_model, create_calib_descriptor_model

Module

Matching

create_calib_descriptor_model (Template : : CamParam, ReferencePose, DetectorType, DetectorParamName, DetectorParamValue, DescriptorParamName, DescriptorParamValue, Seed : ModelID)

Create a descriptor model for calibrated perspective matching.

The operator create_calib_descriptor_model prepares a descriptor model, which is used for calibrated descriptor-based matching, from a template image that is passed in Template (usually the domain of the image is reduced to an ROI showing only the object of interest). Note that the part of the object that is visible in the Template image needs to be planar.

The internal camera parameters CamParam and the ReferencePose, which describes the 3D position and orientation of the object plane shown in Template, are used to internally calculate a model coordinate system. The origin of this model coordinate system is obtained by projecting the center of gravity of an internally rectified version of the template image onto the object plane. The axes of the model coordinate system are built such that they are parallel to the axes of Reference Pose. To obtain a descriptor model from the template image, the points extracted by the detector in the internally rectified template image are projected onto the object plane as well. Their coordinates in the model coordinate system are stored as world coordinates in the model. This model then can be used for the pose estimation of a searched object as described with find_calib_descriptor_model.

A descriptor model created by the use of create_calib_descriptor_model can also be used in find_uncalib_descriptor_model to determine a homography. In contrast, it is not possible to use a model created by create_uncalib_descriptor_model in find_calib_descriptor_model.

The descriptor model describes a set of points of interest. It stores their locations and discriminative descriptions of their local gray value neighborhood. The interest point extraction is parametrized by DetectorType, DetectorParamName and DetectorParamValue. The respective descriptor around the interest points is parametrized by DescriptorParamName and DescriptorParamValue. The parameter Seed seeds the random number generator, which is used during the construction of the descriptor implemented with *randomized ferns*. The returned ModelID is a reference to the generated descriptor model. For further explanation on the descriptor model and its parameter we refer to the description of create_uncalib_descriptor_model.

The parameters and the location of the final descriptor points can be determined with get_descriptor_model_params and get_descriptor_model_points.

create_calib_descriptor_model stores the detector type, detector parameters and descriptor parameters, which are used in every succeeding call of find_calib_descriptor_model or find_uncalib_descriptor_model. The reference point (origin) of the model is the center of gravity of the template's ROI. Its coordinates can be changed by set_descriptor_model_origin.

Parameters

▷ **Template** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **CamParam** (input_control) ..... campar ⇝ *real* / integer / string
The parameters of the internal orientation of the camera.

▷ **ReferencePose** (input_control) . . . . . pose ⇝ *real* / integer
The reference pose of the object in the reference image.

▷ **DetectorType** (input_control) . . . . . string ⇝ *string*
The type of the detector.
Default: 'lepetit'
List of values: DetectorType ∈ {'lepetit', 'harris', 'harris_binomial'}

▷ **DetectorParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
The detector's parameter names.
Default: []
List of values: DetectorParamName ∈ {'alpha', 'check_neighbor', 'mask_size_grd', 'mask_size_smooth', 'min_check_neighbor_diff', 'min_score', 'radius', 'sigma_grad', 'sigma_smooth', 'subpix', 'threshold'}

▷ **DetectorParamValue** (input_control) . . . . . attribute.value-array ⇝ *integer* / real / string
Values of the detector's parameters.
Default: []
Suggested values: DetectorParamValue ∈ {0.08, 1, 1.2, 3, 15, 30, 1000, 'on', 'off'}

▷ **DescriptorParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
The descriptor's parameter names.
Default: []
List of values: DescriptorParamName ∈ {'depth', 'max_rot', 'max_scale', 'min_rot', 'min_scale', 'number_ferns', 'patch_size', 'tilt'}

▷ **DescriptorParamValue** (input_control) . . . . . attribute.value-array ⇝ *integer* / real / string
Values of the descriptor's parameters.
Default: []
Suggested values: DescriptorParamValue ∈ {0.5, 1.4, 11, 21, 30, -180, 180, 'on', 'off'}

▷ **Seed** (input_control) . . . . . integer ⇝ *integer*
The seed for the random number generator.
Default: 42

▷ **ModelID** (output_control) . . . . . descriptor_model ⇝ *handle*
The handle to the descriptor model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

points_lepetit, points_harris, reduce_domain

Possible Successors

get_descriptor_model_params, find_calib_descriptor_model

See also

get_descriptor_model_params, find_calib_descriptor_model

References

V. Lepetit and P. Fua: "Keypoint Recognition using Randomized Trees." IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 28, Nr. 9, pp. 1465-1479, 2006.

M. Ozuysal, P. Fua, and V. Lepetit: "Fast Keypoint Recognition in Ten Lines of Code." In Proceedings of Conference on Computer Vision and Pattern Recognition, 2007.

Module

Matching

create_uncalib_descriptor_model (Template : : DetectorType, DetectorParamName, DetectorParamValue, DescriptorParamName, DescriptorParamValue, Seed : ModelID)

Prepare a descriptor model for interest point matching.

The operator create_uncalib_descriptor_model prepares a descriptor model of an image region, which is passed in the image Template, that can be used for descriptor-based matching. By a subsequent call of find_uncalib_descriptor_model a projective 2D transformation (homography) from Template to a search image can be obtained. The center of gravity of the domain region in Template is used as an origin of the model. In contrast to create_calib_descriptor_model, no calibrated camera is needed and hence the result of a subsequent matching is a 2D projection. Note that the part of the object that is visible in the Template image needs to be planar.

The descriptor model describes a set of points of interest. It stores their locations and discriminative descriptions of their local gray value neighborhood. The interest point extraction is parametrized by DetectorType, DetectorParamName and DetectorParamValue. The respective descriptor around the interest points is parametrized by DescriptorParamName and DescriptorParamValue. The parameter Seed seeds the random number generator, which is used during the construction of the descriptor implemented with *randomized ferns*. The returned ModelID is a reference to the generated descriptor model. This model can be used for an efficient detection of instances of the learned template with find_uncalib_descriptor_model, allowing perspective transformation between model and search image. As descriptor-based matching relies on the existence of stable and discriminative points of interest, the object to be detected must be textured, but not in a repetitive way.

Detector parameters

As mentioned, the detector is used for the extraction of stable interest points within the image. By the parameter DetectorType, it is possible to select the interest point operator to be used. Currently, points_lepetit, points_harris, and its binomial approximation points_harris_binomial are supported ('lepetit', 'harris', 'harris_binomial'). In case of templates or search images with a low contrast, one of the Harris point operators should be used. Depending on the selected DetectorType, appropriate parameter names and values can be set in DetectorParamName and DetectorParamValue.

Possible parameter names for DetectorParamName and corresponding default values are:

'lepetit': ['radius', 'check_neighbor', 'min_check_neighbor_diff', 'min_score', 'subpix']
[3, 1, 15, 30, 'interpolation']

'harris': ['sigma_grad', 'sigma_smooth', 'alpha', 'threshold']
[0.7, 2.0, 0.08, 1000]

'harris_binomial': ['mask_size_grd', 'mask_size_smooth', 'alpha', 'threshold', 'subpix']
[5, 15, 0.08, 1000, 'on']

Further details on the meaning of those parameters can be found in the descriptions of points_harris, points_harris_binomial and points_lepetit, respectively. If an empty tuple is passed or a parameter is not provided in DetectorParamName, the above mentioned default values are taken.

While adjusting the operator parameters it should be aimed that a set of 50 to 450 feature points is extracted (depending on the texture and size of Template), which are uniformly distributed over the template's ROI. Therefore, it is recommended to run the point operator of choice in advance and visualize the results by gen_cross_contour_xld. In most cases it is sufficient to use the default setting.

Descriptor parameters

A point descriptor is a classifier, which builds characteristic descriptions of the gray-value neighborhood for interest points. Currently the descriptor is implemented with the so-called *randomized ferns*, which learns the polarity of the gray-value differences of pairs of pixels certain at *random* locations in the surrounding area of the interest point. The descriptor is later used in find_uncalib_descriptor_model to classify interest points in the search image, or in other words: to identify (match) potential model points in the search image.

The descriptor needs to store only the projectively *stable* interest points (which will appear in many projective views of the template). To assess the *stability* of the interest points, a simulation is performed: Template undergoes many affine transformations and points that can be extracted in most of the views are considered *stable*. The affine transformations are a good approximation of the projective transformations within a local neighborhood of the interest points.

The following descriptor parameters can be set with DescriptorParamName and DescriptorParamValue:

Descriptor size parameters:

'depth': depth of a classification fern. A deeper randomized fern can better discriminate interest points. However, the memory requirement for the fern grows by a factor 2 to the power of *'depth'*.
Typical values are [*5 ... 11*], default value is *11*.

'number_ferns': number of fern structures used. Using more ferns increases the recognition robustness, however it also increases the runtime of the matching. If the memory needed for the descriptor should be small, many ferns with only a small depth should be used (e.g., *'number_ferns'=150*, *'depth'=5*). If detection speed is more important, fewer ferns of larger depth should be used (e.g., *'number_ferns'=10*, *'depth'=11*).
Typical values are [*1 ... 150*], default value is *30*.

'patch_size': side length of a quadratic neighborhood describing a point of interest. Too large values of this parameter can influence the run time.
Typical values are [*15 ... 33*], default value is *17*.

Summing up, the parameters *'depth'*, *'number_ferns'* and *'patch_size'* allow a transparent control over detection robustness, speed and memory consumption.

Simulation parameters:

'tilt': switches on or off the projective transformations during the simulation phase. When switched on, the robustness of the model is improved and objects with bigger tilts can be found. When switched off, training time can be significantly reduced and the model is still able to recognize projectively invariant objects.
Possible values are [*'on'*, *'off'*], default value is *'on'*.

'min_rot': minimal angle of rotation around the normal vector of Template.
Typical values are [*-180 ... 0*], default value is *-180*.

'max_rot': maximal angle of rotation around the normal vector of Template.
Typical values are [*0 ... 180*], default value is *180*.

'min_scale': minimal scale of Template.
Typical values are [*0.1 ... 1.0*], default value is *0.5*.

'max_scale': maximal scale of Template.
Typical values are [*1.0 ... 3.5*], default value is *1.4*.

The parameters *'min_rot'*, *'max_rot'*, *'min_scale'*, and *'max_scale'* allow to manually set which affine transformed views of the template are used to train the descriptor. Setting these parameters is useful to reduce training time especially in combination with the parameter *'tilt'*. Be aware that these parameters have direct influence on the results of find_uncalib_descriptor_model and therefore have to be set carefully. If, for example, the range of rotation is restricted from *'min_rot'* = -10 to *'max_rot'* = 10 it is not possible to find views of Template rotated by an angle outside that scope. A limited training range requires less ferns / ferns' depths to find Template. In such a case the number and depth of the ferns can be further reduced, which optimizes the model.

Remarks

Please note that depending on your hardware, the processing time for training the randomized ferns can vary between several seconds to some minutes. Therefore, a once trained model can be saved and loaded with write_descriptor_model and read_descriptor_model.

The parameters and the location of the final descriptor points can be determined with get_descriptor_model_params and get_descriptor_model_points.

create_uncalib_descriptor_model stores the detector type, detector parameters, and descriptor parameters, which are used in every succeeding call of find_uncalib_descriptor_model. The reference point (origin) of the model is the center of gravity of the template's ROI. Its coordinates can be changed by set_descriptor_model_origin.

Parameters

▷ **Template** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **DetectorType** (input_control) . . . . . string ⇝ *string*
The type of the detector.
Default: 'lepetit'
List of values: `DetectorType` ∈ {'lepetit', 'harris', 'harris_binomial'}

▷ **DetectorParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
The detector's parameter names.
Default: []
List of values: `DetectorParamName` ∈ {'alpha', 'check_neighbor', 'mask_size_grd', 'mask_size_smooth', 'min_check_neighbor_diff', 'min_score', 'radius', 'sigma_grad', 'sigma_smooth', 'subpix', 'threshold'}

▷ **DetectorParamValue** (input_control) . . . . . attribute.value-array ⇝ *integer* / real / string
Values of the detector's parameters.
Default: []
Suggested values: `DetectorParamValue` ∈ {0.08, 1, 1.2, 3, 15, 30, 1000, 'on', 'off', 'none', 'interpolation'}

▷ **DescriptorParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
The descriptor's parameter names.
Default: []
List of values: `DescriptorParamName` ∈ {'depth', 'max_rot', 'max_scale', 'min_rot', 'min_scale', 'number_ferns', 'patch_size', 'tilt'}

▷ **DescriptorParamValue** (input_control) . . . . . attribute.value-array ⇝ *integer* / real / string
Values of the descriptor's parameters.
Default: []
Suggested values: `DescriptorParamValue` ∈ {0.5, 1.4, 11, 21, 30, -180, 180, 'on', 'off'}

▷ **Seed** (input_control) . . . . . integer ⇝ *integer*
The seed for the random number generator.
Default: 42

▷ **ModelID** (output_control) . . . . . descriptor_model ⇝ *handle*
The handle to the descriptor model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`points_lepetit`, `points_harris`, `reduce_domain`

Possible Successors

`get_descriptor_model_params`, `find_uncalib_descriptor_model`

See also

`get_descriptor_model_params`, `find_uncalib_descriptor_model`

References

V. Lepetit and P. Fua: "Keypoint Recognition using Randomized Trees." IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 28, Nr. 9, pp. 1465-1479, 2006.

M. Ozuysal, P. Fua, and V. Lepetit: "Fast Keypoint Recognition in Ten Lines of Code." In Proceedings of Conference on Computer Vision and Pattern Recognition, 2007.

Module

Matching

deserialize_descriptor_model (
: : SerializedItemHandle : ModelID)

Deserialize a descriptor model.

deserialize_descriptor_model deserializes a descriptor model, that was serialized by serialize_descriptor_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized descriptor model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created descriptor model with the handle ModelID.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **ModelID** (output_control) ..... descriptor_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator deserialize_descriptor_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_descriptor_model

Possible Successors

find_uncalib_descriptor_model, find_calib_descriptor_model

See also

create_calib_descriptor_model, create_uncalib_descriptor_model

Module

Matching

find_calib_descriptor_model (Image : : ModelID,
DetectorParamName, DetectorParamValue, DescriptorParamName,
DescriptorParamValue, MinScore, NumMatches, CamParam,
ScoreType : Pose, Score)

Find the best matches of a calibrated descriptor model in an image and return their 3D pose.

The operator find_calib_descriptor_model finds the best NumMatches matches of the descriptor model ModelID in Image. The descriptor model must have been created previously by calling create_calib_descriptor_model or read_descriptor_model.

The operator returns the 3D Pose (in camera coordinate system) of the template object found in the search Image. As two different cameras can be used for the model generation and the online search, the camera parameters of the camera used to create Image have to be set in CamParam. Therefore, there is no need to correct any distortions in Image manually.

The detection process is the same as for find_uncalib_descriptor_model. The setting of DetectorParamName, DetectorParamValue, DescriptorParamName, DescriptorParamValue, MinScore, NumMatches and ScoreType are explained with find_uncalib_descriptor_model. After the object was detected, and using the world (from reference plane) coordinates of the matched model points (explained with create_calib_descriptor_model), the pose of the object is computed similarly to applying vector_to_pose operator.

The point correspondences for each object can be queried with get_descriptor_model_points.

Attention

Note that the domain of the search image should contain the whole object to be searched for because interest points are only extracted inside the domain of the search image. This means that if the domain does not contain the full object to be searched for, the resulting `Score` will decrease. Note also that matches may be found even if the reference point (origin) of the model lies outside of the domain of the search image. Both is in contrast to shape-based matching, where the domain of the search image defines the search space for the reference point of the model.

Parameters

- ▷ **`Image`** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image where the model should be found.
- ▷ **`ModelID`** (input_control) . . . . . descriptor_model ⇝ *handle*
  The handle to the descriptor model.
- ▷ **`DetectorParamName`** (input_control) . . . . . attribute.name-array ⇝ *string*
  The detector's parameter names.
  Default: []
  List of values: `DetectorParamName` ∈ {'alpha', 'check_neighbor', 'mask_size_grd', 'mask_size_smooth', 'min_check_neighbor_diff', 'min_score', 'radius', 'sigma_grad', 'sigma_smooth', 'subpix', 'threshold'}
- ▷ **`DetectorParamValue`** (input_control) . . . . . attribute.value-array ⇝ *integer* / real / string
  Values of the detector's parameters.
  Default: []
  Suggested values: `DetectorParamValue` ∈ {0.08, 1, 1.2, 3, 15, 30, 1000, 'on', 'off'}
- ▷ **`DescriptorParamName`** (input_control) . . . . . attribute.name-array ⇝ *string*
  The descriptor's parameter names.
  Default: []
  List of values: `DescriptorParamName` ∈ {'min_score_descr', 'guided_matching'}
- ▷ **`DescriptorParamValue`** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
  Values of the descriptor's parameters.
  Default: []
  Suggested values: `DescriptorParamValue` ∈ {0.0, 0.001, 0.005, 0.01, 'on', 'off'}
- ▷ **`MinScore`** (input_control) . . . . . real(-array) ⇝ *real*
  Minimum score of the instances of the models to be found.
  Default: 0.2
  Suggested values: `MinScore` ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
  Value range: 0 ≤ `MinScore` ≤ 1
- ▷ **`NumMatches`** (input_control) . . . . . integer ⇝ *integer*
  Maximal number of found instances.
  Default: 1
  Suggested values: `NumMatches` ∈ {1, 2, 3, 4}
  Restriction: NumMatches >= 1
- ▷ **`CamParam`** (input_control) . . . . . campar ⇝ *real* / integer / string
  Camera parameter (inner orientation) obtained from camera calibration.
- ▷ **`ScoreType`** (input_control) . . . . . string(-array) ⇝ *string*
  Score type to be evaluated in `Score`.
  Default: 'num_points'
  List of values: `ScoreType` ∈ {'num_points', 'inlier_ratio'}
- ▷ **`Pose`** (output_control) . . . . . pose(-array) ⇝ *real* / integer
  3D pose of the object.
- ▷ **`Score`** (output_control) . . . . . number(-array) ⇝ *real* / integer
  Score of the found instances according to the ScoreType input.

Example

```
*  Perform camera calibration (with standard calibration plate)
calibrate_cameras (CalibDataID, Error)
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
*  World coordinate system is defined by calibration plate in first image
```

```
get_calib_data (CalibDataID, 'calib_obj_pose',  [0,0], 'pose', \
                ObjInCameraPose)
*  Compensate thickness of plate
set_origin_pose(ObjInCameraPose, 0, 0, 0.0006, WorldPose)

* Create descriptor model:
create_calib_descriptor_model (Image,CamParam,WorldPose, \
                               'harris',[],[],[],[],42,ModelID)

get_descriptor_model_params (ModelID,DetectorType, \
                             DetectorParamName,DetectorParamValue, \
                             DescriptorParamName,DescriptorParamValue)
write_descriptor_model (ModelID,'simple_example.dsm')

read_descriptor_model ('simple_example.dsm',ModelID)
find_calib_descriptor_model (SearchImage,ModelID,[],[],[],[],0.2,1, \
                             CamParam,['num_points','inlier_ratio'], \
                             Pose,Score)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_calib_descriptor_model, read_descriptor_model

See also

vector_to_pose, points_lepetit, create_calib_descriptor_model

Module

Matching

find_uncalib_descriptor_model (Image : : ModelID, DetectorParamName, DetectorParamValue, DescriptorParamName, DescriptorParamValue, MinScore, NumMatches, ScoreType : HomMat2D, Score)

Find the best matches of a descriptor model in an image.

The operator find_uncalib_descriptor_model finds the best matches of a descriptor model ModelID in Image. The descriptor model must have been created previously by calling create_uncalib_descriptor_model, create_calib_descriptor_model or read_descriptor_model.

A match is only accepted if its score exceeds the value of MinScore. This criterion is based on the 'inlier_ratio' score that is described in details below. The main result of the operator find_uncalib_descriptor_model for each match is a 3x3 matrix HomMat2D, which describes a 2D projection of model points to search image points and is represented by a tuple of 9 elements in row-major order. If more matches of the searched object (template) appear and pass the MinScore criterion, the resulting multiple homographies are concatenated. The number of objects actually found is then equal to NumObjects = |HomMat2D|/9.

The detection process is divided into three parts. **First**, interest points are extracted from the search image (only inside the domain of the search image). This is done using the point detector and its parameters selected once during the generation of the model. However, DetectorParamName and DetectorParamValue can be used to specify different detector parameter values during the find_uncalib_descriptor_model call. By changing these parameters, it is possible to adjust to illumination changes between the model generation and the online detection. However, it is recommended to use the same values used to create the model (pass an empty tuple).

The second step of the detection process is to calculate correspondences between the model points and the points that were detected. The run time parameters of the descriptor can be adjusted with `DescriptorParamName` and `DescriptorParamValue`:

'min_score_descr': is the minimal classifier score for an interest point to be regarded as a potential match. The score function is between 0.0 and 1.0, but typically only values between 0.0 and 0.1 make sense. Increasing *'min_score_descr'* can increase significantly the detection speed. Note, however, that using *'min_score_descr'* might have negative effect on the robustness of the detection process, especially when only few points can be found.
Typical values are [*0.0* ... *0.1*], default value is *0.0*.

'guided_matching': enhances the accuracy of the object recognition when switched on. Note that it increases the computational costs up to 10% in some cases.
Possible values are [*'on'*, *'off'*], default value is *'on'*.

The last step is the estimation of a homography that describes the point correspondences. The homography is a 2D projection, which describes a transformation from model points to points in `Image`. Here, Natural 3D Markers (N3Ms) are utilized to identify robustly the point correspondences (see references).

Additionally to the estimated homography in `HomMat2D`, the operator returns one or more `Score` estimations per object instance as specified by the user in a tuple `ScoreType`. Currently the following values for `ScoreType` are supported:

'num_points': number of point correspondences per instance. An object instance should be considered good, if it has 10 or more point correspondences with the model. Fewer points are insufficient, because any random 4 point correspondences define a mathematically correct homography between two images.

'inlier_ratio': the ratio of the number of point correspondences to the number of model points. Although this value can have values of [*0.0* ... *1.0*], it is rather unlikely that this ratio can reach *1.0*. Yet, objects having inlier ratio less than *0.1*, should be disregarded.

Note that the resulting scores for more than one object instance will be concatenated in `Score`, such that |`Score`| = NumObjects*|`ScoreType`|.

The point correspondences for each object can be queried with `get_descriptor_model_points`.

Attention

Note that the domain of the search image should contain the whole object to be searched for because interest points are only extracted inside the domain of the search image. This means that if the domain does not contain the full object to be searched for, the resulting `Score` will decrease. Note also that matches may be found even if the reference point (origin) of the model lies outside of the domain of the search image. Both is in contrast to shape-based matching, where the domain of the search image defines the search space for the reference point of the model.

Parameters

▷ **`Image`** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
Input image where the model should be found.

▷ **`ModelID`** (input_control) ..... descriptor_model ⇝ *handle*
The handle to the descriptor model.

▷ **`DetectorParamName`** (input_control) ..... attribute.name-array ⇝ *string*
The detector's parameter names.
Default: []
List of values: `DetectorParamName` ∈ {'alpha', 'check_neighbor', 'mask_size_grd', 'mask_size_smooth', 'min_check_neighbor_diff', 'min_score', 'radius', 'sigma_grad', 'sigma_smooth', 'subpix', 'threshold'}

▷ **`DetectorParamValue`** (input_control) ..... attribute.value-array ⇝ *integer* / real / string
Values of the detector's parameters.
Default: []
Suggested values: `DetectorParamValue` ∈ {0.08, 1, 1.2, 3, 15, 30, 1000, 'on', 'off'}

▷ **`DescriptorParamName`** (input_control) ..... attribute.name-array ⇝ *string*
The descriptor's parameter names.
Default: []
List of values: `DescriptorParamName` ∈ {'min_score_descr', 'guided_matching'}

▷ **DescriptorParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Values of the descriptor's parameters.
Default: []
Suggested values: DescriptorParamValue ∈ {0.0, 0.001, 0.005, 0.01, 'on', 'off'}

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the models to be found.
Default: 0.2
Suggested values: MinScore ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1

▷ **NumMatches** (input_control) . . . . . integer ⇝ *integer*
Maximal number of found instances.
Default: 1
Suggested values: NumMatches ∈ {1, 2, 3, 4}
Restriction: NumMatches >= 1

▷ **ScoreType** (input_control) . . . . . string(-array) ⇝ *string*
Score type to be evaluated in Score.
Default: 'num_points'
List of values: ScoreType ∈ {'num_points', 'inlier_ratio'}

▷ **HomMat2D** (output_control) . . . . . hom_mat2d(-array) ⇝ *real*
Homography between model and found instance.

▷ **Score** (output_control) . . . . . number(-array) ⇝ *real* / integer
Score of the found instances according to the ScoreType input.

Example

```
create_uncalib_descriptor_model (ImageReduced,'harris',[],[], \
                                 [],[],42,ModelID)

get_descriptor_model_params (ModelID,DetectorType, \
                             DetectorParamName,DetectorParamValue, \
                             DescriptorParamName,DescriptorParamValue)
write_descriptor_model (ModelID,'simple_example.dsm')

read_descriptor_model ('simple_example.dsm',ModelID)
find_uncalib_descriptor_model (SearchImage,ModelID,[],[],[],[],0.2,1, \
                               ['num_points','inlier_ratio'],HomMat2D,Score)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_uncalib_descriptor_model, create_calib_descriptor_model, read_descriptor_model

See also

create_uncalib_descriptor_model, create_calib_descriptor_model, find_calib_descriptor_model, get_descriptor_model_points

References

S. Hinterstoisser, S. Benhimane, and N. Navab: "N3M: Natural 3D Markers for Real-Time Object Detection and Pose Estimation." IEEE 11th International Conference on Computer Vision, 2007. pp. 1-7, ICCV 2007.

Module

Matching

get_descriptor_model_origin (: : ModelID : Row, Column)

Return the origin of a descriptor model.

The operator get_descriptor_model_origin returns the origin (reference point) of the descriptor model ModelID. The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the descriptor model with create_uncalib_descriptor_model, or create_calib_descriptor_model. Hence, an origin of (0,0) means that the center of gravity of the domain of the model image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity.

Parameters

- ▷ **ModelID** (input_control) . . . . . descriptor_model ⇝ *handle*
  Handle of a descriptor model.
- ▷ **Row** (output_control) . . . . . real ⇝ *real* / integer
  Position of origin in row direction.
- ▷ **Column** (output_control) . . . . . real ⇝ *real* / integer
  Position of origin in column direction.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_uncalib_descriptor_model, read_descriptor_model

See also

set_descriptor_model_origin, create_uncalib_descriptor_model

Module

Matching

get_descriptor_model_params (: : ModelID : DetectorType, DetectorParamName, DetectorParamValue, DescriptorParamName, DescriptorParamValue)

Return the parameters of a descriptor model.

The operator get_descriptor_model_params allows to access the parameters of a descriptor model that was created with create_uncalib_descriptor_model or create_calib_descriptor_model. Therefore, a handle of a descriptor model must be passed in ModelID. The operator returns in DetectorType, DetectorParamName and DetectorParamValue the type and parametrization of the detector. In DescriptorType, DescriptorParamName and DescriptorParamValue the respective descriptor is returned.

An application for this operator is the possibility to query the parameters that where used in a stored model. As the operator needs a handle to a descriptor model, it has to be loaded first by using read_descriptor_model.

Parameters

- ▷ **ModelID** (input_control) . . . . . descriptor_model ⇝ *handle*
  The object handle to the descriptor model.
- ▷ **DetectorType** (output_control) . . . . . string ⇝ *string*
  The type of the detector.
- ▷ **DetectorParamName** (output_control) . . . . . attribute.name-array ⇝ *string*
  The detectors parameter names.
- ▷ **DetectorParamValue** (output_control) . . . . . attribute.value-array ⇝ *integer* / real / string
  Values of the detectors parameters.
- ▷ **DescriptorParamName** (output_control) . . . . . attribute.name-array ⇝ *string*
  The descriptors parameter names.

▷ **DescriptorParamValue** (output_control) . . . . . attribute.value-array ⇝ *integer* / real / string
Values of the descriptors parameters.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_uncalib_descriptor_model`, `create_calib_descriptor_model`, `read_descriptor_model`

See also

`create_uncalib_descriptor_model`, `create_calib_descriptor_model`

Module

Matching

get_descriptor_model_points (: : ModelID, Set, Subset : Row, Column)

Query the interest points of the descriptor model or the last processed search image.

With the operator `get_descriptor_model_points` interest points of the descriptor model or the last processed search image can be queried. It requires a `ModelID` returned by `create_uncalib_descriptor_model`, `create_calib_descriptor_model`, or `read_descriptor_model`.

The interest points stored in the model can always be queried by setting `Set` to *'model'* and `Subset` to *'all'*. If a `find_uncalib_descriptor_model` or `find_calib_descriptor_model` precedes, with `get_descriptor_model_points` the interest points of the last search image can be queried as well by setting `Set` to *'search'* and `Subset` to *'all'*. Additionally, the matched (corresponding) points for each object instance found can be queried by setting `Set` to *'model'* or *'search'* (for the correspondences on the model or search image side, respectively) and `Subset` to the result number of the instance. The image coordinates of the queried points are returned in `Row` and `Column`.

Parameters

▷ **ModelID** (input_control) . . . . . descriptor_model ⇝ *handle*
The handle to the descriptor model.

▷ **Set** (input_control) . . . . . string ⇝ *string*
Set of interest points.
Default: 'model'
List of values: `Set` ∈ {'model', 'search'}

▷ **Subset** (input_control) . . . . . integer ⇝ *integer* / string
Subset of interest points.
Default: 'all'
Suggested values: `Subset` ∈ {'all', 0, 1, 2}

▷ **Row** (output_control) . . . . . point.y-array ⇝ *real* / integer
Row coordinates of interest points.

▷ **Column** (output_control) . . . . . point.x-array ⇝ *real* / integer
Column coordinates of interest points.

Example

```
create_uncalib_descriptor_model (Template,'harris',[],[],[],[],42, \
                                 ModelID)
* Model points can be queried from a model, even if just created
get_descriptor_model_points (ModelID,'model','all',ModelRow,ModelColumn)
find_uncalib_descriptor_model (Image,ModelID,[],[],[],[],0.2,1, \
```

```
                                'num_points',HomMat2D,Score)
* Search points can be queried only after a
* find_[un]calib_descriptor_model was executed
get_descriptor_model_points (ModelID,'search','all',SearchRow,SearchColumn)
* Additionally, correspondences for the results can be queried
NumObjects := |HomMat2D|/9
for I := 0 to NumObjects-1 by 1
    * Query corresponding points in the model
    get_descriptor_model_points (ModelID,'model',I, \
                                 CorrModelRow,CorrModelColumn)
    * Query corresponding points in the search image
    get_descriptor_model_points (ModelID,'search',I, \
                                 CorrSearchRow,CorrSearchColumn)
    * Those points are typically for visualizational purposes
    gen_cross_contour_xld (CrossModel,CorrModelRow,CorrModelColumn, \
                           6,0.78)
    gen_cross_contour_xld (CrossSearch,CorrSearchRow,CorrSearchColumn, \
                           6,0.78)

    * ....

endfor
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_uncalib_descriptor_model, create_calib_descriptor_model, find_uncalib_descriptor_model, find_calib_descriptor_model, read_descriptor_model

See also

create_uncalib_descriptor_model, create_calib_descriptor_model

Module

Matching

get_descriptor_model_results (: : ModelID, ObjectID, ResultNames : Results)

Query alphanumerical results that were accumulated during descriptor-based matching.

The operator get_descriptor_model_results allows to access alphanumeric results of the descriptor based matching process. The results are stored in a descriptor based model specified by ModelID. ObjectID specifies for which object the results should be returned. The objects are 0-based indexed and in the same order as they are detected by find_uncalib_descriptor_model or find_calib_descriptor_model, respectively. Results can be queried for all objects at once by specifying *'all'* in ObjectID, if they are single valued (e.g., ResultNames=*'num_points'* or ResultNames=*'inlier_ratio'*).

The results returned in Results can be selected by setting ResultNames to:

'num_points': number of object points that correspond to model points.

'inlier_ratio': ratio of successfully matched object points relative to all descriptor model points.

'homography': a 3x3 projective transformation matrix which transforms model points into object points. Note that the result of the mapping depends on the selected model origin (see set_descriptor_model_origin).

'pose': returns the estimated object pose. Note that this result can only be queried after `find_calib_descriptor_model` was called.

'point_classification': returns concatenated triads representing the classification results for the interest points extracted from the search image during the last call of `find_uncalib_descriptor_model` or `find_calib_descriptor_model`. Each triad consists of a *search point index* on the first, a *model point index* on the second, and a *classification score* on the third position. The search and model point indices correspond to the point coordinates returned by `get_descriptor_model_points` with the parameter `Subset` set to *'all'* and the parameter `Set` set to *'search'* and *'model'*, respectively. If a number is passed in `ObjectID` only the classification results for the points matched to this object are returned. If *'all'* is passed in `ObjectID` the classification results for all points, including the points not matched to any object, are returned. The triads are sorted in descending order with respect to their score. Points having a score less than the value of the descriptor parameter *'min_score_descr'* (see `find_calib_descriptor_model` or `find_uncalib_descriptor_model`) are rejected and are not listed in the resulting classification results.

Parameters

▷ **`ModelID`** (input_control) . . . . . descriptor_model ⇝ *handle*
Handle of a descriptor model.

▷ **`ObjectID`** (input_control) . . . . . integer ⇝ *integer* / string
Handle of the object for which the results are queried.
Default: 'all'
Suggested values: `ObjectID` ∈ {'all', 0, 1, 2, 3}

▷ **`ResultNames`** (input_control) . . . . . attribute.name ⇝ *string*
Name of the results to be queried.
Default: 'num_points'
List of values: `ResultNames` ∈ {'num_points', 'inlier_ratio', 'homography', 'pose', 'point_classification'}

▷ **`Results`** (output_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Returned results.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`find_uncalib_descriptor_model`, `find_calib_descriptor_model`

See also

`get_descriptor_model_origin`, `get_descriptor_model_params`, `get_descriptor_model_points`

Module

Matching

`read_descriptor_model` (: : FileName : ModelID)

Read a descriptor model from a file.

The operator `read_descriptor_model` reads a descriptor model, which has been written with `write_descriptor_model`, from the file `FileName`. The default HALCON file extension for the descriptor model is 'dsm'.

Parameters

▷ **`FileName`** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: `.dsm`

▷ **ModelID** (output_control) . . . . . descriptor_model ⇝ *handle*
Handle of the model.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

find_uncalib_descriptor_model, find_calib_descriptor_model

See also

create_calib_descriptor_model, create_uncalib_descriptor_model

Module

Matching

serialize_descriptor_model (: : ModelID : SerializedItemHandle)

Serialize a descriptor model.

serialize_descriptor_model serializes the data of a descriptor model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_descriptor_model is converted to a serialized item. The descriptor model is defined by the handle ModelID. The serialized descriptor model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_descriptor_model.

Parameters

▷ **ModelID** (input_control) . . . . . descriptor_model ⇝ *handle*
Handle of a model to be saved.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_descriptor_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_lepetit, create_uncalib_descriptor_model, create_calib_descriptor_model

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_descriptor_model

See also

create_uncalib_descriptor_model, create_calib_descriptor_model

Module

Matching

set_descriptor_model_origin (: : ModelID, Row, Column :)

Sets the origin of a descriptor model.

The operator set_descriptor_model_origin sets the origin (reference point) of the descriptor model ModelID to a new value. The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the descriptor model with create_uncalib_descriptor_model, or create_calib_descriptor_model. Hence, an origin of (0,0) means that the center of gravity of the domain of the model image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity. The new setting affects the succeeding find_uncalib_descriptor_model and find_calib_descriptor_model calls.

Parameters

▷ **ModelID** (input_control) ..... descriptor_model ⇝ *handle*
Handle of a descriptor model.

▷ **Row** (input_control) ..... real ⇝ *real* / integer
Translation of origin in row direction.
Default: 0

▷ **Column** (input_control) ..... real ⇝ *real* / integer
Translation of origin in column direction.
Default: 0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_uncalib_descriptor_model, create_calib_descriptor_model, read_descriptor_model

See also

get_descriptor_model_origin, create_uncalib_descriptor_model

Module

Matching

write_descriptor_model (: : ModelID, FileName :)

Write a descriptor model to a file.

The operator write_descriptor_model writes a descriptor model to the file FileName. The model can be read again with read_descriptor_model. The default HALCON file extension for the descriptor model is 'dsm'.

Parameters

▷ **ModelID** (input_control) ..... descriptor_model ⇝ *handle*
Handle of a model to be saved.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
The path and filename of the model to be saved.
File extension: .dsm

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_lepetit, create_uncalib_descriptor_model, create_calib_descriptor_model

See also

create_uncalib_descriptor_model, create_calib_descriptor_model

Module

Matching

18.5 Shape-Based

adapt_shape_model_high_noise (ImageReduced : : ModelID, GenParam : ResultDict)

Adapt a shape model's parameters for high noise images.

adapt_shape_model_high_noise adapts the parameters of a shape model ModelID for high noise images and returns them in ResultDict. Based on a sample search image ImageReduced, the optimal values of the shape model's parameters are estimated and adapted in the model. The domain of ImageReduced should contain the model and surrounding background, such that the typical noise is presented to the algorithm. With GenParam the parameters to be estimated can be controlled. Currently, the operator supports the estimation of the lowest pyramid level used in the matching step. Accordingly, the value *'all'* or *'lowest_level'* should be provided to GenParam.

Attention: The operator changes the shape model ModelID by setting the estimated lowest pyramid level.

Parameters

▷ **ImageReduced** (input_object) ..... (multichannel-)image ⇝ *object :* byte / uint2
Sample search image with reduced domain.

▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
Handle of the shape model.

▷ **GenParam** (input_control) ..... string-array ⇝ *string*
Parameters to be estimated.
Default: 'all'
List of values: GenParam ∈ {'lowest_level', 'all'}

▷ **ResultDict** (output_control) ..... dict ⇝ *handle*
Dictionary with the estimated parameter values.

Result

If the parameters are valid, the operator adapt_shape_model_high_noise returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_generic_shape_model`

Possible Successors

`find_generic_shape_model`

Module

Matching

clear_shape_model (: : ModelID :)

Free the memory of a shape model.

The operator `clear_shape_model` frees the memory of a shape model that was created by `create_shape_model`, `create_scaled_shape_model`, or `create_aniso_shape_model`. After calling `clear_shape_model`, the model can no longer be used. The handle `ModelID` becomes invalid.

Parameters

▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
Handle of the model.

Result

If the handle of the model is valid, the operator `clear_shape_model` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_generic_shape_model`, `write_shape_model`

Module

Matching

create_aniso_shape_model (Template : : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleRMin, ScaleRMax, ScaleRStep, ScaleCMin, ScaleCMax, ScaleCStep, Optimization, Metric, Contrast, MinContrast : ModelID)

Prepare an anisotropically scaled shape model for matching.

The operator `create_aniso_shape_model` prepares a template, which is passed in the image `Template`, as an anisotropically scaled shape model used for matching. The ROI of the model is passed as the domain of `Template`.

The output parameter `ModelID` is a handle for this model, which is used in subsequent calls to `find_aniso_shape_model`. The center of gravity of the domain (region) of the model image `Template` is used as the origin (reference point) of the model. A different origin can be set with `set_shape_model_origin`. The model is generated using multiple image pyramid levels and is stored in

memory. If a complete pregeneration of the model is selected (see below), the model is generated at multiple rotations and anisotropic scales (i.e., independent scales in the row and column direction) on each level. The model can be extended by clutter parameters with set_shape_model_clutter.

Input parameters in detail

NumLevels: The number of pyramid levels is determined with the parameter NumLevels. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, NumLevels must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. This can be checked using the output of inspect_shape_model. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, create_aniso_shape_model returns with an error message.

If NumLevels is set to *'auto'* (or *0* for backwards compatibility), create_aniso_shape_model determines the number of pyramid levels automatically. The automatically computed number of pyramid levels can be queried using get_shape_model_params. In rare cases, it might happen that create_aniso_shape_model determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for MinScore or Greediness in find_aniso_shape_model in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in find_aniso_shape_model may increase. In these cases, the number of pyramid levels should be selected using the output of inspect_shape_model.

AngleStart, AngleExtent, and AngleStep: The parameters AngleStart and AngleExtent determine the range of possible rotations, in which the model can occur in the image. Note that the model can only be found in this range of angles by find_aniso_shape_model. The parameter AngleStep determines the step length within the selected range of angles. Hence, if subpixel accuracy is not specified in find_aniso_shape_model, this parameter specifies the accuracy that is achievable for the angles in find_aniso_shape_model. AngleStep should be chosen based on the size of the object. Smaller models do not have many different discrete rotations in the image, and hence AngleStep should be chosen larger for smaller models. If AngleExtent is not an integer multiple of AngleStep, AngleStep is modified accordingly.

To ensure that for model instances without rotation angle values of exactly 0.0 are returned by find_aniso_shape_model, the range of possible rotations is modified as follows: If there is no positive integer value n such that AngleStart plus n times AngleStep is exactly 0.0, AngleStart is decreased by up to AngleStep and AngleExtent is increased by AngleStep.

ScaleRMin, ScaleRMax, ScaleCMin, ScaleCMax, ScaleRStep, and ScaleCStep: The parameters ScaleRMin, ScaleRMax, ScaleCMin, and ScaleCMax determine the range of possible anisotropic scales of the model in the row and column direction. A scale of 1 in both scale factors corresponds to the original size of the model. The parameters ScaleRStep and ScaleCStep determine the step length within the selected range of scales. Hence, if subpixel accuracy is not specified in find_aniso_shape_model, these parameters specify the accuracy that is achievable for the scales in find_aniso_shape_model. Like AngleStep, ScaleRStep and ScaleCStep should be chosen based on the size of the object. If the respective range of scales is not an integer multiple of ScaleRStep and ScaleCStep, ScaleRStep and ScaleCStep are modified accordingly.

To ensure that for model instances that are not scaled scale values of exactly 1.0 are returned by find_aniso_shape_model, the range of possible scales is modified as follows: If there are no positive integer values n and m such that ScaleRMin plus n times ScaleRStep is exactly 1.0 and ScaleCMin plus m times ScaleCStep is exactly 1.0, ScaleRMin and ScaleCMin are decreased by up to ScaleRStep and ScaleCStep, respectively, and ScaleRMax and ScaleCMax are increased such that the range of possible scales is increased by ScaleRStep and ScaleCStep, respectively.

Note that the transformations are treated internally such that the scalings are applied first, followed by the rotation. Therefore, the model should usually be aligned such that it appears horizontally or vertically in the model image.

Optimization: For particularly large models, it may be useful to reduce the number of model points by setting Optimization to a value different from *'none'*. If Optimization = *'none'*, all model points are stored. In all other cases, the number of points is reduced according to the value of Optimization. If the number of points is reduced, it may be necessary in find_aniso_shape_model to set the parameter Greediness to a smaller value, e.g., *0.7* or *0.8*. For small models, the reduction of the number of model

points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined.

If Optimization is set to *'auto'*, create_aniso_shape_model automatically determines the reduction of the number of model points.

Metric: The parameter Metric determines the conditions under which the model is recognized in the image.

If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background.

If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_aniso_shape_model will increase slightly in this case.

If Metric = *'ignore_local_polarity'*, the model is found even if the contrast changes locally. This mode can, for example, be useful if the object consists of a part with medium gray value, within which either darker or brighter sub-objects lie. Since in this case the runtime of find_aniso_shape_model increases significantly, it is usually better to create several models that reflect the possible contrast variations of the object with create_aniso_shape_model, and to match them simultaneously with find_aniso_shape_models.

The above three metrics can only be applied to single-channel images. If a multichannel image is used as the model image or as the search image, only the first channel will be used (and no error message will be returned).

If Metric = *'ignore_color_polarity'*, the model is found even if the color contrast changes locally. This is, for example, the case if parts of the object can change their color, e.g., from red to green. In particular, this mode is useful if it is not known in advance in which channels the object is visible. In this mode, the runtime of find_aniso_shape_model can also increase significantly. The metric *'ignore_color_polarity'* can be used for images with an arbitrary number of channels. If it is used for single-channel images it has the same effect as *'ignore_local_polarity'*. It should be noted that for Metric = *'ignore_color_polarity'* the number of channels in the model creation with create_aniso_shape_model and in the search with find_aniso_shape_model can be different. This can, for example, be used to create a model from a synthetically generated single-channel image. Furthermore, it should be noted that the channels do not need to contain a spectral subdivision of the light (like in an RGB image). The channels can, for example, also contain images of the same object that were obtained by illuminating the object from different directions.

Contrast: The parameter Contrast determines the contrast the model points must have. The contrast is a measure for local gray value differences between the object and the background and between different parts of the object. Contrast should be chosen such that only the significant features of the template are used for the model. Contrast can also contain a tuple with two values. In this case, the model is segmented using a method similar to the hysteresis threshold method used in edges_image. Here, the first element of the tuple determines the lower threshold, while the second element determines the upper threshold. For more information about the hysteresis threshold method, see hysteresis_threshold. Optionally, Contrast can contain a third value as the last element of the tuple. This value determines a threshold for the selection of significant model components based on the size of the components, i.e., components that have fewer points than the minimum size thus specified are suppressed. As the minimum size is applied on the extent of the components, the derived model contours can still be smaller than the specified minimum size. This threshold for the minimum size is divided by two for each successive pyramid level. If small model components should be suppressed, but hysteresis thresholding should not be performed, nevertheless three values must be specified in Contrast. In this case, the first two values can simply be set to identical values. The effect of this parameter can be checked in advance with inspect_shape_model.

If Contrast is set to *'auto'*, create_aniso_shape_model determines the three above described values automatically. Besides, only the contrast (*'auto_contrast'*), the hysteresis thresholds (*'auto_contrast_hyst'*), or the minimum size (*'auto_min_size'*) can be determined automatically. The remaining values that are not determined automatically can additionally be passed in the form of a tuple. Also various combinations are allowed: If, for example, *['auto_contrast','auto_min_size']* is passed, both the contrast and the minimum size are determined automatically. If *['auto_min_size',20,30]* is passed, the minimum size is determined automatically while the hysteresis thresholds are set to *20* and *30*, etc. In certain cases, it might happen that the automatic determination of the contrast thresholds is not satisfying. For example, a manual setting of these parameters should be preferred if certain model components should be included or suppressed because of application-specific reasons or if the object contains several different contrasts. Therefore, the contrast thresholds should be automatically determined with determine_shape_model_params and subsequently verified using inspect_shape_model before calling create_aniso_shape_model.

Note that MinContrast influences the automatic contrast estimation, and hence also the estimation of the minimum size.

MinContrast: With MinContrast, it can be determined which contrast the model must at least have in the recognition performed by find_aniso_shape_model. In other words, this parameter separates the model from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, MinContrast should be set to 10. If multichannel images are used for the model and the search images, and if the parameter Metric is set to *'ignore_color_polarity'* (see above) the noise in one channel must be multiplied by the square root of the number of channels to determine MinContrast. If, for example, the gray values fluctuate within a range of 10 gray levels in a single channel and the image is a three-channel image MinContrast should be set to 17. Obviously, MinContrast must be smaller than Contrast. If the model should be recognized in very low contrast images, MinContrast must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, MinContrast should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the position and rotation of the model are extracted robustly and accurately by find_aniso_shape_model.

If MinContrast is set to *'auto'*, the minimum contrast is determined automatically based on the noise in the model image. Consequently, an automatic determination only makes sense if the image noise during the recognition is similar to the noise in the model image. Furthermore, in some cases it is advisable to increase the automatically determined value in order to increase the robustness against occlusions (see above). The automatically computed minimum contrast can be queried using get_shape_model_params.

Complete pregeneration of the model

Optionally, a second value can be passed in Optimization. This value determines whether the model is pregenerated completely or not. To do so, the second value of Optimization must be set to either *'pregeneration'* or *'no_pregeneration'*. If the second value is not used (i.e., if only one value is passed), the mode that is set with set_system('pregenerate_shape_models',...) is used. With the default value (*'pregenerate_shape_models'* = *'false'*), the model is not pregenerated completely. The complete pregeneration of the model normally leads to slightly lower runtimes because the model does not need to be transformed at runtime. However, in this case, the memory requirements and the time required to create the model are significantly higher. It should also be noted that it cannot be expected that the two modes return exactly identical results because transforming the model at runtime necessarily leads to different internal data for the transformed models than pregenerating the transformed models. For example, if the model is not pregenerated completely, find_aniso_shape_model typically returns slightly lower scores, which may require setting a slightly lower value for MinScore than for a completely pregenerated model. Furthermore, the poses obtained by interpolation may differ slightly in the two modes. If maximum accuracy is desired, the pose of the model should be determined by least-squares adjustment.

If a complete pregeneration of the model is selected, the model is pregenerated for the selected angle and scale range and stored in memory. The memory required to store the model is proportional to the number of angle steps, the number of scale steps, and the number of points in the model. Hence, if AngleStep, ScaleRStep, or ScaleCStep are too small or AngleExtent or the range of scales are too big, it may happen that the model no longer fits into the (virtual) memory. In this case, AngleStep, ScaleRStep, or ScaleCStep must be enlarged or AngleExtent or the range of scales must be reduced. In any case, it is desirable that the model completely fits into the main memory, because this avoids paging by the operating system, and hence the time to find the object will be much smaller. Since angles can be determined with subpixel resolution by find_aniso_shape_model, AngleStep $\geq 1^\circ$ and ScaleRStep, ScaleCStep ≥ 0.02 can be selected for models of a diameter smaller than about 200 pixels.

If AngleStep $= 'auto'$ or ScaleRStep, ScaleCStep $= 'auto'$ (or *0* for backwards compatibility in both cases) is selected, create_aniso_shape_model automatically determines a suitable angle or scale step length, respectively, based on the size of the model. The automatically computed angle and scale step lengths can be queried using get_shape_model_params.

If a complete pregeneration of the model is not selected, the model is only created in a reference pose on each pyramid level. In this case, the model must be transformed to the different angles and scales at runtime in find_aniso_shape_model. Because of this, the recognition of the model might require slightly more time.

Note that pregenerated shape models are tailored to a specific image size. For runtime reasons using images of different sizes during the search with the same model in parallel is not supported. In this case, copies of the same model must be used, otherwise the program may crash!

Parameters

▷ **Template** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: NumLevels ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the pattern.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **AngleStep** (input_control) . . . . . angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: AngleStep ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep >= 0 && AngleStep <= pi / 2

▷ **ScaleRMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in the row direction.
Default: 0.9
Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the pattern in the row direction.
Default: 1.1
Suggested values: ScaleRMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleRMax >= ScaleRMin

▷ **ScaleRStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in the row direction.
Default: 'auto'
Suggested values: ScaleRStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleRStep >= 0

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in the column direction.
Default: 0.9
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the pattern in the column direction.
Default: 1.1
Suggested values: ScaleCMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleCMax >= ScaleCMin

▷ **ScaleCStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in the column direction.
Default: 'auto'
Suggested values: ScaleCStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleCStep >= 0

▷ **Optimization** (input_control) . . . . . string(-array) ⇝ *string*
Kind of optimization and optionally method used for generating the model.
Default: 'auto'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high', 'pregeneration', 'no_pregeneration'}

▷ **Metric** (input_control) ..... string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **Contrast** (input_control) ..... number(-array) ⇝ *integer* / string
Threshold or hysteresis thresholds for the contrast of the object in the template image and optionally minimum size of the object parts.
Default: 'auto'
Suggested values: Contrast ∈ {'auto', 'auto_contrast', 'auto_contrast_hyst', 'auto_min_size', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}

▷ **MinContrast** (input_control) ..... number ⇝ *integer* / string
Minimum contrast of the objects in the search images.
Default: 'auto'
Suggested values: MinContrast ∈ {'auto', 1, 2, 3, 5, 7, 10, 20, 30, 40}
Restriction: MinContrast < Contrast

▷ **ModelID** (output_control) ..... shape_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_aniso_shape_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameters NumLevels and Contrast are chosen such that the model contains too few points, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

draw_region, reduce_domain, threshold

Possible Successors

find_aniso_shape_model, find_aniso_shape_models, get_shape_model_params, clear_shape_model, write_shape_model, set_shape_model_origin, set_shape_model_clutter

Alternatives

create_generic_shape_model

See also

set_system, get_system

Module

Matching

create_aniso_shape_model_xld (Contours : : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleRMin, ScaleRMax, ScaleRStep, ScaleCMin, ScaleCMax, ScaleCStep, Optimization, Metric, MinContrast : ModelID)

Prepare an anisotropically scaled shape model for matching from XLD contours.

The operator create_aniso_shape_model_xld creates an anisotropically scaled shape model used for matching from the XLD contours passed in Contours. The XLD contours represent the gray value edges of the object to be searched for. In contrast to the operator create_aniso_shape_model, which creates a shape model from a template image, the operator create_aniso_shape_model_xld creates the shape model from XLD contours, i.e., without the use of a template image.

The output parameter ModelID is a handle for this model, which is used in subsequent calls to find_aniso_shape_model. The center of gravity of the smallest surrounding rectangle of the Contours that is parallel to the coordinate axes is used as the origin (reference point) of the model. A different origin can be set with set_shape_model_origin. The model is generated for multiple image pyramid levels and is stored in memory. If a complete pregeneration of the model is selected (see below), the model is generated at multiple rotations and anisotropic scales (i.e., independent scales in the row and column direction) on each level. The model can be extended by clutter parameters with set_shape_model_clutter.

Input parameters in detail

NumLevels: The number of pyramid levels is determined with the parameter NumLevels. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, NumLevels must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, create_aniso_shape_model_xld returns with an error message.

If NumLevels is set to *'auto'*, create_aniso_shape_model_xld determines the number of pyramid levels automatically. The computed number of pyramid levels can be queried using get_shape_model_params. In rare cases, it might happen that create_aniso_shape_model_xld determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for MinScore or Greediness in find_aniso_shape_model in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in find_aniso_shape_model may increase. In these cases, the number of pyramid levels should be selected manually.

AngleStart, AngleExtent, and AngleStep: The parameters AngleStart and AngleExtent determine the range of possible rotations, in which the object can occur in the image. Note that the object can only be found in this range of angles by find_aniso_shape_model. The parameter AngleStep determines the step length within the selected range of angles. Hence, if subpixel accuracy is not specified in find_aniso_shape_model, this parameter specifies the accuracy that is achievable for the angles in find_aniso_shape_model. AngleStep should be chosen based on the size of the object. Smaller models do not have many different discrete rotations in the image, and hence AngleStep should be chosen larger for smaller models. If AngleExtent is not an integer multiple of AngleStep, AngleStep is modified accordingly. To ensure that for model instances without rotation angle values of exactly 0.0 are returned by find_aniso_shape_model, the range of possible rotations is modified as follows: If there is no positive integer value n such that AngleStart plus n times AngleStep is exactly 0.0, AngleStart is decreased by up to AngleStep and AngleExtent is increased by AngleStep.

ScaleRMin, ScaleRMax, ScaleCMin, ScaleCMax, ScaleRStep, and ScaleCStep: The parameters ScaleRMin, ScaleRMax, ScaleCMin, and ScaleCMax determine the range of possible anisotropic scales of the object in the row and column direction. A scale of 1 in both scale factors corresponds to the original size of the model. The parameters ScaleRStep and ScaleCStep determine the step length within the selected range of scales. Hence, if subpixel accuracy is not specified in find_aniso_shape_model, these parameters specify the accuracy that is achievable for the scales in find_aniso_shape_model. Like AngleStep, ScaleRStep and ScaleCStep should be chosen based on the size of the object. If the respective range of scales is not an integer multiple of ScaleRStep and ScaleCStep, ScaleRStep and ScaleCStep are modified accordingly.

To ensure that for model instances that are not scaled scale values of exactly 1.0 are returned by find_aniso_shape_model, the range of possible scales is modified as follows: If there are no positive integer values n and m such that ScaleRMin plus n times ScaleRStep is exactly 1.0 and ScaleCMin plus m times ScaleCStep is exactly 1.0, ScaleRMin and ScaleCMin are decreased by up to ScaleRStep and ScaleCStep, respectively, and ScaleRMax and ScaleCMax are increased such that the range of possible scales is increased by ScaleRStep and ScaleCStep, respectively.

Note that the transformations are treated internally such that the scalings are applied first, followed by the rotation. Therefore, the model should usually be aligned such that it appears horizontally or vertically in the model image.

Optimization: For particularly large models, it may be useful to reduce the number of model points by setting Optimization to a value different from *'none'*. If Optimization = *'none'*, all model points are stored. In all other cases, the number of points is reduced according to the value of Optimization. If

the number of points is reduced, it may be necessary in find_aniso_shape_model to set the parameter Greediness to a smaller value, e.g., *0.7* or *0.8*. For small models, the reduction of the number of model points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined.

If Optimization is set to *'auto'*, create_aniso_shape_model_xld automatically determines the reduction of the number of model points.

Metric: The parameter Metric determines the conditions under which the model is recognized in the image.

If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background.

If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_aniso_shape_model will increase slightly in this case.

Note that the metrics (*'use_polarity'* and *'ignore_global_polarity'*) can only be selected if all Contours provide the attribute *'edge_direction'*, which defines the polarity of the edges. This attribute is available for contours created, e.g., with edges_sub_pix with the parameter Method set to, e.g., *'canny'*. Otherwise, these two metrics can be selected with the operator set_shape_model_metric, which determines the polarity of the edges from an image.

If Metric = *'ignore_local_polarity'*, the model is found even if the contrast changes locally. This mode can, for example, be useful if the object consists of a part with medium gray value, within which either darker or brighter sub-objects lie. Since in this case the runtime of find_aniso_shape_model increases significantly, it is usually better to create several models that reflect the possible contrast variations of the object with create_aniso_shape_model_xld, and to match them simultaneously with find_aniso_shape_models.

The above three metrics can only be applied to single-channel images. If a multichannel image is used as the model image or as the search image, only the first channel will be used (and no error message will be returned).

If Metric = *'ignore_color_polarity'*, the model is found even if the color contrast changes locally. This is, for example, the case if parts of the object can change their color, e.g., from red to green. In particular, this mode is useful if it is not known in advance in which channels the object is visible. In this mode, the runtime of find_aniso_shape_model can also increase significantly. The metric *'ignore_color_polarity'* can be used for images with an arbitrary number of channels. If it is used for single-channel images it has the same effect as *'ignore_local_polarity'*. It should be noted that for Metric = *'ignore_color_polarity'* the channels do not need to contain a spectral subdivision of the light (like in an RGB image). The channels can, for example, also contain images of the same object that were obtained by illuminating the object from different directions.

Note that the first two metrics (*'use_polarity'* and *'ignore_global_polarity'*) can only be selected if all Contours provide the attribute *'edge_direction'*, which defines the polarity of the edges. For more information about contour attributes like *'edge_direction'* see get_contour_attrib_xld. Otherwise, these two metrics can be selected with the operator set_shape_model_metric, which determines the polarity of the edges from an image.

MinContrast: With MinContrast, it can be determined which contrast the object edges must at least have in the recognition performed by find_aniso_shape_model. In other words, this parameter separates the object from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, MinContrast should be set to 10. If multichannel images are used for the model and the search images, and if the parameter Metric is set to *'ignore_color_polarity'* (see above) the noise in one channel must be multiplied by the square root of the number of channels to determine MinContrast. If, for example, the gray values fluctuate within a range of 10 gray levels in a single channel and the image is a three-channel image MinContrast should be set to 17. If the model should be recognized in very low contrast images, MinContrast must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, MinContrast should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the position and rotation of the model are extracted robustly and accurately by find_aniso_shape_model.

Complete pregeneration of the model

Optionally, a second value can be passed in Optimization. This value determines whether the model is pregenerated completely or not. To do so, the second value of Optimization must be set to either *'pregeneration'*

or *'no_pregeneration'*. If the second value is not used (i.e., if only one value is passed), the mode that is set with `set_system('pregenerate_shape_models',...)` is used. With the default value (*'pregenerate_shape_models'* = *'false'*), the model is not pregenerated completely. The complete pregeneration of the model normally leads to slightly lower runtimes because the model does not need to be transformed at runtime. However, in this case, the memory requirements and the time required to create the model are significantly higher. It should also be noted that it cannot be expected that the two modes return exactly identical results because transforming the model at runtime necessarily leads to different internal data for the transformed models than pregenerating the transformed models. For example, if the model is not pregenerated completely, `find_aniso_shape_model` typically returns slightly lower scores, which may require setting a slightly lower value for `MinScore` than for a completely pregenerated model. Furthermore, the poses obtained by interpolation may differ slightly in the two modes. If maximum accuracy is desired, the pose of the model should be determined by least-squares adjustment.

If a complete pregeneration of the model is selected, the model is pregenerated for the selected angle and scale range and stored in memory. The memory required to store the model is proportional to the number of angle steps, the number of scale steps, and the number of points in the model. Hence, if `AngleStep`, `ScaleRStep`, or `ScaleCStep` are too small or `AngleExtent` or the range of scales are too big, it may happen that the model no longer fits into the (virtual) memory. In this case, `AngleStep`, `ScaleRStep`, or `ScaleCStep` must be enlarged or `AngleExtent` or the range of scales must be reduced. In any case, it is desirable that the model completely fits into the main memory, because this avoids paging by the operating system, and hence the time to find the object will be much smaller. Since angles can be determined with subpixel resolution by `find_aniso_shape_model`, `AngleStep` $\geq 1°$ and `ScaleRStep`, `ScaleCStep` ≥ 0.02 can be selected for models of a diameter smaller than about 200 pixels.

If `AngleStep` = *'auto'* or `ScaleRStep`, `ScaleCStep` = *'auto'* is selected, `create_aniso_shape_model_xld` automatically determines a suitable angle or scale step length, respectively, based on the size of the model. The automatically computed angle and scale step lengths can be queried using `get_shape_model_params`.

If a complete pregeneration of the model is not selected, the model is only created in a reference pose on each pyramid level. In this case, the model must be transformed to the different angles and scales at runtime in `find_aniso_shape_model`. Because of this, the recognition of the model might require slightly more time.

Note that pregenerated shape models are tailored to a specific image size. For runtime reasons using images of different sizes during the search with the same model in parallel is not supported. In this case, copies of the same model must be used, otherwise the program may crash!

Attention

The XLD contours passed in `Contours` should have been scaled to approximately the average size of the object in the search images. This means that the products `ScaleRMin` × `ScaleRMax` and `ScaleCMin` × `ScaleCMax` should be approximately equal to 1.

Note that, in contrast to the operator `create_aniso_shape_model`, it is not possible to specify a minimum size of the model components. To avoid small model components in the shape model, short contours can be eliminated before calling `create_aniso_shape_model_xld` with the operator `select_contours_xld`.

Parameters

▷ **`Contours`** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contours that will be used to create the model.

▷ **`NumLevels`** (input_control) ..... integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: `NumLevels` ∈ {'auto', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **`AngleStart`** (input_control) ..... angle.rad ⇝ *real*
Smallest rotation of the pattern.
Default: -0.39
Suggested values: `AngleStart` ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **`AngleExtent`** (input_control) ..... angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: `AngleExtent` ∈ {6.29, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **AngleStep** (input_control) . . . . . angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: AngleStep ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep >= 0 && AngleStep <= pi / 2

▷ **ScaleRMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in the row direction.
Default: 0.9
Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the pattern in the row direction.
Default: 1.1
Suggested values: ScaleRMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleRMax >= ScaleRMin

▷ **ScaleRStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in the row direction.
Default: 'auto'
Suggested values: ScaleRStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleRStep >= 0

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern in the column direction.
Default: 0.9
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the pattern in the column direction.
Default: 1.1
Suggested values: ScaleCMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleCMax >= ScaleCMin

▷ **ScaleCStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution) in the column direction.
Default: 'auto'
Suggested values: ScaleCStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleCStep >= 0

▷ **Optimization** (input_control) . . . . . string(-array) ⇝ *string*
Kind of optimization and optionally method used for generating the model.
Default: 'auto'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high', 'pregeneration', 'no_pregeneration'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'ignore_local_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer*
Minimum contrast of the objects in the search images.
Default: 5
Suggested values: MinContrast ∈ {1, 2, 3, 5, 7, 10, 20, 30, 40}

▷ **ModelID** (output_control) . . . . . shape_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_aniso_shape_model_xld returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameter NumLevels is chosen such that the model contains too few points, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`read_contour_xld_dxf`, `edges_sub_pix`, `select_contours_xld`

Possible Successors

`find_aniso_shape_model`, `find_aniso_shape_models`, `get_shape_model_params`, `clear_shape_model`, `write_shape_model`, `set_shape_model_origin`, `set_shape_model_param`, `set_shape_model_metric`, `set_shape_model_clutter`

Alternatives

`create_generic_shape_model`

See also

`set_system`, `get_system`

Module

Matching

create_generic_shape_model (: : : ModelID)

Create a shape model.

The operator `create_generic_shape_model` creates a shape model and returns its handle in `ModelID`.

The created model is not trained. See `train_generic_shape_model` for information about training a model.

Parameters

▷ **ModelID** (output_control) ..... shape_model ⇝ *handle*
Handle of the shape model.

Result

If successful the operator `create_generic_shape_model` returns the value 2 (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`set_generic_shape_model_param`, `train_generic_shape_model`, `write_shape_model`, `set_shape_model_origin`

Module

Matching

create_scaled_shape_model (Template : : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleMin, ScaleMax, ScaleStep, Optimization, Metric, Contrast, MinContrast : ModelID)

Prepare an isotropically scaled shape model for matching.

The operator create_scaled_shape_model prepares a template, which is passed in the image Template, as an isotropically scaled shape model used for matching. The ROI of the model is passed as the domain of Template.

The output parameter ModelID is a handle for this model, which is used in subsequent calls to find_scaled_shape_model. The center of gravity of the domain (region) of the model image Template is used as the origin (reference point) of the model. A different origin can be set with set_shape_model_origin. The model is generated using multiple image pyramid levels and is stored in memory. If a complete pregeneration of the model is selected (see below), the model is generated at multiple rotations and scales on each level. The model can be extended by clutter parameters with set_shape_model_clutter.

Input parameters in detail

NumLevels: The number of pyramid levels is determined with the parameter NumLevels. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, NumLevels must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. This can be checked using the output of inspect_shape_model. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, create_scaled_shape_model returns with an error message.

If NumLevels is set to *'auto'* (or *0* for backwards compatibility), create_scaled_shape_model determines the number of pyramid levels automatically. The automatically computed number of pyramid levels can be queried using get_shape_model_params. In rare cases, it might happen that create_scaled_shape_model determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for MinScore or Greediness in find_scaled_shape_model in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in find_scaled_shape_model may increase. In these cases, the number of pyramid levels should be selected using the output of inspect_shape_model.

AngleStart, AngleExtent and AngleStep: The parameters AngleStart and AngleExtent determine the range of possible rotations, in which the model can occur in the image. Note that the model can only be found in this range of angles by find_scaled_shape_model. The parameter AngleStep determines the step length within the selected range of angles. Hence, if subpixel accuracy is not specified in find_scaled_shape_model, this parameter specifies the accuracy that is achievable for the angles in find_scaled_shape_model. AngleStep should be chosen based on the size of the object. Smaller models do not have many different discrete rotations in the image, and hence AngleStep should be chosen larger for smaller models. If AngleExtent is not an integer multiple of AngleStep, AngleStep is modified accordingly. To ensure that for model instances without rotation angle values of exactly 0.0 are returned by find_scaled_shape_model, the range of possible rotations is modified as follows: If there is no positive integer value n such that AngleStart plus n times AngleStep is exactly 0.0, AngleStart is decreased by up to AngleStep and AngleExtent is increased by AngleStep.

ScaleMin, ScaleMax, and ScaleStep: The parameters ScaleMin and ScaleMax determine the range of possible scales (sizes) of the model. A scale of 1 corresponds to the original size of the model. The parameter ScaleStep determines the step length within the selected range of scales. Hence, if subpixel accuracy is not specified in find_scaled_shape_model, this parameter specifies the accuracy that is achievable for the scales in find_scaled_shape_model. Like AngleStep, ScaleStep should be chosen based on the size of the object. If the range of scales is not an integer multiple of ScaleStep, ScaleStep is modified accordingly. To ensure that for model instances that are not scaled scale values of exactly 1.0 are returned by find_scaled_shape_model, the range of possible scales is modified as follows: If there is no positive integer value n such that ScaleMin plus n times ScaleStep is exactly 1.0, ScaleMin is decreased by up to ScaleStep and ScaleMax is increased such that the range of possible scales is increased by ScaleStep.

Optimization: For particularly large models, it may be useful to reduce the number of model points by setting Optimization to a value different from *'none'*. If Optimization = *'none'*, all model points are stored. In all other cases, the number of points is reduced according to the value of Optimization. If the number of points is reduced, it may be necessary in find_scaled_shape_model to set the parameter Greediness to a smaller value, e.g., *0.7* or *0.8*. For small models, the reduction of the number of model

points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined.

If Optimization is set to *'auto'*, create_scaled_shape_model automatically determines the reduction of the number of model points.

Metric: The parameter Metric determines the conditions under which the model is recognized in the image.

If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background.

If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_scaled_shape_model will increase slightly in this case.

If Metric = *'ignore_local_polarity'*, the model is found even if the contrast changes locally. This mode can, for example, be useful if the object consists of a part with medium gray value, within which either darker or brighter sub-objects lie. Since in this case the runtime of find_scaled_shape_model increases significantly, it is usually better to create several models that reflect the possible contrast variations of the object with create_scaled_shape_model, and to match them simultaneously with find_scaled_shape_models.

The above three metrics can only be applied to single-channel images. If a multichannel image is used as the model image or as the search image, only the first channel will be used (and no error message will be returned).

If Metric = *'ignore_color_polarity'*, the model is found even if the color contrast changes locally. This is, for example, the case if parts of the object can change their color, e.g., from red to green. In particular, this mode is useful if it is not known in advance in which channels the object is visible. In this mode, the runtime of find_scaled_shape_model can also increase significantly. The metric *'ignore_color_polarity'* can be used for images with an arbitrary number of channels. If it is used for single-channel images it has the same effect as *'ignore_local_polarity'*. It should be noted that for Metric = *'ignore_color_polarity'* the number of channels in the model creation with create_scaled_shape_model and in the search with find_scaled_shape_model can be different. This can, for example, be used to create a model from a synthetically generated single-channel image. Furthermore, it should be noted that the channels do not need to contain a spectral subdivision of the light (like in an RGB image). The channels can, for example, also contain images of the same object that were obtained by illuminating the object from different directions.

Contrast: The parameter Contrast determines the contrast the model points must have. The contrast is a measure for local gray value differences between the object and the background and between different parts of the object. Contrast should be chosen such that only the significant features of the template are used for the model. Contrast can also contain a tuple with two values. In this case, the model is segmented using a method similar to the hysteresis threshold method used in edges_image. Here, the first element of the tuple determines the lower threshold, while the second element determines the upper threshold. For more information about the hysteresis threshold method, see hysteresis_threshold. Optionally, Contrast can contain a third value as the last element of the tuple. This value determines a threshold for the selection of significant model components based on the size of the components, i.e., components that have fewer points than the minimum size thus specified are suppressed. As the minimum size is applied on the extent of the components, the derived model contours can still be smaller than the specified minimum size. This threshold for the minimum size is divided by two for each successive pyramid level. If small model components should be suppressed, but hysteresis thresholding should not be performed, nevertheless three values must be specified in Contrast. In this case, the first two values can simply be set to identical values. The effect of this parameter can be checked in advance with inspect_shape_model.

If Contrast is set to *'auto'*, create_scaled_shape_model determines the three above described values automatically. Besides, only the contrast (*'auto_contrast'*), the hysteresis thresholds (*'auto_contrast_hyst'*), or the minimum size (*'auto_min_size'*) can be determined automatically. The remaining values that are not determined automatically can additionally be passed in the form of a tuple. Also various combinations are allowed: If, for example, *['auto_contrast','auto_min_size']* is passed, both the contrast and the minimum size are determined automatically. If *['auto_min_size',20,30]* is passed, the minimum size is determined automatically while the hysteresis thresholds are set to *20* and *30*, etc. In certain cases, it might happen that the automatic determination of the contrast thresholds is not satisfying. For example, a manual setting of these parameters should be preferred if certain model components should be included or suppressed because of application-specific reasons or if the object contains several different contrasts. Therefore, the contrast thresholds should be automatically determined with determine_shape_model_params and subsequently verified using inspect_shape_model before calling create_scaled_shape_model.

Note that MinContrast influences the automatic contrast estimation, and hence also the estimation of the minimum size.

MinContrast: With MinContrast, it can be determined which contrast the model must at least have in the recognition performed by find_scaled_shape_model. In other words, this parameter separates the model from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, MinContrast should be set to 10. If multichannel images are used for the model and the search images, and if the parameter Metric is set to *'ignore_color_polarity'* (see above) the noise in one channel must be multiplied by the square root of the number of channels to determine MinContrast. If, for example, the gray values fluctuate within a range of 10 gray levels in a single channel and the image is a three-channel image MinContrast should be set to 17. Obviously, MinContrast must be smaller than Contrast. If the model should be recognized in very low contrast images, MinContrast must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, MinContrast should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the position and rotation of the model are extracted robustly and accurately by find_scaled_shape_model.

If MinContrast is set to *'auto'*, the minimum contrast is determined automatically based on the noise in the model image. Consequently, an automatic determination only makes sense if the image noise during the recognition is similar to the noise in the model image. Furthermore, in some cases it is advisable to increase the automatically determined value in order to increase the robustness against occlusions (see above). The automatically computed minimum contrast can be queried using get_shape_model_params.

Complete pregeneration of the model

Optionally, a second value can be passed in Optimization. This value determines whether the model is pregenerated completely or not. To do so, the second value of Optimization must be set to either *'pregeneration'* or *'no_pregeneration'*. If the second value is not used (i.e., if only one value is passed), the mode that is set with set_system('pregenerate_shape_models',...) is used. With the default value (*'pregenerate_shape_models'* = *'false'*), the model is not pregenerated completely. The complete pregeneration of the model normally leads to slightly lower runtimes because the model does not need to be transformed at runtime. However, in this case, the memory requirements and the time required to create the model are significantly higher. It should also be noted that it cannot be expected that the two modes return exactly identical results because transforming the model at runtime necessarily leads to different internal data for the transformed models than pregenerating the transformed models. For example, if the model is not pregenerated completely, find_scaled_shape_model typically returns slightly lower scores, which may require setting a slightly lower value for MinScore than for a completely pregenerated model. Furthermore, the poses obtained by interpolation may differ slightly in the two modes. If maximum accuracy is desired, the pose of the model should be determined by least-squares adjustment.

If a complete pregeneration of the model is selected, the model is pregenerated for the selected angle and scale range and stored in memory. The memory required to store the model is proportional to the number of angle steps, the number of scale steps, and the number of points in the model. Hence, if AngleStep or ScaleStep are too small or AngleExtent or the range of scales are too big, it may happen that the model no longer fits into the (virtual) memory. In this case, either AngleStep or ScaleStep must be enlarged or AngleExtent or the range of scales must be reduced. In any case, it is desirable that the model completely fits into the main memory, because this avoids paging by the operating system, and hence the time to find the object will be much smaller. Since angles can be determined with subpixel resolution by find_scaled_shape_model, AngleStep $\geq 1^\circ$ and ScaleStep ≥ 0.02 can be selected for models of a diameter smaller than about 200 pixels.

If AngleStep = *'auto'* or ScaleStep = *'auto'* (or *0* for backwards compatibility in both cases) is selected, create_scaled_shape_model automatically determines a suitable angle or scale step length, respectively, based on the size of the model. The automatically computed angle and scale step lengths can be queried using get_shape_model_params.

If a complete pregeneration of the model is not selected, the model is only created in a reference pose on each pyramid level. In this case, the model must be transformed to the different angles and scales at runtime in find_scaled_shape_model. Because of this, the recognition of the model might require slightly more time.

Note that pregenerated shape models are tailored to a specific image size. For runtime reasons using images of different sizes during the search with the same model in parallel is not supported. In this case, copies of the same model must be used, otherwise the program may crash!

Parameters

▷ **Template** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: NumLevels ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the pattern.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **AngleStep** (input_control) . . . . . angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: AngleStep ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep >= 0 && AngleStep <= pi / 2

▷ **ScaleMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern.
Default: 0.9
Suggested values: ScaleMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleMin > 0

▷ **ScaleMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the pattern.
Default: 1.1
Suggested values: ScaleMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleMax >= ScaleMin

▷ **ScaleStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution).
Default: 'auto'
Suggested values: ScaleStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleStep >= 0

▷ **Optimization** (input_control) . . . . . string(-array) ⇝ *string*
Kind of optimization and optionally method used for generating the model.
Default: 'auto'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high', 'pregeneration', 'no_pregeneration'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **Contrast** (input_control) . . . . . number(-array) ⇝ *integer* / string
Threshold or hysteresis thresholds for the contrast of the object in the template image and optionally minimum size of the object parts.
Default: 'auto'
Suggested values: Contrast ∈ {'auto', 'auto_contrast', 'auto_contrast_hyst', 'auto_min_size', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer* / string
Minimum contrast of the objects in the search images.
Default: 'auto'
Suggested values: MinContrast ∈ {'auto', 1, 2, 3, 5, 7, 10, 20, 30, 40}
Restriction: MinContrast < Contrast

▷ **ModelID** (output_control) ..... shape_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_scaled_shape_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameters NumLevels and Contrast are chosen such that the model contains too few points, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

draw_region, reduce_domain, threshold

Possible Successors

find_scaled_shape_model, find_scaled_shape_models, get_shape_model_params, clear_shape_model, write_shape_model, set_shape_model_origin, set_shape_model_clutter

Alternatives

create_generic_shape_model

See also

set_system, get_system

Module

Matching

create_scaled_shape_model_xld (Contours : : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleMin, ScaleMax, ScaleStep, Optimization, Metric, MinContrast : ModelID)

Prepare an isotropically scaled shape model for matching from XLD contours.

The operator create_scaled_shape_model_xld creates an isotropically scaled shape model used for matching from the XLD contours passed in Contours. The XLD contours represent the gray value edges of the object to be searched for. In contrast to the operator create_scaled_shape_model, which creates a shape model from a template image, the operator create_scaled_shape_model_xld creates the shape model from XLD contours, i.e., without the use of a template image.

The output parameter ModelID is a handle for this model, which is used in subsequent calls to find_scaled_shape_model. The center of gravity of the smallest surrounding rectangle of the Contours that is parallel to the coordinate axes is used as the origin (reference point) of the model. A different origin can be set with set_shape_model_origin. The model is generated for multiple image pyramid levels and is stored in memory. If a complete pregeneration of the model is selected (see below), the model is generated at multiple rotations and scales on each level. The model can be extended by clutter parameters with set_shape_model_clutter.

Input parameters in detail

NumLevels: The number of pyramid levels is determined with the parameter NumLevels. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, NumLevels must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, create_scaled_shape_model_xld returns with an error message.

If NumLevels is set to *'auto'*, create_scaled_shape_model_xld determines the number of pyramid levels automatically. The computed number of pyramid levels can be queried using get_shape_model_params. In rare cases, it might happen that create_scaled_shape_model_xld determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for MinScore or Greediness in find_scaled_shape_model in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in find_scaled_shape_model may increase. In these cases, the number of pyramid levels should be selected manually.

AngleStart, AngleExtent, and AngleStep: The parameters AngleStart and AngleExtent determine the range of possible rotations, in which the object can occur in the image during the search. Note that the object can only be found in this range of angles by find_scaled_shape_model. The parameter AngleStep determines the step length within the selected range of angles. Hence, if subpixel accuracy is not specified in find_scaled_shape_model, this parameter specifies the accuracy that is achievable for the angles in find_scaled_shape_model. AngleStep should be chosen based on the size of the object. Smaller models do not have many different discrete rotations in the image, and hence AngleStep should be chosen larger for smaller models. If AngleExtent is not an integer multiple of AngleStep, AngleStep is modified accordingly. To ensure that for model instances without rotation angle values of exactly 0.0 are returned by find_scaled_shape_model, the range of possible rotations is modified as follows: If there is no positive integer value n such that AngleStart plus n times AngleStep is exactly 0.0, AngleStart is decreased by up to AngleStep and AngleExtent is increased by AngleStep.

ScaleMin, ScaleMax, and ScaleStep: The parameters ScaleMin and ScaleMax determine the range of possible scales (sizes) of the object. A scale of 1 corresponds to the original size of the model. The parameter ScaleStep determines the step length within the selected range of scales. Hence, if subpixel accuracy is not specified in find_scaled_shape_model, this parameter specifies the accuracy that is achievable for the scales in find_scaled_shape_model. Like AngleStep, ScaleStep should be chosen based on the size of the object. If the range of scales is not an integer multiple of ScaleStep, ScaleStep is modified accordingly. To ensure that for model instances that are not scaled scale values of exactly 1.0 are returned by find_scaled_shape_model, the range of possible scales is modified as follows: If there is no positive integer value n such that ScaleMin plus n times ScaleStep is exactly 1.0, ScaleMin is decreased by up to ScaleStep and ScaleMax is increased such that the range of possible scales is increased by ScaleStep.

Optimization: For particularly large models, it may be useful to reduce the number of model points by setting Optimization to a value different from *'none'*. If Optimization = *'none'*, all model points are stored. In all other cases, the number of points is reduced according to the value of Optimization. If the number of points is reduced, it may be necessary in find_scaled_shape_model to set the parameter Greediness to a smaller value, e.g., *0.7* or *0.8*. For small models, the reduction of the number of model points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined.

If Optimization is set to *'auto'*, create_scaled_shape_model_xld automatically determines the reduction of the number of model points.

Metric: The parameter Metric determines the conditions under which the model is recognized in the image.

If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background.

If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_scaled_shape_model will increase slightly in this case.

Note that the metrics (*'use_polarity'* and *'ignore_global_polarity'*) can only be selected if all Contours provide the attribute *'edge_direction'*, which defines the polarity of the edges. This attribute is available for contours created, e.g., with edges_sub_pix with the parameter Method set to, e.g., *'canny'*. Otherwise, these two metrics can be selected with the operator set_shape_model_metric, which determines the polarity of the edges from an image.

If Metric = *'ignore_local_polarity'*, the model is found even if the contrast changes locally. This mode can, for example, be useful if the object consists of a part with medium gray value, within which either darker or brighter sub-objects lie. Since in this case the runtime of find_scaled_shape_model increases significantly, it is usually better to create several models that reflect the possible contrast varia-

tions of the object with create_scaled_shape_model_xld, and to match them simultaneously with find_scaled_shape_models.

The above three metrics can only be applied to single-channel images. If a multichannel image is used as the model image or as the search image, only the first channel will be used (and no error message will be returned).

If Metric = *'ignore_color_polarity'*, the model is found even if the color contrast changes locally. This is, for example, the case if parts of the object can change their color, e.g., from red to green. In particular, this mode is useful if it is not known in advance in which channels the object is visible. In this mode, the runtime of find_scaled_shape_model can also increase significantly. The metric *'ignore_color_polarity'* can be used for images with an arbitrary number of channels. If it is used for single-channel images it has the same effect as *'ignore_local_polarity'*. It should be noted that for Metric = *'ignore_color_polarity'* the channels do not need to contain a spectral subdivision of the light (like in an RGB image). The channels can, for example, also contain images of the same object that were obtained by illuminating the object from different directions.

Note that the first two metrics (*'use_polarity'* and *'ignore_global_polarity'*) can only be selected if all Contours provide the attribute *'edge_direction'*, which defines the polarity of the edges. For more information about contour attributes like *'edge_direction'* see get_contour_attrib_xld. Otherwise, these two metrics can be selected with the operator set_shape_model_metric, which determines the polarity of the edges from an image.

MinContrast: With MinContrast, it can be determined which contrast the object edges must at least have in the recognition performed by find_scaled_shape_model. In other words, this parameter separates the object from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, MinContrast should be set to 10. If multichannel images are used for the model and the search images, and if the parameter Metric is set to *'ignore_color_polarity'* (see above) the noise in one channel must be multiplied by the square root of the number of channels to determine MinContrast. If, for example, the gray values fluctuate within a range of 10 gray levels in a single channel and the image is a three-channel image MinContrast should be set to 17. If the model should be recognized in very low contrast images, MinContrast must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, MinContrast should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the position and rotation of the model are extracted robustly and accurately by find_scaled_shape_model.

Complete pregeneration of the model

Optionally, a second value can be passed in Optimization. This value determines whether the model is pregenerated completely or not. To do so, the second value of Optimization must be set to either *'pregeneration'* or *'no_pregeneration'*. If the second value is not used (i.e., if only one value is passed), the mode that is set with set_system('pregenerate_shape_models',...) is used. With the default value (*'pregenerate_shape_models'* = *'false'*), the model is not pregenerated completely. The complete pregeneration of the model normally leads to slightly lower runtimes because the model does not need to be transformed at runtime. However, in this case, the memory requirements and the time required to create the model are significantly higher. It should also be noted that it cannot be expected that the two modes return exactly identical results because transforming the model at runtime necessarily leads to different internal data for the transformed models than pregenerating the transformed models. For example, if the model is not pregenerated completely, find_scaled_shape_model typically returns slightly lower scores, which may require setting a slightly lower value for MinScore than for a completely pregenerated model. Furthermore, the poses obtained by interpolation may differ slightly in the two modes. If maximum accuracy is desired, the pose of the model should be determined by least-squares adjustment.

If a complete pregeneration of the model is selected, the model is pregenerated for the selected angle and scale range and stored in memory. The memory required to store the model is proportional to the number of angle steps, the number of scale steps, and the number of points in the model. Hence, if AngleStep or ScaleStep are too small or AngleExtent or the range of scales are too big, it may happen that the model no longer fits into the (virtual) memory. In this case, either AngleStep or ScaleStep must be enlarged or AngleExtent or the range of scales must be reduced. In any case, it is desirable that the model completely fits into the main memory, because this avoids paging by the operating system, and hence the time to find the object will be much smaller. Since angles can be determined with subpixel resolution by find_scaled_shape_model, AngleStep $\geq 1^\circ$ and ScaleStep ≥ 0.02 can be selected for models of a diameter smaller than about 200 pixels.

If AngleStep = *'auto'* or ScaleStep = *'auto'* is selected, create_scaled_shape_model_xld automatically determines a suitable angle or scale step length, respectively, based on the size of the model. The

automatically computed angle and scale step lengths can be queried using get_shape_model_params.

If a complete pregeneration of the model is not selected, the model is only created in a reference pose on each pyramid level. In this case, the model must be transformed to the different angles and scales at runtime in find_scaled_shape_model. Because of this, the recognition of the model might require slightly more time.

Note that pregenerated shape models are tailored to a specific image size. For runtime reasons using images of different sizes during the search with the same model in parallel is not supported. In this case, copies of the same model must be used, otherwise the program may crash!

Attention

The XLD contours passed in Contours should have been scaled to approximately the average size of the object in the search images. This means that the product ScaleMin × ScaleMax should be approximately equal to 1.

Note that, in contrast to the operator create_scaled_shape_model, it is not possible to specify a minimum size of the model components. To avoid small model components in the shape model, short contours can be eliminated before calling create_scaled_shape_model_xld with the operator select_contours_xld.

Parameters

▷ **Contours** (input_object) . . . . . xld_cont(-array) ⇝ *object*
Input contours that will be used to create the model.

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: NumLevels ∈ {'auto', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the pattern.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **AngleStep** (input_control) . . . . . angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: AngleStep ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep >= 0 && AngleStep <= pi / 2

▷ **ScaleMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the pattern.
Default: 0.9
Suggested values: ScaleMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleMin > 0

▷ **ScaleMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the pattern.
Default: 1.1
Suggested values: ScaleMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleMax >= ScaleMin

▷ **ScaleStep** (input_control) . . . . . number ⇝ *real* / string
Scale step length (resolution).
Default: 'auto'
Suggested values: ScaleStep ∈ {'auto', 0.01, 0.02, 0.05, 0.1, 0.15, 0.2}
Restriction: ScaleStep >= 0

▷ **Optimization** (input_control) . . . . . string(-array) ⇝ *string*
Kind of optimization and optionally method used for generating the model.
Default: 'auto'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high', 'pregeneration', 'no_pregeneration'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'ignore_local_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **MinContrast** (input_control) . . . . . number ⇝ *integer*
Minimum contrast of the objects in the search images.
Default: 5
Suggested values: MinContrast ∈ {1, 2, 3, 5, 7, 10, 20, 30, 40}

▷ **ModelID** (output_control) . . . . . shape_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_scaled_shape_model_xld returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameter NumLevels is chosen such that the model contains too few points, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

read_contour_xld_dxf, edges_sub_pix, select_contours_xld

Possible Successors

find_scaled_shape_model, find_scaled_shape_models, get_shape_model_params, clear_shape_model, write_shape_model, set_shape_model_origin, set_shape_model_param, set_shape_model_metric, set_shape_model_clutter

Alternatives

create_generic_shape_model

See also

set_system, get_system

Module

Matching

```
create_shape_model ( Template : : NumLevels, AngleStart,
    AngleExtent, AngleStep, Optimization, Metric, Contrast,
    MinContrast : ModelID )
```

Prepare a shape model for matching.

The operator create_shape_model prepares a template, which is passed in the image Template, as a shape model used for matching. The ROI of the model is passed as the domain of Template.

The output parameter ModelID is a handle for this model, which is used in subsequent calls to find_shape_model. The center of gravity of the domain (region) of the model image Template is used as the origin (reference point) of the model. A different origin can be set with set_shape_model_origin. The model is generated using multiple image pyramid levels and is stored in memory. If a complete pregeneration of the model is selected (see below), the model is generated at multiple rotations on each level. The model can be extended by clutter parameters with set_shape_model_clutter.

Input parameters in detail

NumLevels: The number of pyramid levels is determined with the parameter NumLevels. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the

other hand, NumLevels must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. This can be checked using the output of inspect_shape_model. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, create_shape_model returns with an error message.

If NumLevels is set to *'auto'* (or *0* for backwards compatibility), create_shape_model determines the number of pyramid levels automatically. The automatically computed number of pyramid levels can be queried using get_shape_model_params. In rare cases, it might happen that create_shape_model determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for MinScore or Greediness in find_shape_model in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in find_shape_model may increase. In these cases, the number of pyramid levels should be selected using the output of inspect_shape_model.

AngleStart, AngleExtent, and AngleStep: The parameters AngleStart and AngleExtent determine the range of possible rotations, in which the model can occur in the image. Note that the model can only be found in this range of angles by find_shape_model. The parameter AngleStep determines the step length within the selected range of angles. Hence, if subpixel accuracy is not specified in find_shape_model, this parameter specifies the accuracy that is achievable for the angles in find_shape_model. AngleStep should be chosen based on the size of the object. Smaller models do not possess many different discrete rotations in the image, and hence AngleStep should be chosen larger for smaller models. If AngleExtent is not an integer multiple of AngleStep, AngleStep is modified accordingly. To ensure that for model instances without rotation angle values of exactly 0.0 are returned by find_shape_model, the range of possible rotations is modified as follows: If there is no positive integer value n such that AngleStart plus n times AngleStep is exactly 0.0, AngleStart is decreased by up to AngleStep and AngleExtent is increased by AngleStep.

Optimization: For particularly large models, it may be useful to reduce the number of model points by setting Optimization to a value different from *'none'*. If Optimization = *'none'*, all model points are stored. In all other cases, the number of points is reduced according to the value of Optimization. If the number of points is reduced, it may be necessary in find_shape_model to set the parameter Greediness to a smaller value, e.g., *0.7* or *0.8*. For small models, the reduction of the number of model points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined.

If Optimization is set to *'auto'*, create_shape_model automatically determines the reduction of the number of model points.

Metric: The parameter Metric determines the conditions under which the model is recognized in the image.

If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background.

If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_shape_model will increase slightly in this case.

If Metric = *'ignore_local_polarity'*, the model is found even if the contrast changes locally. This mode can, for example, be useful if the object consists of a part with medium gray value, within which either darker or brighter sub-objects lie. Since in this case the runtime of find_shape_model increases significantly, it is usually better to create several models that reflect the possible contrast variations of the object with create_shape_model, and to match them simultaneously with find_shape_models.

The above three metrics can only be applied to single-channel images. If a multichannel image is used as the model image or as the search image, only the first channel will be used (and no error message will be returned).

If Metric = *'ignore_color_polarity'*, the model is found even if the color contrast changes locally. This is, for example, the case if parts of the object can change their color, e.g., from red to green. In particular, this mode is useful if it is not known in advance in which channels the object is visible. In this mode, the runtime of find_shape_model can also increase significantly. The metric *'ignore_color_polarity'* can be used for images with an arbitrary number of channels. If it is used for single-channel images it has the same effect as *'ignore_local_polarity'*. It should be noted that for Metric = *'ignore_color_polarity'* the number of channels in the model creation with create_shape_model and in the search with find_shape_model can

be different. This can, for example, be used to create a model from a synthetically generated single-channel image. Furthermore, it should be noted that the channels do not need to contain a spectral subdivision of the light (like in an RGB image). The channels can, for example, also contain images of the same object that were obtained by illuminating the object from different directions.

Contrast: The parameter Contrast determines the contrast the model points must have. The contrast is a measure for local gray value differences between the object and the background and between different parts of the object. Contrast should be chosen such that only the significant features of the template are used for the model. Contrast can also contain a tuple with two values. In this case, the model is segmented using a method similar to the hysteresis threshold method used in edges_image. Here, the first element of the tuple determines the lower threshold, while the second element determines the upper threshold. For more information about the hysteresis threshold method, see hysteresis_threshold. Optionally, Contrast can contain a third value as the last element of the tuple. This value determines a threshold for the selection of significant model components based on the size of the components, i.e., components that have fewer points than the minimum size thus specified are suppressed. As the minimum size is applied on the extent of the components, the derived model contours can still be smaller than the specified minimum size. This threshold for the minimum size is divided by two for each successive pyramid level. If small model components should be suppressed, but hysteresis thresholding should not be performed, nevertheless three values must be specified in Contrast. In this case, the first two values can simply be set to identical values. The effect of this parameter can be checked in advance with inspect_shape_model.

If Contrast is set to *'auto'*, create_shape_model determines the three above described values automatically. only the contrast (*'auto_contrast'*), the hysteresis thresholds (*'auto_contrast_hyst'*), or the minimum size (*'auto_min_size'*) can be determined automatically. The remaining values that are not determined automatically can additionally be passed in the form of a tuple. Also various combinations are allowed: If, for example, *['auto_contrast','auto_min_size']* is passed, both the contrast and the minimum size are determined automatically. If *['auto_min_size',20,30]* is passed, the minimum size is determined automatically while the hysteresis thresholds are set to *20* and *30*, etc. In certain cases, it might happen that the automatic determination of the contrast thresholds is not satisfying. For example, a manual setting of these parameters should be preferred if certain model components should be included or suppressed because of application-specific reasons or if the object contains several different contrasts. Therefore, the contrast thresholds should be automatically determined with determine_shape_model_params and subsequently verified using inspect_shape_model before calling create_shape_model. Note that MinContrast influences the automatic contrast estimation, and hence also the estimation of the minimum size.

MinContrast: With MinContrast, it can be determined which contrast the model must at least have in the recognition performed by find_shape_model. In other words, this parameter separates the model from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, MinContrast should be set to 10. If multichannel images are used for the model and the search images, and if the parameter Metric is set to *'ignore_color_polarity'* (see above) the noise in one channel must be multiplied by the square root of the number of channels to determine MinContrast. If, for example, the gray values fluctuate within a range of 10 gray levels in a single channel and the image is a three-channel image MinContrast should be set to 17. Obviously, MinContrast must be smaller than Contrast. If the model should be recognized in very low contrast images, MinContrast must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, MinContrast should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the position and rotation of the model are extracted robustly and accurately by find_shape_model.

If MinContrast is set to *'auto'*, the minimum contrast is determined automatically based on the noise in the model image. Consequently, an automatic determination only makes sense if the image noise during the recognition is similar to the noise in the model image. Furthermore, in some cases it is advisable to increase the automatically determined value in order to increase the robustness against occlusions (see above). The automatically computed minimum contrast can be queried using get_shape_model_params.

Complete pregeneration of the model

Optionally, a second value can be passed in Optimization. This value determines whether the model is pregenerated completely or not. To do so, the second value of Optimization must be set to either *'pregeneration'* or *'no_pregeneration'*. If the second value is not used (i.e., if only one value is passed), the mode that is set with set_system('pregenerate_shape_models',...) is used. With the default value (*'pregenerate_shape_models'* = *'false'*), the model is not pregenerated completely. The complete pregeneration of the model normally leads to slightly lower runtimes because the model does not need to be transformed at runtime. However,

in this case, the memory requirements and the time required to create the model are significantly higher. It should also be noted that it cannot be expected that the two modes return exactly identical results because transforming the model at runtime necessarily leads to different internal data for the transformed models than pregenerating the transformed models. For example, if the model is not pregenerated completely, `find_shape_model` typically returns slightly lower scores, which may require setting a slightly lower value for `MinScore` than for a completely pregenerated model. Furthermore, the poses obtained by interpolation may differ slightly in the two modes. If maximum accuracy is desired, the pose of the model should be determined by least-squares adjustment.

If a complete pregeneration of the model is selected, the model is pregenerated for the selected angle range and stored in memory. The memory required to store the model is proportional to the number of angle steps and the number of points in the model. Hence, if `AngleStep` is too small or `AngleExtent` too big, it may happen that the model no longer fits into the (virtual) memory. In this case, either `AngleStep` must be enlarged or `AngleExtent` must be reduced. In any case, it is desirable that the model completely fits into the main memory, because this avoids paging by the operating system, and hence the time to find the object will be much smaller. Since angles can be determined with subpixel resolution by `find_shape_model`, `AngleStep` ≥ 1 can be selected for models of a diameter smaller than about 200 pixels.

If `AngleStep` $= 'auto'$ (or *0* for backwards compatibility) is selected, `create_shape_model` automatically determines a suitable angle step length based on the size of the model. The automatically computed angle step length can be queried using `get_shape_model_params`.

If a complete pregeneration of the model is not selected, the model is only created in a reference pose on each pyramid level. In this case, the model must be transformed to the different angles and scales at runtime in `find_shape_model`. Because of this, the recognition of the model might require slightly more time.

Note that pregenerated shape models are tailored to a specific image size. For runtime reasons using images of different sizes during the search with the same model in parallel is not supported. In this case, copies of the same model must be used, otherwise the program may crash!

Parameters

▷ **`Template`** (input_object) ..... (multichannel-)image ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **`NumLevels`** (input_control) ..... integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: `NumLevels` ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **`AngleStart`** (input_control) ..... angle.rad ⇝ *real*
Smallest rotation of the pattern.
Default: -0.39
Suggested values: `AngleStart` ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **`AngleExtent`** (input_control) ..... angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: `AngleExtent` ∈ {6.29, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **`AngleStep`** (input_control) ..... angle.rad ⇝ *real* / string
Step length of the angles (resolution).
Default: 'auto'
Suggested values: `AngleStep` ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
Restriction: AngleStep >= 0 && AngleStep <= pi / 2

▷ **`Optimization`** (input_control) ..... string(-array) ⇝ *string*
Kind of optimization and optionally method used for generating the model.
Default: 'auto'
List of values: `Optimization` ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high', 'pregeneration', 'no_pregeneration'}

▷ **`Metric`** (input_control) ..... string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: `Metric` ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **Contrast** (input_control) ..... number(-array) ⇝ *integer* / string
Threshold or hysteresis thresholds for the contrast of the object in the template image and optionally minimum size of the object parts.
Default: 'auto'
Suggested values: Contrast ∈ {'auto', 'auto_contrast', 'auto_contrast_hyst', 'auto_min_size', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}

▷ **MinContrast** (input_control) ..... number ⇝ *integer* / string
Minimum contrast of the objects in the search images.
Default: 'auto'
Suggested values: MinContrast ∈ {'auto', 1, 2, 3, 5, 7, 10, 20, 30, 40}
Restriction: MinContrast < Contrast

▷ **ModelID** (output_control) ..... shape_model ⇝ *handle*
Handle of the model.

Result

If the parameters are valid, the operator create_shape_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameters NumLevels and Contrast are chosen such that the model contains too few points, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

draw_region, reduce_domain, threshold

Possible Successors

find_shape_model, find_shape_models, get_shape_model_params, clear_shape_model, write_shape_model, set_shape_model_origin, set_shape_model_clutter

Alternatives

create_generic_shape_model

See also

set_system, get_system

Module

Matching

create_shape_model_xld (Contours : : NumLevels, AngleStart, AngleExtent, AngleStep, Optimization, Metric, MinContrast : ModelID)

Prepare a shape model for matching from XLD contours.

The operator create_shape_model_xld creates a shape model used for matching from the XLD contours passed in Contours. The XLD contours represent the gray value edges of the object to be searched for. In contrast to the operator create_shape_model, which creates a shape model from a template image, the operator create_shape_model_xld creates the shape model from XLD contours, i.e., without the use of a template image.

The output parameter ModelID is a handle for this model, which is used in subsequent calls to find_shape_model. The center of gravity of the smallest surrounding rectangle of the Contours that is parallel to the coordinate axes is used as the origin (reference point) of the model. A different origin can be set with set_shape_model_origin. The model is generated for multiple image pyramid levels and is stored in memory. If a complete pregeneration of the model is selected (see below), the model is generated at multiple rotations on each level. The model can be extended by clutter parameters with set_shape_model_clutter.

Input parameters in detail

NumLevels: The number of pyramid levels is determined with the parameter NumLevels. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, NumLevels must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, create_shape_model_xld returns with an error message.

If NumLevels is set to *'auto'*, create_shape_model_xld determines the number of pyramid levels automatically. The computed number of pyramid levels can be queried using get_shape_model_params. In rare cases, it might happen that create_shape_model_xld determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for MinScore or Greediness in find_shape_model in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in find_shape_model may increase. In these cases, the number of pyramid levels should be selected manually.

AngleStart, AngleExtent, and AngleStep: The parameters AngleStart and AngleExtent determine the range of possible rotations, in which the object can occur in the image during the search. Note that the object can only be found in this range of angles by find_shape_model. The parameter AngleStep determines the step length within the selected range of angles. Hence, if subpixel accuracy is not specified in find_shape_model, this parameter specifies the accuracy that is achievable for the angles in find_shape_model. AngleStep should be chosen based on the size of the object. Smaller models do not possess many different discrete rotations in the image, and hence AngleStep should be chosen larger for smaller models. If AngleExtent is not an integer multiple of AngleStep, AngleStep is modified accordingly.

To ensure that for model instances without rotation angle values of exactly 0.0 are returned by find_shape_model, the range of possible rotations is modified as follows: If there is no positive integer value n such that AngleStart plus n times AngleStep is exactly 0.0, AngleStart is decreased by up to AngleStep and AngleExtent is increased by AngleStep.

Optimization: For particularly large models, it may be useful to reduce the number of model points by setting Optimization to a value different from *'none'*. If Optimization = *'none'*, all model points are stored. In all other cases, the number of points is reduced according to the value of Optimization. If the number of points is reduced, it may be necessary in find_shape_model to set the parameter Greediness to a smaller value, e.g., *0.7* or *0.8*. For small models, the reduction of the number of model points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined.

If Optimization is set to *'auto'*, create_shape_model_xld automatically determines the reduction of the number of model points.

Metric: The parameter Metric determines the conditions under which the model is recognized in the image.

If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background.

If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_shape_model will increase slightly in this case.

Note that the metrics (*'use_polarity'* and *'ignore_global_polarity'*) can only be selected if all Contours provide the attribute *'edge_direction'*, which defines the polarity of the edges. This attribute is available for contours created, e.g., with edges_sub_pix with the parameter Method set to, e.g., *'canny'*. Otherwise, these two metrics can be selected with the operator set_shape_model_metric, which determines the polarity of the edges from an image.

If Metric = *'ignore_local_polarity'*, the model is found even if the contrast changes locally. This mode can, for example, be useful if the object consists of a part with medium gray value, within which either darker or brighter sub-objects lie. Since in this case the runtime of find_shape_model increases significantly, it is usually better to create several models that reflect the possible contrast variations of the object with create_shape_model_xld, and to match them simultaneously with find_shape_models.

The above three metrics can only be applied to single-channel images. If a multichannel image is used as the model image or as the search image, only the first channel will be used (and no error message will be returned).

If Metric = *'ignore_color_polarity'*, the model is found even if the color contrast changes locally. This is, for example, the case if parts of the object can change their color, e.g., from red to green. In particular, this mode is useful if it is not known in advance in which channels the object is visible. In this mode, the runtime of find_shape_model can also increase significantly. The metric *'ignore_color_polarity'* can be used for images with an arbitrary number of channels. If it is used for single-channel images it has the same effect as *'ignore_local_polarity'*. It should be noted that for Metric = *'ignore_color_polarity'* the channels do not need to contain a spectral subdivision of the light (like in an RGB image). The channels can, for example, also contain images of the same object that were obtained by illuminating the object from different directions.

Note that the first two metrics (*'use_polarity'* and *'ignore_global_polarity'*) can only be selected if all Contours provide the attribute *'edge_direction'*, which defines the polarity of the edges. For more information about contour attributes like *'edge_direction'* see get_contour_attrib_xld. Otherwise, these two metrics can be selected with the operator set_shape_model_metric, which determines the polarity of the edges from an image.

MinContrast: With MinContrast, it can be determined which contrast the object edges must at least have in the recognition performed by find_shape_model. In other words, this parameter separates the object from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, MinContrast should be set to 10. If multichannel images are used for the model and the search images, and if the parameter Metric is set to *'ignore_color_polarity'* (see above) the noise in one channel must be multiplied by the square root of the number of channels to determine MinContrast. If, for example, the gray values fluctuate within a range of 10 gray levels in a single channel and the image is a three-channel image MinContrast should be set to 17. If the model should be recognized in very low contrast images, MinContrast must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, MinContrast should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the position and rotation of the model are extracted robustly and accurately by find_shape_model.

Complete pregeneration of the model

Optionally, a second value can be passed in Optimization. This value determines whether the model is pregenerated completely or not. To do so, the second value of Optimization must be set to either *'pregeneration'* or *'no_pregeneration'*. If the second value is not used (i.e., if only one value is passed), the mode that is set with set_system('pregenerate_shape_models',...) is used. With the default value (*'pregenerate_shape_models'* = *'false'*), the model is not pregenerated completely. The complete pregeneration of the model normally leads to slightly lower runtimes because the model does not need to be transformed at runtime. However, in this case, the memory requirements and the time required to create the model are significantly higher. It should also be noted that it cannot be expected that the two modes return exactly identical results because transforming the model at runtime necessarily leads to different internal data for the transformed models than pregenerating the transformed models. For example, if the model is not pregenerated completely, find_shape_model typically returns slightly lower scores, which may require setting a slightly lower value for MinScore than for a completely pregenerated model. Furthermore, the poses obtained by interpolation may differ slightly in the two modes. If maximum accuracy is desired, the pose of the model should be determined by least-squares adjustment.

If a complete pregeneration of the model is selected, the model is pregenerated for the selected angle range and stored in memory. The memory required to store the model is proportional to the number of angle steps and the number of points in the model. Hence, if AngleStep is too small or AngleExtent too big, it may happen that the model no longer fits into the (virtual) memory. In this case, either AngleStep must be enlarged or AngleExtent must be reduced. In any case, it is desirable that the model completely fits into the main memory, because this avoids paging by the operating system, and hence the time to find the object will be much smaller. Since angles can be determined with subpixel resolution by find_shape_model, AngleStep $\geq$ 1 can be selected for models of a diameter smaller than about 200 pixels.

If AngleStep = *'auto'* is selected, create_shape_model_xld automatically determines a suitable angle step length based on the size of the model. The automatically computed angle step length can be queried using get_shape_model_params.

If a complete pregeneration of the model is not selected, the model is only created in a reference pose on each pyramid level. In this case, the model must be transformed to the different angles and scales at runtime in find_shape_model. Because of this, the recognition of the model might require slightly more time.

Note that pregenerated shape models are tailored to a specific image size. For runtime reasons using images of different sizes during the search with the same model in parallel is not supported. In this case, copies of the same model must be used, otherwise the program may crash!

Attention

Note that, in contrast to the operator `create_shape_model`, it is not possible to specify a minimum size of the model components. To avoid small model components in the shape model, short contours can be eliminated before calling `create_shape_model_xld` with the operator `select_contours_xld`.

Parameters

- ▷ **`Contours`** (input_object) ..... xld_cont(-array) ⇝ *object*
  Input contours that will be used to create the model.
- ▷ **`NumLevels`** (input_control) ..... integer ⇝ *integer* / string
  Maximum number of pyramid levels.
  Default: 'auto'
  List of values: `NumLevels` ∈ {'auto', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
- ▷ **`AngleStart`** (input_control) ..... angle.rad ⇝ *real*
  Smallest rotation of the pattern.
  Default: -0.39
  Suggested values: `AngleStart` ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}
- ▷ **`AngleExtent`** (input_control) ..... angle.rad ⇝ *real*
  Extent of the rotation angles.
  Default: 0.79
  Suggested values: `AngleExtent` ∈ {6.29, 3.14, 1.57, 0.79, 0.39}
  Restriction: AngleExtent >= 0
- ▷ **`AngleStep`** (input_control) ..... angle.rad ⇝ *real* / string
  Step length of the angles (resolution).
  Default: 'auto'
  Suggested values: `AngleStep` ∈ {'auto', 0.0175, 0.0349, 0.0524, 0.0698, 0.0873}
  Restriction: AngleStep >= 0 && AngleStep <= pi / 2
- ▷ **`Optimization`** (input_control) ..... string(-array) ⇝ *string*
  Kind of optimization and optionally method used for generating the model.
  Default: 'auto'
  List of values: `Optimization` ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high', 'pregeneration', 'no_pregeneration'}
- ▷ **`Metric`** (input_control) ..... string ⇝ *string*
  Match metric.
  Default: 'ignore_local_polarity'
  List of values: `Metric` ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}
- ▷ **`MinContrast`** (input_control) ..... number ⇝ *integer*
  Minimum contrast of the objects in the search images.
  Default: 5
  Suggested values: `MinContrast` ∈ {1, 2, 3, 5, 7, 10, 20, 30, 40}
- ▷ **`ModelID`** (output_control) ..... shape_model ⇝ *handle*
  Handle of the model.

Result

If the parameters are valid, the operator `create_shape_model_xld` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised. If the parameter `NumLevels` is chosen such that the model contains too few points, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`read_contour_xld_dxf`, `edges_sub_pix`, `select_contours_xld`

Possible Successors

find_shape_model, find_shape_models, get_shape_model_params, clear_shape_model, write_shape_model, set_shape_model_origin, set_shape_model_param, set_shape_model_metric, set_shape_model_clutter

Alternatives

create_generic_shape_model

See also

set_system, get_system

Module

Matching

deserialize_shape_model (: : SerializedItemHandle : ModelID)

Deserialize a serialized shape model.

deserialize_shape_model deserializes a shape model, that was serialized by serialize_shape_model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized shape model is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created shape model with the handle ModelID.

Parameters

- ▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
  Handle of the serialized item.
- ▷ **ModelID** (output_control) ..... shape_model ⇝ *handle*
  Handle of the model.

Result

If the parameters are valid, the operator deserialize_shape_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_shape_model

Possible Successors

find_generic_shape_model

Module

Matching

determine_shape_model_params (Template : : NumLevels, AngleStart, AngleExtent, ScaleMin, ScaleMax, Optimization, Metric, Contrast, MinContrast, Parameters : ParameterName, ParameterValue)

Determine the parameters of a shape model.

determine_shape_model_params determines certain parameters of a shape model automatically from the model image Template. The parameters to be determined can be specified with Parameters. determine_shape_model_params can be used to determine the same parameters that are determined automatically when the respective parameter in create_shape_model, create_scaled_shape_model, or create_aniso_shape_model is set to *'auto'*: the number of pyramid levels (Parameters =

'num_levels'), the angle step length (Parameters = *'angle_step'*), the scale step length (Parameters = *'scale_step'* for isotropic scaling and *'scale_r_step'* and/or *'scale_c_step'* for anisotropic scaling), the kind of optimization (Parameters = *'optimization'*), the threshold (Parameters = *'contrast'*) or the hysteresis thresholds (Parameters = *'contrast_hyst'*) for the contrast, the minimum size of the object parts (Parameters = *'min_size'*), and the minimum contrast (Parameters = *'min_contrast'*). By passing a tuple of the above values in Parameters, an arbitrary combination of these parameters can be determined. If all of the above parameters should be determined, the value *'all'* can be passed. In this case both hysteresis thresholds are determined, i.e., the operator behaves like passing *'contrast_hyst'* instead of *'contrast'*.

determine_shape_model_params is mainly useful to determine the above parameters before creating the model, e.g., in an interactive system, which makes suggestions for these parameters to the user, but enables the user to modify the suggested values.

The automatically determined parameters are returned as a name-value pair in ParameterName and ParameterValue. The parameter names in ParameterName are identical to the names in Parameters, where, of course, the value *'all'* is replaced by the actual parameter names. An exception is the parameter *'contrast_hyst'*, for which the two values *'contrast_low'* and *'contrast_high'* are returned.

The remaining parameters (NumLevels, AngleStart, AngleExtent, ScaleMin, ScaleMax, Optimization, Metric, Contrast, and MinContrast) have the same meaning as in create_shape_model, create_scaled_shape_model, and create_aniso_shape_model. The description of these parameters can be looked up with these operators. These parameters are used by determine_shape_model_params to calculate the parameters to be determined in the same manner as in create_shape_model, create_scaled_shape_model, and create_aniso_shape_model. It should be noted that if the parameters of a shape model with isotropic scaling should be determined, i.e., if Parameters contains *'scale_step'* either explicitly or implicitly via *'all'*, the parameters ScaleMin and ScaleMax must contain one value each. If the parameters of a shape model with anisotropic scaling should be determined, i.e., if Parameters contains *'scale_r_step'* or *'scale_c_step'* either explicitly or implicitly, the parameters ScaleMin and ScaleMax must contain two values each. In this case, the first value of the respective parameter refers to the scaling in the row direction, while the second value refers to the scaling in the column direction.

Note that in determine_shape_model_params some parameters appear that can also be determined automatically (NumLevels, Optimization, Contrast, MinContrast). If these parameters should not be determined automatically, i.e., their name is not passed in Parameters, the corresponding parameters must contain valid values and must not be set to *'auto'*. In contrast, if these parameters are to be determined automatically, their values are treated in the following way: If the optimization or the (hysteresis) contrast is to be determined automatically, i.e., Parameters contains the value *'optimization'* or *'contrast'* (*'contrast_hyst'*), the values passed in Optimization and Contrast are ignored. In contrast, if the maximum number of pyramid levels or the minimum contrast is to be determined automatically, i.e., Parameters contains the value *'num_levels'* or *'min_contrast'*, you can let HALCON determine suitable values and at the same time specify an upper or lower boundary, respectively:

If the maximum number of pyramid levels should be specified in advance, NumLevels can be set to the particular value. If in this case Parameters contains the value *'num_levels'*, the computed number of pyramid levels is at most NumLevels. If NumLevels is set to *'auto'* (or *0* for backwards compatibility), the number of pyramid levels is determined without restrictions as large as possible.

If the minimum contrast should be specified in advance, MinContrast can be set to the particular value. If in this case Parameters contains the value *'min_contrast'*, the computed minimum contrast is at least MinContrast. If MinContrast is set to *'auto'*, the minimum contrast is determined without restrictions.

Attention

In some cases, the maximum number of pyramid levels that is returned by determine_shape_model_params is higher than the number of levels that are actually used in the shape model. The latter can be queried by using get_shape_model_params after creating the model. This might happen if the model is created by using create_scaled_shape_model or create_aniso_shape_model and ScaleMin, ScaleRMin, or ScaleCMin is below *1.0*.

Parameters

▷ **Template** (input_object) ..... (multichannel-)image ⇝ *object :* byte / uint2
Input image whose domain will be used to create the model.

▷ **NumLevels** (input_control) . . . . . integer ⇝ *integer* / string
Maximum number of pyramid levels.
Default: 'auto'
List of values: NumLevels ∈ {'auto', 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39}
Restriction: AngleExtent >= 0

▷ **ScaleMin** (input_control) . . . . . number(-array) ⇝ *real*
Minimum scale of the model.
Default: 0.9
Suggested values: ScaleMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleMin > 0

▷ **ScaleMax** (input_control) . . . . . number(-array) ⇝ *real*
Maximum scale of the model.
Default: 1.1
Suggested values: ScaleMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleMax >= ScaleMin

▷ **Optimization** (input_control) . . . . . string ⇝ *string*
Kind of optimization.
Default: 'auto'
List of values: Optimization ∈ {'auto', 'none', 'point_reduction_low', 'point_reduction_medium', 'point_reduction_high'}

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: Metric ∈ {'use_polarity', 'ignore_global_polarity', 'ignore_local_polarity', 'ignore_color_polarity'}

▷ **Contrast** (input_control) . . . . . number(-array) ⇝ *integer* / string
Threshold or hysteresis thresholds for the contrast of the object in the template image and optionally minimum size of the object parts.
Default: 'auto'
Suggested values: Contrast ∈ {'auto', 'auto_contrast', 'auto_contrast_hyst', 'auto_min_size', 10, 20, 30, 40, 60, 80, 100, 120, 140, 160}

▷ **MinContrast** (input_control) . . . . . integer ⇝ *integer* / string
Minimum contrast of the objects in the search images.
Default: 'auto'
Suggested values: MinContrast ∈ {'auto', 1, 2, 3, 5, 7, 10, 20, 30, 40}
Restriction: MinContrast < Contrast

▷ **Parameters** (input_control) . . . . . string(-array) ⇝ *string*
Parameters to be determined automatically.
Default: 'all'
List of values: Parameters ∈ {'all', 'num_levels', 'angle_step', 'scale_step', 'scale_r_step', 'scale_c_step', 'optimization', 'contrast', 'contrast_hyst', 'min_size', 'min_contrast'}

▷ **ParameterName** (output_control) . . . . . string-array ⇝ *string*
Name of the automatically determined parameter.

▷ **ParameterValue** (output_control) . . . . . number-array ⇝ *real* / integer
Value of the automatically determined parameter.

Result

If the parameters are valid, the operator determine_shape_model_params returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameters NumLevels and Contrast are chosen such that the model contains too few points, or the input image does not contain a sufficient number of significant features, the error 8510 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

draw_region, reduce_domain, threshold

Possible Successors

create_generic_shape_model

Module

Matching

```
find_aniso_shape_model ( Image : : ModelID, AngleStart,
    AngleExtent, ScaleRMin, ScaleRMax, ScaleCMin, ScaleCMax,
    MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels,
    Greediness : Row, Column, Angle, ScaleR, ScaleC, Score )
```

Find the best matches of an anisotropically scaled shape model in an image.

The operator find_aniso_shape_model finds the best NumMatches instances of the anisotropically scaled shape model ModelID in the input image Image. The model must have been created previously by calling create_aniso_shape_model or read_shape_model.

The position, rotation, and scale in the row and column direction of the found instances of the model are returned in Row, Column, Angle, ScaleR, and ScaleC. Additionally, the score of each found instance is returned in Score.

Input parameters in detail

Image and its domain: The domain of the image Image determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with create_aniso_shape_model. A different origin set with set_shape_model_origin is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than MinScore (see below). Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed with set_system('border_shape_models','true') for all models or with set_shape_model_param(ModelID,'border_shape_models','true') for a specific model, which will cause models that extend beyond the image border to be found if they achieve a score greater than MinScore. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the search will increase in this mode. Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels the model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., dilation_circle.

AngleStart, AngleExtent, ScaleRMin, ScaleRMax, ScaleCMin, ScaleCMax: The parameters AngleStart and AngleExtent determine the range of rotations for which the model is searched. The parameters ScaleRMin, ScaleRMax, ScaleCMin, and ScaleCMax determine the range of scales in the row and column directions for which the model is searched. If necessary, both ranges are clipped to the range given when the model was created with create_aniso_shape_model. In particular, this means that the angle ranges of the model and the search must overlap.

If in ModelID a model is passed that was created by using create_shape_model or create_scaled_shape_model then the model is searched with an isotropic scaling of 1.0 or with an isotropic scaling within the range from ScaleRMin to ScaleRMax, respectively. In this case, for ScaleR and ScaleC identical values are returned.

Note that in some cases instances with a rotation or scale that is slightly outside the specified range are found. This may happen if the specified range is smaller than the range given during the creation of the model. AngleStart and AngleExtent as well as ScaleRMin/ScaleCMin and ScaleRMax/ScaleCMax are checked only at the highest pyramid level. Matches that are found on the highest pyramid level are refined to the lowest pyramid level. For performance reasons, however, during the refinement it is no longer checked whether the matches are still within the specified ranges.

MinScore: The parameter MinScore determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger MinScore is chosen, the faster the search is. If the model can be expected never to be occluded in the images, MinScore may be set as high as 0.8 or even 0.9. If the matches are not tracked to the lowest pyramid level (see below) it might happen that instances with a score slightly below MinScore are found.

In case that the shape model has been extended by clutter parameters with set_shape_model_clutter and thus *'use_clutter'* is enabled, MinScore expects a second value which determines what clutter value a potential match must at most have to be regarded as an instance of the model in the image. The runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*. Changing this second value does not influence the runtime.

NumMatches: The maximum number of instances to be found can be determined with NumMatches. If more than NumMatches instances with a score greater than MinScore are found in the image, only the best NumMatches instances are returned. If fewer than NumMatches are found, only that number is returned, i.e., the parameter MinScore takes precedence over NumMatches. If all model instances exceeding MinScore in the image should be found, NumMatches must be set to *0*.

When tracking the matches through the image pyramid, on each level (except the top level), some less promising matches are rejected based on NumMatches. Thus, it is possible that some matches are rejected that would have had a higher score on the lowest pyramid level. Due to this, for example, the found match for NumMatches set to *1* might be different from the match with the highest score returned when setting NumMatches to *0* or > *1*.

If multiple objects with a similar score are expected, but only the one with the highest score should be returned, it might be preferable to raise NumMatches, and then select the match with the highest score.

In case that the shape model has been extended by clutter parameters using set_shape_model_clutter, NumMatches also considers the second value passed in MinScore: If more than NumMatches instances with a score greater than the first entry of MinScore and a clutter score smaller than the second entry of MinScore are found in the image, only the best NumMatches instances with respect to clutter are returned. Still, MinScore takes precedence over NumMatches and NumMatches must be set to *0* if all model instances fulfilling the conditions imposed by MinScore should be found. Please note that using clutter parameters, when tracking the matches through the image pyramid, no matches are rejected. Thus the runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*.

MaxOverlap: If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter MaxOverlap determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than MaxOverlap only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see smallest_rectangle2) of the found instances. If MaxOverlap $= 0$, the found instances may not overlap at all, while for MaxOverlap $= 1$ all instances are returned.

SubPixel: The parameter SubPixel determines whether the instances should be extracted with subpixel accuracy. If SubPixel is set to *'none'* (or *'false'* for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle and scale resolution that was specified with create_aniso_shape_model. If SubPixel is set to *'interpolation'* (or *'true'*) the position as well as the rotation and scale are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast to *'interpolation'*, this mode requires additional computation time. The different modes for least-squares adjustment (*'least_squares'*, *'least_squares_high'*, and *'least_squares_very_high'*) can be used to determine the accuracy with which the minimum distance is being searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, SubPixel should be set to *'interpolation'*. If least-squares adjustment

is desired, *'least_squares'* should be chosen because this results in the best trade-off between runtime and accuracy.

Objects that are slightly deformed with respect to the model, in some cases cannot be found or are found but only with a low accuracy. For such objects it is possible to additionally pass a maximal allowable object deformation in the parameter SubPixel. The deformation must be specified in pixels. This can be done by passing the optional parameter value *'max_deformation '* followed by an integer value between *0* and *32* (in the same string), which specifies the maximum deformation. For example, if the shape of the object may be deformed by up to 2 pixels with respect to the shape that is stored in the model, the value *'max_deformation 2'* must be passed in SubPixel in addition to the above described mode for the subpixel extraction, i.e., for example *['least_squares', 'max_deformation 2']*. Passing the value *'max_deformation 0'* corresponds to a search without allowing deformations, i.e., the behavior is the same as if no *'max_deformation '* is passed. Note that higher values for the maximum deformation often result in an increased runtime. Furthermore, the higher the deformation value is chosen, the higher is the risk of finding wrong model instances. Both problems mainly arise when searching for small objects or for objects with fine structures. This is because such kinds of objects for higher deformations lose their characteristic shape, which is important for a robust search. Also note that for higher deformations the accuracy of partially occluded objects might decrease if clutter is present close to the object. Consequently, the maximum deformation should be chosen as small as possible and only as high as necessary. Approximately rotationally symmetric objects may not be found if *'max_deformation'* and AngleExtent are both set to a value greater than 0. In that case, ambiguities may occur that cannot be resolved, and the match is rejected as false. If this happens, try to set either *'max_deformation'* or AngleExtent to 0, or adjust the model such that symmetries are reduced. When specifying a deformation higher than *0* the computation of the score depends on the chosen value for the subpixel extraction. In most cases, the score of a match changes if *'least_squares'*, *'least_squares_high'*, or *'least_squares_very_high'* (see above) is chosen for the subpixel extraction (in comparison to *'none'* or *'interpolation'*). Furthermore, if one of the least-squares adjustments is selected the score might increase when increasing the maximum deformation because then for the model points more corresponding image points can be found. To get a meaningful score value and to avoid erroneous matches, we recommend to always combine the allowance of a deformation with a least-squares adjustment.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the shape model was created with create_aniso_shape_model. If NumLevels is set to *0*, the number of pyramid levels specified in create_aniso_shape_model is used.

In certain cases, the number of pyramid levels that was determined automatically with, for example, create_aniso_shape_model may be too high. The consequence may be that some matches that may have a high final score are rejected on the highest pyramid level and thus are not found. Instead of setting MinScore to a very low value to find all matches, it may be better to query the value of NumLevels with get_shape_model_params and then use a slightly lower value in find_aniso_shape_model. This approach is often better regarding the speed and robustness of the matching.

Optionally, NumLevels can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of *[4,2]* for NumLevels means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, SubPixel should be set to at least *'least_squares'*. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

In input images of poor quality, i.e., in images that are, e.g., defocused, deformed, or noisy, often no instances of the shape model can be found on the lowest pyramid level. The reason for this behavior is the missing or deformed edge information which is a result of the poor image quality. Nevertheless, the edge information may be sufficient on higher pyramid levels. But keep in mind the above mentioned restrictions on accuracy and robustness if instances that were found on higher pyramid levels are used. The selection of the suitable pyramid level, i.e., the lowest pyramid level on which at least one instance of the shape model can be found, depends on the model and on the input image. This pyramid level may vary from image to image. To facilitate the matching on images of poor quality, the lowest pyramid level on which at least one instance of the model can be found can be determined automatically during the matching. To activate this mechanism,

i.e., to use the so-called 'increased tolerance mode', the lowest pyramid level must be specified negatively in NumLevels. If, e.g., NumLevels is set to *[4,-2]*, the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level. This means that an instance of the shape model is searched on the pyramid level 2. If no instance of the model can be found on this pyramid level, the lowest pyramid level is determined on which at least one instance of the model can be found. The instances of this pyramid level will then be returned.

If the ModelID was adapted with adapt_shape_model_high_noise the estimated lowest pyramid level will be used by default. However, the user can override the estimated lowest pyramid level by providing two values to NumLevels and explicitly setting the lowest pyramid level.

Greediness: The parameter Greediness determines how "greedily" the search should be carried out. If Greediness $= 0$, a safe search heuristic is used, which always finds the model if it is visible in the image and the other parameters are set appropriately. However, the search will be relatively time consuming in this case. If Greediness $= 1$, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For Greediness $= 1$, the maximum search speed is achieved. In almost all cases, the shape model will always be found for Greediness $= 0.9$.

Output parameters in detail

Row, Column, Angle, ScaleR, ScaleC: The position, rotation, and scale in the row and column direction of the found instances of the model are returned in Row, Column, Angle, ScaleR, and ScaleC. The coordinates Row and Column are related to the position of the origin of the shape model in the search image. However, Row and Column do not exactly correspond to this position. Instead, find_aniso_shape_model returns slightly modified values that are optimized for creating a transformation matrix, that can be used for alignment or visualization of the model contours. (This has to do with the way HALCON transforms iconic objects, see affine_trans_pixel). The example below shows how to create the transformation matrix for alignment of the found matches and how to use it to display them in the search image.

By default, the model origin is the center of gravity of the domain (region) of the image that was used to create the shape model with create_aniso_shape_model. A different origin can be set with set_shape_model_origin.

Score: The score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

In case that the shape model has been extended by clutter parameters using set_shape_model_clutter, following the above values Score also returns the clutter scores of each found instance. If, for example, half of the clutter region is filled by clutter edges, the clutter score will equal 0.5. If, e.g., two instances are found, the score is 0.9 for the first instance and 0.8 for the second instance, and the clutter score is 0.2 for the first instance and 0.1 for the second instance, Score $= [0.9, 0.8, 0.2, 0.1]$ is returned. Please note that of all shape-based matching results, clutter scores are affected the most when a variation of illumination occurs.

Specifying a timeout

Using the operator set_shape_model_param you can specify a *'timeout'* for find_aniso_shape_model. If find_aniso_shape_model reaches this *'timeout'*, it terminates without results and returns the error code 9400 (H_ERR_TIMEOUT). Depending on the scaling ranges specified by ScaleRMin, ScaleRMax, ScaleCMin, and ScaleCMax, find_aniso_shape_model needs a significant amount of time to free cached transformations if the shape model is not pregenerated. As this transformations have to be freed after a timeout occurs, the runtime of find_aniso_shape_model exceeds the value of the specified *'timeout'* by this time.

Visualization of the results

To display the results found by shape-based matching, we highly recommend the usage of the procedure dev_display_shape_matching_results.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image in which the model should be found.

▷ **ModelID** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the model.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **ScaleRMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the model in the row direction.
Default: 0.9
Suggested values: ScaleRMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the model in the row direction.
Default: 1.1
Suggested values: ScaleRMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleRMax >= ScaleRMin

▷ **ScaleCMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the model in the column direction.
Default: 0.9
Suggested values: ScaleCMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the model in the column direction.
Default: 1.1
Suggested values: ScaleCMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleCMax >= ScaleCMin

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the model to be found.
Default: 0.5
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer ⇝ *integer*
Number of instances of the model to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real ⇝ *real*
Maximum overlap of the instances of the model to be found.
Default: 0.5
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **SubPixel** (input_control) . . . . . string(-array) ⇝ *string*
Subpixel accuracy if not equal to *'none'*.
Default: 'least_squares'
Suggested values: SubPixel ∈ {'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high', 'max_deformation 1', 'max_deformation 2', 'max_deformation 3', 'max_deformation 4', 'max_deformation 5', 'max_deformation 6'}

▷ **NumLevels** (input_control) ..... integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Greediness** (input_control) ..... real ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **Row** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the found instances of the model.

▷ **Column** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the found instances of the model.

▷ **Angle** (output_control) ..... angle.rad-array ⇝ *real*
Rotation angle of the found instances of the model.

▷ **ScaleR** (output_control) ..... number-array ⇝ *real*
Scale of the found instances of the model in the row direction.

▷ **ScaleC** (output_control) ..... number-array ⇝ *real*
Scale of the found instances of the model in the column direction.

▷ **Score** (output_control) ..... real-array ⇝ *real*
Score of the found instances of the model.

Example

```
create_aniso_shape_model (ImageReduced, 0, rad(-15), rad(30), 0, \
                          0.9, 1.1, 0, 0.9, 1.1, 0, 'none', \
                          'use_polarity', 30, 10, ModelID)
get_shape_model_contours (ModelXLD, ModelID, 1)
find_aniso_shape_model (SearchImage, ModelID, rad(-15), rad(30), \
                        0.9, 1.1, 0.9, 1.1, 0.5, 1, 0.5, 'interpolation', \
                        0, 0, Row, Column, Angle, ScaleR, ScaleC, Score)
* Create transformation matrix
hom_mat2d_identity (HomMat2DIdentity)
hom_mat2d_scale (HomMat2DIdentity, ScaleR, ScaleC, 0, 0, HomMat2DScale)
hom_mat2d_rotate (HomMat2DScale, Angle, 0, 0, HomMat2DRotate)
hom_mat2d_translate (HomMat2DRotate, Row, Column, HomMat2DObject)
* Calculate true position of the model origin in the search image
affine_trans_pixel (HomMat2DObject, 0, 0, RowObject, ColObject)
* Display results
dev_display_shape_matching_results (ModelID, 'red', Row, Column, Angle, \
                                    ScaleR, ScaleC, 0)
```

Result

If the parameter values are correct, the operator find_aniso_shape_model returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

create_aniso_shape_model, read_shape_model, set_shape_model_origin, set_shape_model_clutter

Possible Successors

clear_shape_model

Alternatives

find_generic_shape_model

See also

set_system, get_system

Module

Matching

find_aniso_shape_models (Image : : ModelIDs, AngleStart, AngleExtent, ScaleRMin, ScaleRMax, ScaleCMin, ScaleCMax, MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels, Greediness : Row, Column, Angle, ScaleR, ScaleC, Score, Model)

Find the best matches of multiple anisotropically scaled shape models.

The operator find_aniso_shape_models finds the best NumMatches instances of the anisotropically scaled shape models that are passed in ModelIDs in the input image Image. The models must have been created previously by calling create_aniso_shape_model or read_shape_model. In contrast to find_aniso_shape_model, multiple models can be searched in the same image in one call.

The position, rotation, and scale in the row and column direction of the found instances of the model are returned in Row, Column, Angle, ScaleR, and ScaleC. The score of each found instance is returned in Score. The type of the found instances of the models is returned in Model. For details see respective sections below.

Characteristics of the parameter semantics

Compared to find_aniso_shape_model, the semantics of all input parameters have changed to some extent. All input parameters must either contain one element, in which case the parameter is used for all models, or must contain the same number of elements as ModelIDs, in which case each parameter element refers to the corresponding element in ModelIDs. (NumLevels may also contain either two or twice the number of elements as ModelIDs.) More details can be found below in the sections containing information for the respective parameters. Note that a call to find_aniso_shape_models with multiple values for ModelIDs, NumMatches and MaxOverlap has the same effect as multiple independent calls to find_aniso_shape_model with the respective parameters. However, a single call to find_aniso_shape_models is considerably more efficient.

Input parameters in detail

Image and its domain: The domain of the Image determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with create_aniso_shape_model. A different origin set with set_shape_model_origin is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than MinScore (see below). Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed with set_system('border_shape_models','true') for all models or with set_shape_model_param(ModelID,'border_shape_models','true') for a specific model, which will cause models that extend beyond the image border to be found if they achieve a score greater than MinScore. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the search will increase in this mode. When searching multiple models *'border_shape_models'* is treated as *'true'* for all models even if *'border_shape_models'* only evaluates to *'true'* for one of the models in a search. Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels the model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., dilation_circle.

As usual, the domain of the input Image is used to restrict the search space for the reference point of the models ModelIDs. Consistent with the above semantics, the input Image can therefore contain a single image object or an image object tuple containing multiple image objects. If Image contains a single image

object, its domain is used as the region of interest for all models in ModelIDs. If Image contains multiple image objects, each domain is used as the region of interest for the corresponding model in ModelIDs. In this case, the images have to be identical except for their domains, i.e., their pointers have to be identical (the pointers can be checked using get_image_pointer1). As a consequence, Image cannot be constructed in an arbitrary manner using concat_obj, but must be created from the same image using add_channels or equivalent calls. If this is not the case, an error message is returned.

AngleStart, AngleExtent, ScaleRMin, ScaleRMax, ScaleCMin, and ScaleCMax: The parameters AngleStart and AngleExtent determine the range of rotations for which the model is searched. The parameters ScaleRMin, ScaleRMax, ScaleCMin, and ScaleCMax determine the range of scales in the row and column directions for which the model is searched. If necessary, both ranges are clipped to the range given when the model was created with create_aniso_shape_model. In particular, this means that the angle ranges of the model and the search must overlap.

If in ModelIDs a model is passed that was created by using create_shape_model or create_scaled_shape_model then the model is searched with an isotropic scaling of 1.0 or with an isotropic scaling within the range from ScaleRMin to ScaleRMax, respectively. In this case, for ScaleR and ScaleC identical values are returned.

Note that in some cases instances with a rotation or scale that is slightly outside the specified range are found. This may happen if the specified range is smaller than the range given during the creation of the model. AngleStart and AngleExtent as well as ScaleRMin/ScaleCMin and ScaleRMax/ScaleCMax are checked only at the highest pyramid level. Matches that are found on the highest pyramid level are refined to the lowest pyramid level. For performance reasons, however, during the refinement it is no longer checked whether the matches are still within the specified ranges.

MinScore: The parameter MinScore determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger MinScore is chosen, the faster the search is. If the model can be expected never to be occluded in the images, MinScore may be set as high as 0.8 or even 0.9. If the matches are not tracked to the lowest pyramid level (see below) it might happen that instances with a score slightly below MinScore are found.

If a single value is passed in MinScore, this value is applied to found instances of all models. If, on the other hand, MinScore contains multiple values, the values are applied separately for the respective model.

In case that the shape models have been extended by clutter parameters with set_shape_model_clutter and thus *'use_clutter'* is enabled, MinScore expects for each minimum score an additional value which determines what clutter score a potential match must at most have to be regarded as an instance of the model in the image. The runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*. Changing this second value does not influence the runtime. Note that the different shape models must have the same value for *'use_clutter'*.

If the maximum clutter is specified separately for each model, which is needed if also the minimum score is set for each model individually, MinScore must contain twice the number of elements as ModelIDs. In this case, the minimum score and the maximum clutter must be specified interleaved in MinScore. If, for example, two models are specified in ModelIDs, the minimum score is 0.9 for the first model and 0.8 for the second model, and the maximum clutter is 0.1 for the first model and 0.2 for the second model, MinScore $= [0.9, 0.1, 0.8, 0.2]$ must be selected.

NumMatches: The maximum number of instances to be found can be determined with NumMatches. If more than NumMatches instances with a score greater than MinScore are found in the image, only the best NumMatches instances are returned. If fewer than NumMatches are found, only that number is returned, i.e., the parameter MinScore takes precedence over NumMatches. If all model instances exceeding MinScore in the image should be found, NumMatches must be set to *0*.

When tracking the matches through the image pyramid, on each level (except the top level), some less promising matches are rejected based on NumMatches. Thus, it is possible that some matches are rejected that would have had a higher score on the lowest pyramid level. Due to this, for example, the found match for NumMatches set to *1* might be different from the match with the highest score returned when setting NumMatches to *0* or > *1*.

If multiple objects with a similar score are expected, but only the one with the highest score should be returned, it might be preferable to raise NumMatches, and then select the match with the highest score.

In case that the shape models have been extended by clutter parameters using set_shape_model_clutter, NumMatches also considers the second value passed in MinScore: If more than NumMatches instances with a score greater than the first entry of MinScore and a clutter score smaller than the second entry of MinScore are found in the image, only the best NumMatches

instances with respect to clutter are returned. Still, MinScore takes precedence over NumMatches and NumMatches must be set to *0* if all model instances fulfilling the conditions imposed by MinScore should be found. Please note that using clutter parameters, when tracking the matches through the image pyramid, no matches are rejected. Thus the runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*.

If NumMatches contains one element, find_aniso_shape_models returns the best NumMatches instances of the model irrespective of the type of the model. If, for example, two models are passed in ModelIDs and NumMatches $= 2$ is selected, it can happen that two instances of the first model and no instances of the second model, one instance of the first model and one instance of the second model, or no instances of the first model and two instances of the second model are returned. If, on the other hand, NumMatches contains multiple values, the number of instances returned of the different models corresponds to the number specified in the respective entry in NumMatches. If, for example, NumMatches $= [1, 1]$ is selected, one instance of the first model and one instance of the second model is returned.

MaxOverlap: If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter MaxOverlap determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than MaxOverlap only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see smallest_rectangle2) of the found instances. If MaxOverlap $= 0$, the found instances may not overlap at all, while for MaxOverlap $= 1$ all instances are returned.

If a single value is passed in MaxOverlap, the overlap is computed for all found instances of the different models, irrespective of the model type, i.e., instances of the same or of different models that overlap too much are eliminated. If, on the other hand, multiple values are passed in MaxOverlap, the overlap is only computed for found instances of the model that have the same model type, i.e., only instances of the same model that overlap too much are eliminated. In this mode, models of different types may overlap completely.

SubPixel: The parameter SubPixel determines whether the instances should be extracted with subpixel accuracy. If SubPixel is set to *'none'* (or *'false'* for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle and scale resolution that was specified with create_aniso_shape_model. If SubPixel is set to *'interpolation'* (or *'true'*) the position as well as the rotation and scale are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast to *'interpolation'*, this mode requires additional computation time. The different modes for least-squares adjustment (*'least_squares'*, *'least_squares_high'*, and *'least_squares_very_high'*) can be used to determine the accuracy with which the minimum distance is being searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, SubPixel should be set to *'interpolation'*. If least-squares adjustment is desired, *'least_squares'* should be chosen because this results in the best trade-off between runtime and accuracy.

Objects that are slightly deformed with respect to the model, in some cases cannot be found or are found but only with a low accuracy. For such objects it is possible to additionally pass a maximal allowable object deformation in the parameter SubPixel. The deformation must be specified in pixels. This can be done by passing the optional parameter value *'max_deformation '* followed by an integer value between *0* and *32* (in the same string), which specifies the maximum deformation. For example, if the shape of the object may be deformed by up to 2 pixels with respect to the shape that is stored in the model, the value *'max_deformation 2'* must be passed in SubPixel in addition to the above described mode for the subpixel extraction, i.e., for example *['least_squares', 'max_deformation 2']*. Passing the value *'max_deformation 0'* corresponds to a search without allowing deformations, i.e., the behavior is the same as if no *'max_deformation '* is passed. Note that higher values for the maximum deformation often result in an increased runtime. Furthermore, the higher the deformation value is chosen, the higher is the risk of finding wrong model instances. Both problems mainly arise when searching for small objects or for objects with fine structures. This is because such kinds of objects for higher deformations lose their characteristic shape, which is important for a robust search. Also note that for higher deformations the accuracy of partially occluded objects might decrease if clutter is present close to the object. Consequently, the maximum deformation should be chosen as small as possible and only as high as necessary. Approximately rotationally symmetric objects may not be found if *'max_deformation'* and AngleExtent are both set to a value greater than 0. In that case, ambiguities may occur that cannot be resolved, and the match is rejected as false. If this happens, try to set either

'max_deformation' or AngleExtent to 0, or adjust the model such that symmetries are reduced. When specifying a deformation higher than *0* the computation of the score depends on the chosen value for the subpixel extraction. In most cases, the score of a match changes if *'least_squares'*, *'least_squares_high'*, or *'least_squares_very_high'* (see above) is chosen for the subpixel extraction (in comparison to *'none'* or *'interpolation'*). Furthermore, if one of the least-squares adjustments is selected the score might increase when increasing the maximum deformation because then for the model points more corresponding image points can be found. To get a meaningful score value and to avoid erroneous matches, we recommend to always combine the allowance of a deformation with a least-squares adjustment.

If the subpixel extraction and/or the maximum object deformation is specified separately for each model, for each model passed in ModelIDs exactly one value for the subpixel extraction must be passed in SubPixel. After each value for the subpixel extraction optionally a second value can be passed, which describes the maximum object deformation of the corresponding mode. If for a certain model no value for the maximum object deformation is passed, the model is searched without taking deformations into account. For example, if two models are passed in ModelIDs and for the first model the subpixel extraction is set to *'interpolation'* and no object deformations are allowed and for the second model the subpixel extraction is set to *'least_squares'* and a maximum object deformation of *3* pixels is allowed, then the tuple *['interpolation', 'least_squares', 'max_deformation 3']* must be passed in SubPixel. Alternatively, the equivalent tuple *['interpolation', 'max_deformation 0', 'least_squares', 'max_deformation 3']* may be passed.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the shape model was created with create_aniso_shape_model. If NumLevels is set to *0*, the number of pyramid levels specified in create_aniso_shape_model is used.

In certain cases, the number of pyramid levels that was determined automatically with, for example, create_aniso_shape_model may be too high. The consequence may be that some matches that may have a high final score are rejected on the highest pyramid level and thus are not found. Instead of setting MinScore to a very low value to find all matches, it may be better to query the value of NumLevels with get_shape_model_params and then use a slightly lower value in find_aniso_shape_models. This approach is often better regarding the speed and robustness of the matching.

Optionally, NumLevels can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of *[4,2]* for NumLevels means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, SubPixel should be set to at least *'least_squares'*. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

If the lowest pyramid level is specified separately for each model, NumLevels must contain twice the number of elements as ModelIDs. In this case, the number of pyramid levels and the lowest pyramid level must be specified interleaved in NumLevels. If, for example, two models are specified in ModelIDs, the number of pyramid levels is 5 for the first model and 4 for the second model, and the lowest pyramid level is 2 for the first model and 1 for the second model, NumLevels $= [5, 2, 4, 1]$ must be selected. If exactly two models are specified in ModelIDs, a special case occurs. If in this case the lowest pyramid level is to be specified, the number of pyramid levels and the lowest pyramid level must be specified explicitly for both models, even if they are identical, because specifying two values in NumLevels is interpreted as the explicit specification of the number of pyramid levels for the two models.

In input images of poor quality, i.e., in images that are, e.g., defocused, deformed, or noisy, often no instances of the shape model can be found on the lowest pyramid level. The reason for this behavior is the missing or deformed edge information which is a result of the poor image quality. Nevertheless, the edge information may be sufficient on higher pyramid levels. But keep in mind the above mentioned restrictions on accuracy and robustness if instances that were found on higher pyramid levels are used. The selection of the suitable pyramid level, i.e., the lowest pyramid level on which at least one instance of the shape model can be found, depends on the model and on the input image. This pyramid level may vary from image to image. To facilitate the matching on images of poor quality, the lowest pyramid level on which at least one instance of the model can be found can be determined automatically during the matching. To activate this mechanism, i.e., to use the so-called 'increased tolerance mode', the lowest pyramid level must be specified negatively

in NumLevels. If, e.g., NumLevels is set to *[5,2,4,-1]*, the lowest pyramid level for the first model is set to 2. If no instance of the first model can be found on this pyramid level, no result will be returned for this model. For the second shape model, the lowest pyramid level is set to *-1*. Therefore, an instance of the shape model is searched on the pyramid level 1. If no instance of the second model can be found on this pyramid level, the lowest pyramid level is determined on which at least one instance of the model can be found. The instances of this pyramid level will then be returned.

If a model was adapted with adapt_shape_model_high_noise the estimated lowest pyramid level will be used by default. However, the user can override the estimated lowest pyramid level by providing explicitly the lowest pyramid level as described above.

Greediness: The parameter Greediness determines how "greedily" the search should be carried out. If Greediness $= 0$, a safe search heuristic is used, which always finds the model if it is visible in the image and the other parameters are set appropriately. However, the search will be relatively time consuming in this case. If Greediness $= 1$, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For Greediness $= 1$, the maximum search speed is achieved. In almost all cases, the shape model will always be found for Greediness $= 0.9$.

Output parameters in detail

Row, Column, Angle, ScaleR, ScaleC: The position, rotation, and scale in the row and column direction of the found instances of the model are returned in Row, Column, Angle, ScaleR, and ScaleC. The coordinates Row and Column are the coordinates of the origin of the shape model in the search image. By default, the origin is the center of gravity of the domain (region) of the image that was used to create the shape model with create_aniso_shape_model. A different origin can be set with set_shape_model_origin.

Note that the coordinates Row and Column do not exactly correspond to the position of the model in the search image. Thus, you cannot directly use them. Instead, the values are optimized for creating the transformation matrix with which you can use the results of the matching for various tasks, e.g., to align ROIs for other processing steps. The example given for find_aniso_shape_model shows how to create this matrix and use it to display the model at the found position in the search image.

Note also that for visualizing the model at the found position, also the procedure dev_display_shape_matching_results can be used.

Score: The score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

In case that the shape models have been extended by clutter parameters using set_shape_model_clutter, following the above values Score also returns the clutter scores of each found instance. If, for example, half of the clutter region is filled by clutter edges, the clutter score will equal 0.5. If, e.g., two instances are found, the score is 0.9 for the first instance and 0.8 for the second instance, and the clutter score is 0.2 for the first instance and 0.1 for the second instance, Score $= [0.9, 0.8, 0.2, 0.1]$ is returned. Please note that of all shape-based matching results, clutter scores are affected the most when a variation of illumination occurs.

Model: The type of the found instances of the models is returned in Model. The elements of Model are indices into the tuple ModelIDs, i.e., they can contain values from 0 to $|\text{ModelIDs}| - 1$. Hence, a value of 0 in an element of Model corresponds to an instance of the first model in ModelIDs.

Specifying a timeout

Using the operator set_shape_model_param you can specify a *'timeout'* for find_aniso_shape_models. If the shape models referenced by ModelIDs hold different values for *'timeout'*, find_aniso_shape_models uses the smallest one. If find_aniso_shape_models reaches this *'timeout'*, it terminates without results and returns the error code 9400 (H_ERR_TIMEOUT). Depending on the scaling ranges specified by ScaleRMin, ScaleRMax, ScaleCMin and ScaleCMax, find_aniso_shape_models needs a significant amount of time to free cached transformations if the shape model is not pregenerated. As this transformations have to be freed after a timeout occurs, the runtime of find_aniso_shape_models exceeds the value of the specified *'timeout'* by this time.

MinContrast with multiple models

Please note, that the different models that are given with the parameter ModelIDs should have been created with the same value of MinContrast. If they were created with different values for MinContrast, find_aniso_shape_models will use the smallest of these values.

Visualization of the results

To display the results found by shape-based matching, we highly recommend the usage of the procedure `dev_display_shape_matching_results`.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Input image in which the models should be found.

▷ **ModelIDs** (input_control) . . . . . shape_model(-array) ⇝ *handle*
Handle of the models.

▷ **AngleStart** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Smallest rotation of the models.
Default: -0.39
Suggested values: `AngleStart` ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: `AngleExtent` ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **ScaleRMin** (input_control) . . . . . number(-array) ⇝ *real*
Minimum scale of the models in the row direction.
Default: 0.9
Suggested values: `ScaleRMin` ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleRMin > 0

▷ **ScaleRMax** (input_control) . . . . . number(-array) ⇝ *real*
Maximum scale of the models in the row direction.
Default: 1.1
Suggested values: `ScaleRMax` ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleRMax >= ScaleRMin

▷ **ScaleCMin** (input_control) . . . . . number(-array) ⇝ *real*
Minimum scale of the models in the column direction.
Default: 0.9
Suggested values: `ScaleCMin` ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleCMin > 0

▷ **ScaleCMax** (input_control) . . . . . number(-array) ⇝ *real*
Maximum scale of the models in the column direction.
Default: 1.1
Suggested values: `ScaleCMax` ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleCMax >= ScaleCMin

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the models to be found.
Default: 0.5
Suggested values: `MinScore` ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: $0 \leq$ `MinScore` ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of instances of the models to be found (or 0 for all matches).
Default: 1
Suggested values: `NumMatches` ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) ..... real(-array) ⇝ *real*
Maximum overlap of the instances of the models to be found.
Default: 0.5
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **SubPixel** (input_control) ..... string(-array) ⇝ *string*
Subpixel accuracy if not equal to *'none'*.
Default: 'least_squares'
Suggested values: SubPixel ∈ {'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high', 'max_deformation 1', 'max_deformation 2', 'max_deformation 3', 'max_deformation 4', 'max_deformation 5', 'max_deformation 6'}

▷ **NumLevels** (input_control) ..... integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Greediness** (input_control) ..... real(-array) ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **Row** (output_control) ..... point.y-array ⇝ *real*
Row coordinate of the found instances of the models.

▷ **Column** (output_control) ..... point.x-array ⇝ *real*
Column coordinate of the found instances of the models.

▷ **Angle** (output_control) ..... angle.rad-array ⇝ *real*
Rotation angle of the found instances of the models.

▷ **ScaleR** (output_control) ..... number-array ⇝ *real*
Scale of the found instances of the models in the row direction.

▷ **ScaleC** (output_control) ..... number-array ⇝ *real*
Scale of the found instances of the models in the column direction.

▷ **Score** (output_control) ..... real-array ⇝ *real*
Score of the found instances of the models.

▷ **Model** (output_control) ..... integer-array ⇝ *integer*
Index of the found instances of the models.

Example

```
read_image (Image, 'pcb_focus/pcb_focus_telecentric_061')
gen_rectangle1 (ROI_0, 236, 241, 313, 321)
gen_circle (ROI_1, 281, 653, 41)
reduce_domain (Image, ROI_0, ImageReduced1)
reduce_domain (Image, ROI_1, ImageReduced2)
create_aniso_shape_model (ImageReduced1,'auto', -0.39, 0.79, 'auto', 0.9, \
                          1.1, 'auto', 0.9, 1.1, 'auto', 'auto', \
                          'use_polarity', 'auto', 'auto', ModelID1)
create_aniso_shape_model (ImageReduced2,'auto', -0.39, 0.79, 'auto', 0.9, \
                          1.1, 'auto', 0.9, 1.1, 'auto', 'auto', \
                          'use_polarity', 'auto', 'auto', ModelID2)
ModelIDs:=[ModelID1, ModelID2]
find_aniso_shape_models(Image, ModelIDs, -0.39, 0.79, 0.9, 1.1, 0.9, 1.1, \
                        0.5, 1, 0.5, 'least_squares', 0, 0.9, Row, Column, \
                        Angle, ScaleR, ScaleC, Score, Model)
* Display results
dev_display_shape_matching_results (ModelIDs, 'red', Row, Column, Angle, \
                                    ScaleR, ScaleC, Model)
```

Result

If the parameter values are correct, the operator `find_aniso_shape_models` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input images are available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`add_channels`, `create_aniso_shape_model`, `read_shape_model`, `set_shape_model_origin`, `set_shape_model_clutter`

Possible Successors

`clear_shape_model`

Alternatives

`find_generic_shape_model`

See also

`set_system`, `get_system`, `set_shape_model_param`

Module

Matching

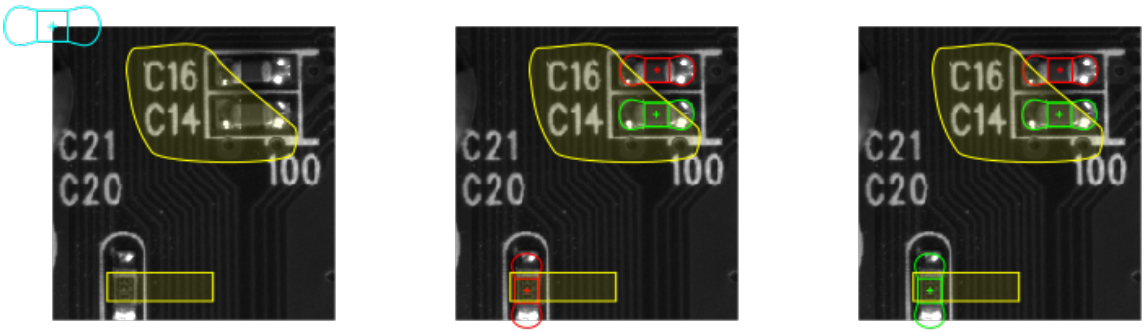
find_generic_shape_model (SearchImage : : ModelID : MatchResultID, NumMatchResult)

Find the best matches of one or multiple shape models in an image.

The operator `find_generic_shape_model` finds the best instances of one or multiple shape models passed in `ModelID` in the input image `SearchImage`. The found matches are returned in `MatchResultID` and can be queried using `get_generic_shape_model_result` and `get_generic_shape_model_result_object`. The number of found matches is returned in `NumMatchResult`. The model can be parameterized using `set_generic_shape_model_param` and `set_generic_shape_model_object`, through which one can control the search.

The input image `SearchImage` may be a multichannel image. Which channels of a multichannel image are used depends on the model parameter *'metric'* (see `set_generic_shape_model_param`). The domain of `SearchImage` sets the ROI for the search. It limits the search space as it is considered as boundary for the center of gravity of `ModelID` in order to accelerate the matching process. For information on passing a tuple of images to `SearchImage` see below. Furthermore, the search space is limited by the size of `SearchImage`. By default, `ModelID` is only searched within those points of the domain, in which `ModelID` fits completely. Hence, `ModelID` will not be found if it exceeds the border of the image. This holds for all levels of the used image pyramid.

In rare cases which typically occur for artificial images, instances of `ModelID` cannot be found in an image, if they touch the border of the domain on any level of the image pyramid. As a rule of thumb, `ModelID` might not be found if its distance to an image border is smaller than $2^{NumLevels-1}$ pixels (where $NumLevels$ is the number of pyramid levels). This behavior can be changed with *'border_shape_models'*, see `set_generic_shape_model_param`. When searching multiple models *'border_shape_models'* is treated as *'true'* for all models even if *'border_shape_models'* only evaluates to *'true'* for one of the models in a search.



(1) (2) (3)

(1) Search image with reduced domain (yellow) and the shape model (cyan). (2) Instances that will be returned (green) or not returned (red) in case *'border_shape_models'* is set to *'false'*. (3) Instances that will be returned (green) or not returned (red) in case *'border_shape_models'* is set to *'true'*.

When searching multiple models, the search space can be restricted for all models simultaneously by passing a single image with reduced domain in `SearchImage`.

Alternatively, the search space can be restricted for each model individually by passing an object containing multiple image objects, one for each model in `ModelID`. The search space is determined by the domain of the corresponding image. Except the domain, the images have to be identical, i.e., the pointers of the image objects need to refer to the same image (thus they need to be identical). The pointers can be checked using `get_image_pointer1`.

When applying multiple models, different identifiers (*'model_identifier'*) have to be set for each model, so that each instance can be assigned to the shape model it has been found with. Otherwise an exception is raised.

Parameters

▷ **`SearchImage`** (input_object) . . . . . (multichannel-)object(-array) ⇝ *object*
Image in which the model is searched.

▷ **`ModelID`** (input_control) . . . . . shape_model(-array) ⇝ *handle*
Handle of the shape model.

▷ **`MatchResultID`** (output_control) . . . . . generic_shape_model_result ⇝ *handle*
Handle with the found matches.

▷ **`NumMatchResult`** (output_control) . . . . . integer ⇝ *integer*
Number of found matches.

Result

If the parameters are valid, the operator `find_generic_shape_model` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`train_generic_shape_model`, `set_generic_shape_model_param`

Module

Matching

```
find_scaled_shape_model ( Image : : ModelID, AngleStart,
    AngleExtent, ScaleMin, ScaleMax, MinScore, NumMatches,
    MaxOverlap, SubPixel, NumLevels, Greediness : Row, Column,
    Angle, Scale, Score )
```

Find the best matches of an isotropically scaled shape model in an image.

The operator `find_scaled_shape_model` finds the best `NumMatches` instances of the isotropically scaled shape model `ModelID` in the input image `Image`. The model must have been created previously by calling `create_scaled_shape_model` or `read_shape_model`.

The position, rotation, and scale of the found instances of the model are returned in `Row`, `Column`, `Angle`, and `Scale`. The score of each found instance is returned in `Score`.

Input parameters in detail

`Image` and its domain: The domain of the image `Image` determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with `create_scaled_shape_model`. A different origin set with `set_shape_model_origin` is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than `MinScore` (see below). Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed with `set_system('border_shape_models','true')` for all models or with `set_shape_model_param(ModelID,'border_shape_models','true')` for a specific model, which will cause models that extend beyond the image border to be found if they achieve a score greater than `MinScore`. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the search will increase in this mode. Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels the model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., `dilation_circle`.

`AngleStart`, `AngleExtent`, `ScaleMin`, `ScaleMax`: The parameters `AngleStart` and `AngleExtent` determine the range of rotations for which the model is searched. The parameters `ScaleMin` and `ScaleMax` determine the range of scales for which the model is searched. If necessary, both ranges are clipped to the range given when the model was created with `create_scaled_shape_model`. In particular, this means that the angle ranges of the model and the search must overlap.

Note that in some cases instances with a rotation or scale that is slightly outside the specified range are found. This may happen if the specified range is smaller than the range given during the creation of the model. `AngleStart` and `AngleExtent` as well as `ScaleMin` and `ScaleMax` are checked only at the highest pyramid level. Matches that are found on the highest pyramid level are refined to the lowest pyramid level. For performance reasons, however, during the refinement it is no longer checked whether the matches are still within the specified ranges.

`MinScore`: The parameter `MinScore` determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger `MinScore` is chosen, the faster the search is. If the model can be expected never to be occluded in the images, `MinScore` may be set as high as 0.8 or even 0.9. If the matches are not tracked to the lowest pyramid level (see below) it might happen that instances with a score slightly below `MinScore` are found.

In case that the shape model has been extended by clutter parameters with `set_shape_model_clutter` and thus *'use_clutter'* is enabled, `MinScore` expects a second value which determines what clutter score a potential match must at most have to be regarded as an instance of the model in the image. The runtime using clutter parameters will be at least as high as the runtime without clutter parameters and `NumMatches` set to *0*. Changing this second value does not influence the runtime.

`NumMatches`: The maximum number of instances to be found can be determined with `NumMatches`. If more than `NumMatches` instances with a score greater than `MinScore` are found in the image, only the best `NumMatches` instances are returned. If fewer than `NumMatches` are found, only that number is returned, i.e., the parameter `MinScore` takes precedence over `NumMatches`. If all model instances exceeding `MinScore` in the image should be found, `NumMatches` must be set to *0*.

When tracking the matches through the image pyramid, on each level (except the top level), some less promising matches are rejected based on NumMatches. Thus, it is possible that some matches are rejected that would have had a higher score on the lowest pyramid level. Due to this, for example, the found match for NumMatches set to *1* might be different from the match with the highest score returned when setting NumMatches to *0* or > *1*.

If multiple objects with a similar score are expected, but only the one with the highest score should be returned, it might be preferable to raise NumMatches, and then select the match with the highest score.

In case that the shape model has been extended by clutter parameters using set_shape_model_clutter, NumMatches also considers the second value passed in MinScore: If more than NumMatches instances with a score greater than the first entry of MinScore and a clutter score smaller than the second entry of MinScore are found in the image, only the best NumMatches instances with respect to clutter are returned. Still, MinScore takes precedence over NumMatches and NumMatches must be set to *0* if all model instances fulfilling the conditions imposed by MinScore should be found. Please note that using clutter parameters, when tracking the matches through the image pyramid, no matches are rejected. Thus the runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*.

MaxOverlap: If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter MaxOverlap determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than MaxOverlap only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see smallest_rectangle2) of the found instances. If MaxOverlap $= 0$, the found instances may not overlap at all, while for MaxOverlap $= 1$ all instances are returned.

SubPixel: The parameter SubPixel determines whether the instances should be extracted with subpixel accuracy. If SubPixel is set to *'none'* (or *'false'* for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle and scale resolution that was specified with create_scaled_shape_model. If SubPixel is set to *'interpolation'* (or *'true'*) the position as well as the rotation and scale are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast to *'interpolation'*, this mode requires additional computation time. The different modes for least-squares adjustment (*'least_squares'*, *'least_squares_high'*, and *'least_squares_very_high'*) can be used to determine the accuracy with which the minimum distance is being searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, SubPixel should be set to *'interpolation'*. If least-squares adjustment is desired, *'least_squares'* should be chosen because this results in the best trade-off between runtime and accuracy.

Objects that are slightly deformed with respect to the model, in some cases cannot be found or are found but only with a low accuracy. For such objects it is possible to additionally pass a maximal allowable object deformation in the parameter SubPixel. The deformation must be specified in pixels. This can be done by passing the optional parameter value *'max_deformation '* followed by an integer value between *0* and *32* (in the same string), which specifies the maximum deformation. For example, if the shape of the object may be deformed by up to 2 pixels with respect to the shape that is stored in the model, the value *'max_deformation 2'* must be passed in SubPixel in addition to the above described mode for the subpixel extraction, i.e., for example *['least_squares', 'max_deformation 2']*. Passing the value *'max_deformation 0'* corresponds to a search without allowing deformations, i.e., the behavior is the same as if no *'max_deformation '* is passed. Note that higher values for the maximum deformation often result in an increased runtime. Furthermore, the higher the deformation value is chosen, the higher is the risk of finding wrong model instances. Both problems mainly arise when searching for small objects or for objects with fine structures. This is because such kinds of objects for higher deformations lose their characteristic shape, which is important for a robust search. Also note that for higher deformations the accuracy of partially occluded objects might decrease if clutter is present close to the object. Consequently, the maximum deformation should be chosen as small as possible and only as high as necessary. Approximately rotationally symmetric objects may not be found if *'max_deformation'* and AngleExtent are both set to a value greater than 0. In that case, ambiguities may occur that cannot be resolved, and the match is rejected as false. If this happens, try to set either *'max_deformation'* or AngleExtent to 0, or adjust the model such that symmetries are reduced. When specifying a deformation higher than *0* the computation of the score depends on the chosen value for the

subpixel extraction. In most cases, the score of a match changes if *'least_squares'*, *'least_squares_high'*, or *'least_squares_very_high'* (see above) is chosen for the subpixel extraction (in comparison to *'none'* or *'interpolation'*). Furthermore, if one of the least-squares adjustments is selected the score might increase when increasing the maximum deformation because then for the model points more corresponding image points can be found. To get a meaningful score value and to avoid erroneous matches, we recommend to always combine the allowance of a deformation with a least-squares adjustment.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the shape model was created with create_scaled_shape_model. If NumLevels is set to *0*, the number of pyramid levels specified in create_scaled_shape_model is used.

In certain cases, the number of pyramid levels that was determined automatically with, for example, create_scaled_shape_model may be too high. The consequence may be that some matches that may have a high final score are rejected on the highest pyramid level and thus are not found. Instead of setting MinScore to a very low value to find all matches, it may be better to query the value of NumLevels with get_shape_model_params and then use a slightly lower value in find_scaled_shape_model. This approach is often better regarding the speed and robustness of the matching.

Optionally, NumLevels can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of *[4,2]* for NumLevels means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, SubPixel should be set to at least *'least_squares'*. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

In input images of poor quality, i.e., in images that are, e.g., defocused, deformed, or noisy, often no instances of the shape model can be found on the lowest pyramid level. The reason for this behavior is the missing or deformed edge information which is a result of the poor image quality. Nevertheless, the edge information may be sufficient on higher pyramid levels. But keep in mind the above mentioned restrictions on accuracy and robustness if instances that were found on higher pyramid levels are used. The selection of the suitable pyramid level, i.e., the lowest pyramid level on which at least one instance of the shape model can be found, depends on the model and on the input image. This pyramid level may vary from image to image. To facilitate the matching on images of poor quality, the lowest pyramid level on which at least one instance of the model can be found can be determined automatically during the matching. To activate this mechanism, i.e., to use the so-called 'increased tolerance mode', the lowest pyramid level must be specified negatively in NumLevels. If, e.g., NumLevels is set to *[4,-2]*, the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level. This means that an instance of the shape model is searched on the pyramid level 2. If no instance of the model can be found on this pyramid level, the lowest pyramid level is determined on which at least one instance of the model can be found. The instances of this pyramid level will then be returned.

If the ModelID was adapted with adapt_shape_model_high_noise the estimated lowest pyramid level will be used by default. However, the user can override the estimated lowest pyramid level by providing two values to NumLevels and explicitly setting the lowest pyramid level.

Greediness: The parameter Greediness determines how "greedily" the search should be carried out. If Greediness $= 0$, a safe search heuristic is used, which always finds the model if it is visible in the image and the other parameters are set appropriately. However, the search will be relatively time consuming in this case. If Greediness $= 1$, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For Greediness $= 1$, the maximum search speed is achieved. In almost all cases, the shape model will always be found for Greediness $= 0.9$.

Output parameters in detail

Row, Column, Angle, and Scale: The position, rotation, and scale of the found instances of the model are returned in Row, Column, Angle, and Scale. The coordinates Row and Column are related to the position of the origin of the shape model in the search image. However, Row and Column do not exactly correspond to this position. Instead, find_scaled_shape_model returns slightly modified values that are optimized

for creating a transformation matrix, that can be used for alignment or visualization of the model contours. (This has to do with the way HALCON transforms iconic objects, see affine_trans_pixel). The example below shows how to create the transformation matrix for alignment of the found matches and how to use it to display them in the search image.

By default, the model origin is the center of gravity of the domain (region) of the image that was used to create the shape model with create_shape_model. A different origin can be set with set_shape_model_origin.

Score: The score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

In case that the shape model has been extended by clutter parameters using set_shape_model_clutter, following the above values Score also returns the clutter scores of each found instance. If, for example, half of the clutter region is filled by clutter edges, the clutter score will equal 0.5. If, e.g., two instances are found, the score is 0.9 for the first instance and 0.8 for the second instance, and the clutter score is 0.2 for the first instance and 0.1 for the second instance, Score $= [0.9, 0.8, 0.2, 0.1]$ is returned. Please note that of all shape-based matching results, clutter scores are affected the most when a variation of illumination occurs.

Specifying a timeout

Using the operator set_shape_model_param you can specify a *'timeout'* for find_scaled_shape_model. If find_scaled_shape_model reaches this *'timeout'*, it terminates without results and returns the error code 9400 (H_ERR_TIMEOUT). Depending on the scaling range specified by ScaleMin, and ScaleMax, find_scaled_shape_model needs a significant amount of time to free cached transformations if the shape model is not pregenerated. As this transformations have to be freed after a timeout occurs, the runtime of find_scaled_shape_model exceeds the value of the specified *'timeout'* by this time.

Visualization of the results

To display the results found by shape-based matching, we highly recommend the usage of the procedure dev_display_shape_matching_results.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image in which the model should be found.

▷ **ModelID** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the model.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.78
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **ScaleMin** (input_control) . . . . . number ⇝ *real*
Minimum scale of the model.
Default: 0.9
Suggested values: ScaleMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleMin > 0

▷ **ScaleMax** (input_control) . . . . . number ⇝ *real*
Maximum scale of the model.
Default: 1.1
Suggested values: ScaleMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleMax >= ScaleMin

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the model to be found.
Default: 0.5
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer ⇝ *integer*
Number of instances of the model to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real ⇝ *real*
Maximum overlap of the instances of the model to be found.
Default: 0.5
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **SubPixel** (input_control) . . . . . string(-array) ⇝ *string*
Subpixel accuracy if not equal to *'none'*.
Default: 'least_squares'
Suggested values: SubPixel ∈ {'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high', 'max_deformation 1', 'max_deformation 2', 'max_deformation 3', 'max_deformation 4', 'max_deformation 5', 'max_deformation 6'}

▷ **NumLevels** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Greediness** (input_control) . . . . . real ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **Row** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinate of the found instances of the model.

▷ **Column** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinate of the found instances of the model.

▷ **Angle** (output_control) . . . . . angle.rad-array ⇝ *real*
Rotation angle of the found instances of the model.

▷ **Scale** (output_control) . . . . . number-array ⇝ *real*
Scale of the found instances of the model.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the model.

Example

```
create_scaled_shape_model (ImageReduced, 0, rad(-45), rad(180), 0, \
                           0.9, 1.1, 0, 'none', 'use_polarity', \
                           30, 10, ModelID)
get_shape_model_contours (ModelXLD, ModelID, 1)
find_scaled_shape_model (SearchImage, ModelID, rad(-45), rad(180), \
                         0.9, 1.1, 0.5, 1, 0.5, 'interpolation', \
                         0, 0, Row, Column, Angle, Scale, Score)
* Create transformation matrix
vector_angle_to_rigid (0, 0, 0, Row, Column, Angle, HomMat2DTmp)
hom_mat2d_scale (HomMat2DTmp, Scale, Scale, Row, Column, HomMat2DObject)
```

```
* Calculate true position of the model origin in the search image
affine_trans_pixel (HomMat2DObject, 0, 0, RowObject, ColObject)
* Display results
dev_display_shape_matching_results (ModelID, 'red', Row, Column, Angle, \
                    Scale, 1, 0)
```

Result

If the parameter values are correct, the operator `find_scaled_shape_model` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input images are available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`create_scaled_shape_model`, `read_shape_model`, `set_shape_model_origin`, `set_shape_model_clutter`

Possible Successors

`clear_shape_model`

Alternatives

`find_generic_shape_model`

See also

`set_system`, `get_system`, `set_shape_model_param`

Module

Matching

find_scaled_shape_models (Image : : ModelIDs, AngleStart, AngleExtent, ScaleMin, ScaleMax, MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels, Greediness : Row, Column, Angle, Scale, Score, Model)

Find the best matches of multiple isotropically scaled shape models.

The operator `find_scaled_shape_models` finds the best `NumMatches` instances of the isotropically scaled shape models that are passed in `ModelIDs` in the input image `Image`. The models must have been created previously by calling `create_scaled_shape_model` or `read_shape_model`. In contrast to `find_scaled_shape_model`, multiple models can be searched in the same image in one call.

The position, rotation, and scale of the found instances of the model are returned in `Row`, `Column`, `Angle`, and `Scale`. The score of each found instance is returned in `Score`. The type of the found instances of the models is returned in `Model`. For details see respective sections below.

Characteristics of the parameter semantics

Compared to `find_scaled_shape_model`, the semantics of all input parameters have changed to some extent. All input parameters must either contain one element, in which case the parameter is used for all models, or must contain the same number of elements as `ModelIDs`, in which case each parameter element refers to the corresponding element in `ModelIDs`. (`NumLevels` may also contain either two or twice the number of elements as `ModelIDs`.) More details can be found below in the sections containing information for the respective parameters. Note that a call to `find_scaled_shape_models` with multiple values for `ModelIDs`, `NumMatches` and `MaxOverlap` has the same effect as multiple independent calls to `find_scaled_shape_model` with the respective parameters. However, a single call to `find_scaled_shape_models` is considerably more efficient.

Input parameters in detail

Image and its domain: The domain of the Image determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with create_scaled_shape_model. A different origin set with set_shape_model_origin is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than MinScore (see below). Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed with set_system('border_shape_models','true') for all models or with set_shape_model_param(ModelID,'border_shape_models','true') for a specific model, which will cause models that extend beyond the image border to be found if they achieve a score greater than MinScore. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the search will increase in this mode. When searching multiple models *'border_shape_models'* is treated as *'true'* for all models even if *'border_shape_models'* only evaluates to *'true'* for one of the models in a search. Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels the model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., dilation_circle.

As usual, the domain of the input Image is used to restrict the search space for the reference point of the models ModelIDs. Consistent with the above semantics, the input Image can therefore contain a single image object or an image object tuple containing multiple image objects. If Image contains a single image object, its domain is used as the region of interest for all models in ModelIDs. If Image contains multiple image objects, each domain is used as the region of interest for the corresponding model in ModelIDs. In this case, the images have to be identical except for their domains, i.e., their pointers have to be identical (the pointers can be checked using get_image_pointer1). As a consequence, Image cannot be constructed in an arbitrary manner using concat_obj, but must be created from the same image using add_channels or equivalent calls. If this is not the case, an error message is returned.

AngleStart, AngleExtent, ScaleMin, ScaleMax: The parameters AngleStart and AngleExtent determine the range of rotations for which the model is searched. The parameters ScaleMin and ScaleMax determine the range of scales for which the model is searched. If necessary, both ranges are clipped to the range given when the model was created with create_scaled_shape_model. In particular, this means that the angle ranges of the model and the search must overlap.

Note that in some cases instances with a rotation or scale that is slightly outside the specified range are found. This may happen if the specified range is smaller than the range given during the creation of the model. AngleStart and AngleExtent as well as ScaleMin and ScaleMax are checked only at the highest pyramid level. Matches that are found on the highest pyramid level are refined to the lowest pyramid level. For performance reasons, however, during the refinement it is no longer checked whether the matches are still within the specified ranges.

MinScore: The parameter MinScore determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger MinScore is chosen, the faster the search is. If the model can be expected never to be occluded in the images, MinScore may be set as high as 0.8 or even 0.9. If the matches are not tracked to the lowest pyramid level (see below) it might happen that instances with a score slightly below MinScore are found.

If a single value is passed in MinScore, this value is applied to found instances of all models. If, on the other hand, MinScore contains multiple values, the values are applied separately for the respective model.

In case that the shape models have been extended by clutter parameters with set_shape_model_clutter and thus *'use_clutter'* is enabled, MinScore expects for each minimum score an additional value which determines what clutter score a potential match must at most have to be regarded as an instance of the model in the image. The runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*. Changing this second value does not influence the runtime. Note that the different shape models must have the same value for *'use_clutter'*.

If the maximum clutter is specified separately for each model, which is needed if also the minimum score is set for each model individually, MinScore must contain twice the number of elements as ModelIDs. In this case, the minimum score and the maximum clutter must be specified interleaved in MinScore. If, for example, two models are specified in ModelIDs, the minimum score is 0.9 for the first model and 0.8 for the second model, and the maximum clutter is 0.1 for the first model and 0.2 for the second model, MinScore $= [0.9, 0.1, 0.8, 0.2]$ must be selected.

NumMatches: The maximum number of instances to be found can be determined with NumMatches. If more than NumMatches instances with a score greater than MinScore are found in the image, only the best NumMatches instances are returned. If fewer than NumMatches are found, only that number is returned, i.e., the parameter MinScore takes precedence over NumMatches. If all model instances exceeding MinScore in the image should be found, NumMatches must be set to *0*.

When tracking the matches through the image pyramid, on each level (except the top level), some less promising matches are rejected based on NumMatches. Thus, it is possible that some matches are rejected that would have had a higher score on the lowest pyramid level. Due to this, for example, the found match for NumMatches set to *1* might be different from the match with the highest score returned when setting NumMatches to *0* or > *1*.

If multiple objects with a similar score are expected, but only the one with the highest score should be returned, it might be preferable to raise NumMatches, and then select the match with the highest score.

In case that the shape models have been extended by clutter parameters using set_shape_model_clutter, NumMatches also considers the second value passed in MinScore: If more than NumMatches instances with a score greater than the first entry of MinScore and a clutter score smaller than the second entry of MinScore are found in the image, only the best NumMatches instances with respect to clutter are returned. Still, MinScore takes precedence over NumMatches and NumMatches must be set to *0* if all model instances fulfilling the conditions imposed by MinScore should be found. Please note that using clutter parameters, when tracking the matches through the image pyramid, no matches are rejected. Thus the runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*.

If NumMatches contains one element, find_scaled_shape_models returns the best NumMatches instances of the model irrespective of the type of the model. If, for example, two models are passed in ModelIDs and NumMatches $= 2$ is selected, it can happen that two instances of the first model and no instances of the second model, one instance of the first model and one instance of the second model, or no instances of the first model and two instances of the second model are returned. If, on the other hand, NumMatches contains multiple values, the number of instances returned of the different models corresponds to the number specified in the respective entry in NumMatches. If, for example, NumMatches $= [1, 1]$ is selected, one instance of the first model and one instance of the second model is returned.

MaxOverlap: If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter MaxOverlap determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than MaxOverlap only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see smallest_rectangle2) of the found instances. If MaxOverlap $= 0$, the found instances may not overlap at all, while for MaxOverlap $= 1$ all instances are returned.

If a single value is passed in MaxOverlap, the overlap is computed for all found instances of the different models, irrespective of the model type, i.e., instances of the same or of different models that overlap too much are eliminated. If, on the other hand, multiple values are passed in MaxOverlap, the overlap is only computed for found instances of the model that have the same model type, i.e., only instances of the same model that overlap too much are eliminated. In this mode, models of different types may overlap completely.

SubPixel: The parameter SubPixel determines whether the instances should be extracted with subpixel accuracy. If SubPixel is set to *'none'* (or *'false'* for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle and scale resolution that was specified with create_scaled_shape_model. If SubPixel is set to *'interpolation'* (or *'true'*) the position as well as the rotation and scale are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast to *'interpolation'*, this mode requires additional computation time. The different modes for least-squares adjustment (*'least_squares'*, *'least_squares_high'*, and *'least_squares_very_high'*) can be used to determine the accuracy with which the minimum distance is being searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, SubPixel should be set to *'interpolation'*. If least-squares adjustment is desired, *'least_squares'* should be chosen because this results in the best trade-off between runtime and accuracy.

Objects that are slightly deformed with respect to the model, in some cases cannot be found or are found but only with a low accuracy. For such objects it is possible to additionally pass a maximal allowable object

deformation in the parameter SubPixel. The deformation must be specified in pixels. This can be done by passing the optional parameter value *'max_deformation '* followed by an integer value between *0* and *32* (in the same string), which specifies the maximum deformation. For example, if the shape of the object may be deformed by up to 2 pixels with respect to the shape that is stored in the model, the value *'max_deformation 2'* must be passed in SubPixel in addition to the above described mode for the subpixel extraction, i.e., for example *['least_squares', 'max_deformation 2']*. Passing the value *'max_deformation 0'* corresponds to a search without allowing deformations, i.e., the behavior is the same as if no *'max_deformation '* is passed. Note that higher values for the maximum deformation often result in an increased runtime. Furthermore, the higher the deformation value is chosen, the higher is the risk of finding wrong model instances. Both problems mainly arise when searching for small objects or for objects with fine structures. This is because such kinds of objects for higher deformations lose their characteristic shape, which is important for a robust search. Also note that for higher deformations the accuracy of partially occluded objects might decrease if clutter is present close to the object. Consequently, the maximum deformation should be chosen as small as possible and only as high as necessary. Approximately rotationally symmetric objects may not be found if *'max_deformation'* and AngleExtent are both set to a value greater than 0. In that case, ambiguities may occur that cannot be resolved, and the match is rejected as false. If this happens, try to set either *'max_deformation'* or AngleExtent to 0, or adjust the model such that symmetries are reduced. When specifying a deformation higher than *0* the computation of the score depends on the chosen value for the subpixel extraction. In most cases, the score of a match changes if *'least_squares'*, *'least_squares_high'*, or *'least_squares_very_high'* (see above) is chosen for the subpixel extraction (in comparison to *'none'* or *'interpolation'*). Furthermore, if one of the least-squares adjustments is selected the score might increase when increasing the maximum deformation because then for the model points more corresponding image points can be found. To get a meaningful score value and to avoid erroneous matches, we recommend to always combine the allowance of a deformation with a least-squares adjustment.

If the subpixel extraction and/or the maximum object deformation is specified separately for each model, for each model passed in ModelIDs exactly one value for the subpixel extraction must be passed in SubPixel. After each value for the subpixel extraction optionally a second value can be passed, which describes the maximum object deformation of the corresponding mode. If for a certain model no value for the maximum object deformation is passed, the model is searched without taking deformations into account. For example, if two models are passed in ModelIDs and for the first model the subpixel extraction is set to *'interpolation'* and no object deformations are allowed and for the second model the subpixel extraction is set to *'least_squares'* and a maximum object deformation of *3* pixels is allowed, then the tuple *['interpolation', 'least_squares', 'max_deformation 3']* must be passed in SubPixel. Alternatively, the equivalent tuple *['interpolation', 'max_deformation 0', 'least_squares', 'max_deformation 3']* may be passed.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the shape model was created with create_scaled_shape_model. If NumLevels is set to *0*, the number of pyramid levels specified in create_scaled_shape_model is used.

In certain cases, the number of pyramid levels that was determined automatically with, for example, create_scaled_shape_model may be too high. The consequence may be that some matches that may have a high final score are rejected on the highest pyramid level and thus are not found. Instead of setting MinScore to a very low value to find all matches, it may be better to query the value of NumLevels with get_shape_model_params and then use a slightly lower value in find_scaled_shape_models. This approach is often better regarding the speed and robustness of the matching.

Optionally, NumLevels can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of *[4,2]* for NumLevels means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, SubPixel should be set to at least *'least_squares'*. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

If the lowest pyramid level is specified separately for each model, NumLevels must contain twice the number of elements as ModelIDs. In this case, the number of pyramid levels and the lowest pyramid level must be specified interleaved in NumLevels. If, for example, two models are specified in ModelIDs, the

number of pyramid levels is 5 for the first model and 4 for the second model, and the lowest pyramid level is 2 for the first model and 1 for the second model, NumLevels $= [5, 2, 4, 1]$ must be selected. If exactly two models are specified in ModelIDs, a special case occurs. If in this case the lowest pyramid level is to be specified, the number of pyramid levels and the lowest pyramid level must be specified explicitly for both models, even if they are identical, because specifying two values in NumLevels is interpreted as the explicit specification of the number of pyramid levels for the two models.

In input images of poor quality, i.e., in images that are, e.g., defocused, deformed, or noisy, often no instances of the shape model can be found on the lowest pyramid level. The reason for this behavior is the missing or deformed edge information which is a result of the poor image quality. Nevertheless, the edge information may be sufficient on higher pyramid levels. But keep in mind the above mentioned restrictions on accuracy and robustness if instances that were found on higher pyramid levels are used. The selection of the suitable pyramid level, i.e., the lowest pyramid level on which at least one instance of the shape model can be found, depends on the model and on the input image. This pyramid level may vary from image to image. To facilitate the matching on images of poor quality, the lowest pyramid level on which at least one instance of the model can be found can be determined automatically during the matching. To activate this mechanism, i.e., to use the so-called 'increased tolerance mode', the lowest pyramid level must be specified negatively in NumLevels. If, e.g., NumLevels is set to *[5,2,4,-1]*, the lowest pyramid level for the first model is set to 2. If no instance of the first model can be found on this pyramid level, no result will be returned for this model. For the second shape model, the lowest pyramid level is set to *-1*. Therefore, an instance of the shape model is searched on the pyramid level 1. If no instance of the second model can be found on this pyramid level, the lowest pyramid level is determined on which at least one instance of the model can be found. The instances of this pyramid level will then be returned.

If a model was adapted with adapt_shape_model_high_noise the estimated lowest pyramid level will be used by default. However, the user can override the estimated lowest pyramid level by providing explicitly the lowest pyramid level as described above.

Greediness: The parameter Greediness determines how "greedily" the search should be carried out. If Greediness $= 0$, a safe search heuristic is used, which always finds the model if it is visible in the image and the other parameters are set appropriately. However, the search will be relatively time consuming in this case. If Greediness $= 1$, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For Greediness $= 1$, the maximum search speed is achieved. In almost all cases, the shape model will always be found for Greediness $= 0.9$.

Output parameters in detail

Row, Column, Angle, and Scale: The position, rotation, and scale of the found instances of the model are returned in Row, Column, Angle, and Scale. The coordinates Row and Column are the coordinates of the origin of the shape model in the search image. By default, the origin is the center of gravity of the domain (region) of the image that was used to create the shape model with create_scaled_shape_model. A different origin can be set with set_shape_model_origin.

Note that the coordinates Row and Column do not exactly correspond to the position of the model in the search image. Thus, you cannot directly use them. Instead, the values are optimized for creating the transformation matrix with which you can use the results of the matching for various tasks, e.g., to align ROIs for other processing steps. The example given for find_scaled_shape_model shows how to create this matrix and use it to display the model at the found position in the search image.

Note also that for visualizing the model at the found position, also the procedure dev_display_shape_matching_results can be used.

Score: The score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

In case that the shape models have been extended by clutter parameters using set_shape_model_clutter, following the above values Score also returns the clutter scores of each found instance. If, for example, half of the clutter region is filled by clutter edges, the clutter score will equal 0.5. If, e.g., two instances are found, the score is 0.9 for the first instance and 0.8 for the second instance, and the clutter score is 0.2 for the first instance and 0.1 for the second instance, Score $= [0.9, 0.8, 0.2, 0.1]$ is returned. Please note that of all shape-based matching results, clutter scores are affected the most when a variation of illumination occurs.

Model: The type of the found instances of the models is returned in Model. The elements of Model are indices into the tuple ModelIDs, i.e., they can contain values from 0 to $|\text{ModelIDs}| - 1$. Hence, a value of 0 in an element of Model corresponds to an instance of the first model in ModelIDs.

Specifying a timeout

Using the operator set_shape_model_param you can specify a *'timeout'* for find_scaled_shape_models. If the shape models referenced by ModelIDs hold different values for *'timeout'*, find_scaled_shape_models uses the smallest one. If find_scaled_shape_models reaches this *'timeout'*, it terminates without results and returns the error code 9400 (H_ERR_TIMEOUT). Depending on the scaling range specified by ScaleMin and ScaleMax, find_scaled_shape_models needs a significant amount of time to free cached transformations if the shape model is not pregenerated. As this transformations have to be freed after a timeout occurs, the runtime of find_scaled_shape_models exceeds the value of the specified *'timeout'* by this time.

MinContrast with multiple models

Please note, that the different models that are given with the parameter ModelIDs should have been created with the same value of MinContrast. If they were created with different values for MinContrast, find_scaled_shape_models will use the smallest of these values.

Visualization of the results

To display the results found by shape-based matching, we highly recommend the usage of the procedure dev_display_shape_matching_results.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Input image in which the models should be found.

▷ **ModelIDs** (input_control) . . . . . shape_model(-array) ⇝ *handle*
Handle of the models.

▷ **AngleStart** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Smallest rotation of the models.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Extent of the rotation angles.
Default: 0.78
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **ScaleMin** (input_control) . . . . . number(-array) ⇝ *real*
Minimum scale of the models.
Default: 0.9
Suggested values: ScaleMin ∈ {0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: ScaleMin > 0

▷ **ScaleMax** (input_control) . . . . . number(-array) ⇝ *real*
Maximum scale of the models.
Default: 1.1
Suggested values: ScaleMax ∈ {1.0, 1.1, 1.2, 1.3, 1.4, 1.5}
Restriction: ScaleMax >= ScaleMin

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the models to be found.
Default: 0.5
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of instances of the models to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real(-array) ⇝ *real*
Maximum overlap of the instances of the models to be found.
Default: 0.5
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **SubPixel** (input_control) . . . . . string(-array) ⇝ *string*
Subpixel accuracy if not equal to *'none'*.
Default: 'least_squares'
Suggested values: SubPixel ∈ {'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high', 'max_deformation 1', 'max_deformation 2', 'max_deformation 3', 'max_deformation 4', 'max_deformation 5', 'max_deformation 6'}

▷ **NumLevels** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Greediness** (input_control) . . . . . real(-array) ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **Row** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinate of the found instances of the models.

▷ **Column** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinate of the found instances of the models.

▷ **Angle** (output_control) . . . . . angle.rad-array ⇝ *real*
Rotation angle of the found instances of the models.

▷ **Scale** (output_control) . . . . . number-array ⇝ *real*
Scale of the found instances of the models.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the models.

▷ **Model** (output_control) . . . . . integer-array ⇝ *integer*
Index of the found instances of the models.

Example

```
read_image (Image, 'pcb_focus/pcb_focus_telecentric_061')
gen_rectangle1 (ROI_0, 236, 241, 313, 321)
gen_circle (ROI_1, 281, 653, 41)
reduce_domain (Image, ROI_0, ImageReduced1)
reduce_domain (Image, ROI_1, ImageReduced2)
create_scaled_shape_model (ImageReduced1,'auto', -0.39, 0.79, 'auto', 0.9, \
                           1.1, 'auto', 'auto', 'use_polarity', 'auto', \
                           'auto', ModelID1)
create_scaled_shape_model (ImageReduced2, 'auto', -0.39, 0.79, 'auto', 0.9,\
                           1.1, 'auto', 'auto', 'use_polarity', 'auto', \
                           'auto', ModelID2)
ModelIDs:=[ModelID1, ModelID2]
find_scaled_shape_models (Image, ModelIDs, -0.39, 0.78, 0.9, 1.1, 0.5, 1, \
                          0.5, 'least_squares', 0, 0.9, Row, Column, Angle,\
                          Scale, Score, Model)
* Display results
dev_display_shape_matching_results (ModelIDs, 'red', Row, Column, Angle, \
                                    Scale, 1, Model)
```

Result

If the parameter values are correct, the operator `find_scaled_shape_models` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input images are available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`add_channels`, `create_scaled_shape_model`, `read_shape_model`, `set_shape_model_origin`, `set_shape_model_clutter`

Possible Successors

`clear_shape_model`

Alternatives

`find_generic_shape_model`

See also

`set_system`, `get_system`, `set_shape_model_param`

Module

Matching

```
find_shape_model ( Image : : ModelID, AngleStart, AngleExtent,
    MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels,
    Greediness : Row, Column, Angle, Score )
```

Find the best matches of a shape model in an image.

The operator `find_shape_model` finds the best `NumMatches` instances of the shape model `ModelID` in the input image `Image`. The model must have been created previously by calling `create_shape_model` or `read_shape_model`.

The position and rotation of the found instances of the model is returned in `Row`, `Column`, and `Angle`. Additionally, the score of each found instance is returned in `Score`.

Input parameters in detail

`Image` and its domain: The domain of the image `Image` determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with `create_shape_model`. A different origin set with `set_shape_model_origin` is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than `MinScore` (see below). Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed with `set_system('border_shape_models','true')` for all models or with `set_shape_model_param(ModelID,'border_shape_models','true')` for a specific model, which will cause models that extend beyond the image border to be found if they achieve a score greater than `MinScore`. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the search will increase in this mode. Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels the model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., `dilation_circle`.

`AngleStart` and `AngleExtent`: The parameters `AngleStart` and `AngleExtent` determine the range of rotations for which the model is searched. If necessary, the range of rotations is clipped to the range given

when the model was created with create_shape_model. In particular, this means that the angle ranges of the model and the search must overlap.

Note that in some cases instances with a rotation that is slightly outside the specified range are found. This may happen if the specified range is smaller than the range given during the creation of the model. AngleStart and AngleExtent are checked only at the highest pyramid level. Matches that are found on the highest pyramid level are refined to the lowest pyramid level. For performance reasons, however, during the refinement it is no longer checked whether the matches are still within the specified range.

MinScore: The parameter MinScore determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger MinScore is chosen, the faster the search is. If the model can be expected never to be occluded in the images, MinScore may be set as high as 0.8 or even 0.9. If the matches are not tracked to the lowest pyramid level (see below) it might happen that instances with a score slightly below MinScore are found.

In case that the shape model has been extended by clutter parameters with set_shape_model_clutter and thus *'use_clutter'* is enabled, MinScore expects a second value which determines what clutter score a potential match must at most have to be regarded as an instance of the model in the image. The runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*. Changing this second value does not influence the runtime.

NumMatches: The maximum number of instances to be found can be determined with NumMatches. If more than NumMatches instances with a score greater than MinScore are found in the image, only the best NumMatches instances are returned. If fewer than NumMatches are found, only that number is returned, i.e., the parameter MinScore takes precedence over NumMatches. If all model instances exceeding MinScore in the image should be found, NumMatches must be set to *0*.

When tracking the matches through the image pyramid, on each level (except the top level), some less promising matches are rejected based on NumMatches. Thus, it is possible that some matches are rejected that would have had a higher score on the lowest pyramid level. Due to this, for example, the found match for NumMatches set to *1* might be different from the match with the highest score returned when setting NumMatches to *0* or > 1.

If multiple objects with a similar score are expected, but only the one with the highest score should be returned, it might be preferable to raise NumMatches, and then select the match with the highest score.

In case that the shape model has been extended by clutter parameters using set_shape_model_clutter, NumMatches also considers the second value passed in MinScore. Matched instances will be in this case instances with a score greater than the first entry of MinScore and a clutter score smaller than its second entry. If the number of matched instances in the image is larger than NumMatches, the best NumMatches instances with respect to clutter scores are returned. Still, MinScore takes precedence over NumMatches and NumMatches must be set to *0* if all model instances fulfilling the conditions imposed by MinScore should be returned. Please note that using clutter parameters, when tracking the matches through the image pyramid, no matches are rejected. Thus the runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*.

MaxOverlap: If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter MaxOverlap determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than MaxOverlap only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see smallest_rectangle2) of the found instances. If MaxOverlap $= 0$, the found instances may not overlap at all, while for MaxOverlap $= 1$ all instances are returned.

SubPixel: The parameter SubPixel determines whether the instances should be extracted with subpixel accuracy. If SubPixel is set to *'none'* (or *'false'* for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle resolution that was specified with create_shape_model. If SubPixel is set to *'interpolation'* (or *'true'*) the position as well as the rotation are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast to *'interpolation'*, this mode requires additional computation time. The different modes for least-squares adjustment (*'least_squares'*, *'least_squares_high'*, and *'least_squares_very_high'*) can be used to determine the accuracy with which the minimum distance is being

searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, SubPixel should be set to *'interpolation'*. If least-squares adjustment is desired, *'least_squares'* should be chosen because this results in the best trade-off between runtime and accuracy.

Objects that are slightly deformed with respect to the model, in some cases cannot be found or are found but only with a low accuracy. For such objects it is possible to additionally pass a maximal allowable object deformation in the parameter SubPixel. The deformation must be specified in pixels. This can be done by passing the optional parameter value *'max_deformation '* followed by an integer value between *0* and *32* (in the same string), which specifies the maximum deformation. For example, if the shape of the object may be deformed by up to 2 pixels with respect to the shape that is stored in the model, the value *'max_deformation 2'* must be passed in SubPixel in addition to the above described mode for the subpixel extraction, i.e., for example *['least_squares', 'max_deformation 2']*. Passing the value *'max_deformation 0'* corresponds to a search without allowing deformations, i.e., the behavior is the same as if no *'max_deformation '* is passed. Note that higher values for the maximum deformation often result in an increased runtime. Furthermore, the higher the deformation value is chosen, the higher is the risk of finding wrong model instances. Both problems mainly arise when searching for small objects or for objects with fine structures. This is because such kinds of objects for higher deformations lose their characteristic shape, which is important for a robust search. Also note that for higher deformations the accuracy of partially occluded objects might decrease if clutter is present close to the object. Consequently, the maximum deformation should be chosen as small as possible and only as high as necessary. Approximately rotationally symmetric objects may not be found if *'max_deformation'* and AngleExtent are both set to a value greater than 0. In that case, ambiguities may occur that cannot be resolved, and the match is rejected as false. If this happens, try to set either *'max_deformation'* or AngleExtent to 0, or adjust the model such that symmetries are reduced. When specifying a deformation higher than *0* the computation of the score depends on the chosen value for the subpixel extraction. In most cases, the score of a match changes if *'least_squares'*, *'least_squares_high'*, or *'least_squares_very_high'* (see above) is chosen for the subpixel extraction (in comparison to *'none'* or *'interpolation'*). Furthermore, if one of the least-squares adjustments is selected the score might increase when increasing the maximum deformation because then for the model points more corresponding image points can be found. To get a meaningful score value and to avoid erroneous matches, we recommend to always combine the allowance of a deformation with a least-squares adjustment.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the shape model was created with create_shape_model. If NumLevels is set to *0*, the number of pyramid levels specified in create_shape_model is used.

In certain cases, the number of pyramid levels that was determined automatically with, for example, create_shape_model may be too high. The consequence may be that some matches that may have a high final score are rejected on the highest pyramid level and thus are not found. Instead of setting MinScore to a very low value to find all matches, it may be better to query the value of NumLevels with get_shape_model_params and then use a slightly lower value in find_shape_model. This approach is often better regarding the speed and robustness of the matching.

Optionally, NumLevels can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of *[4,2]* for NumLevels means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, SubPixel should be set to at least *'least_squares'*. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

In input images of poor quality, i.e., in images that are, e.g., defocused, deformed, or noisy, often no instances of the shape model can be found on the lowest pyramid level. The reason for this behavior is the missing or deformed edge information which is a result of the poor image quality. Nevertheless, the edge information may be sufficient on higher pyramid levels. But keep in mind the above mentioned restrictions on accuracy and robustness if instances that were found on higher pyramid levels are used. The selection of the suitable pyramid level, i.e., the lowest pyramid level on which at least one instance of the shape model can be found, depends on the model and on the input image. This pyramid level may vary from image to image. To facilitate the matching on images of poor quality, the lowest pyramid level on which at least one instance of

the model can be found can be determined automatically during the matching. To activate this mechanism, i.e., to use the so-called 'increased tolerance mode', the lowest pyramid level must be specified negatively in NumLevels. If, e.g., NumLevels is set to *[4,-2]*, the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level. This means that an instance of the shape model is searched on the pyramid level 2. If no instance of the model can be found on this pyramid level, the lowest pyramid level is determined on which at least one instance of the model can be found. The instances of this pyramid level will then be returned.

If the ModelID was adapted with adapt_shape_model_high_noise the estimated lowest pyramid level will be used by default. However, the user can override the estimated lowest pyramid level by providing two values to NumLevels and explicitly setting the lowest pyramid level.

Greediness: The parameter Greediness determines how "greedily" the search should be carried out. If Greediness $= 0$, a safe search heuristic is used, which always finds the model if it is visible in the image and the other parameters are set appropriately. However, the search will be relatively time consuming in this case. If Greediness $= 1$, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For Greediness $= 1$, the maximum search speed is achieved. In almost all cases, the shape model will always be found for Greediness $= 0.9$.

Output parameters in detail

Row, Column, and Angle: The position and rotation of the found instances of the model is returned in Row, Column, and Angle. The coordinates Row and Column are related to the position of the origin of the shape model in the search image. However, Row and Column do not exactly correspond to this position. Instead, find_shape_model returns slightly modified values that are optimized for creating a transformation matrix, that can be used for alignment or visualization of the model contours. (This has to do with the way HALCON transforms iconic objects, see affine_trans_pixel). The example below shows how to create the transformation matrix for alignment of the found matches and how to use it to display them in the search image.

By default, the model origin is the center of gravity of the domain (region) of the image that was used to create the shape model with create_aniso_shape_model. A different origin can be set with set_shape_model_origin.

Score: The score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

In case that the shape model has been extended by clutter parameters using set_shape_model_clutter, following the above values Score also returns the clutter score of each found instance. If, for example, half of the clutter region is filled by clutter edges, the clutter score will equal 0.5. If, e.g., two instances are found, the score is 0.9 for the first instance and 0.8 for the second instance, and the clutter score is 0.2 for the first instance and 0.1 for the second instance, Score $= [0.9, 0.8, 0.2, 0.1]$ is returned. Please note that of all shape-based matching results, clutter scores are affected the most when a variation of illumination occurs.

Specifying a timeout

Using the operator set_shape_model_param you can specify a *'timeout'* for find_shape_model. If find_shape_model reaches this *'timeout'*, it terminates without results and returns the error code 9400 (H_ERR_TIMEOUT).

Visualization of the results

To display the results found by shape-based matching, we highly recommend the usage of the procedure dev_display_shape_matching_results.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
Input image in which the model should be found.

▷ **ModelID** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the model.

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Smallest rotation of the model.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the model to be found.
Default: 0.5
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer ⇝ *integer*
Number of instances of the model to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real ⇝ *real*
Maximum overlap of the instances of the model to be found.
Default: 0.5
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **SubPixel** (input_control) . . . . . string(-array) ⇝ *string*
Subpixel accuracy if not equal to *'none'*.
Default: 'least_squares'
Suggested values: SubPixel ∈ {'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high', 'max_deformation 1', 'max_deformation 2', 'max_deformation 3', 'max_deformation 4', 'max_deformation 5', 'max_deformation 6'}

▷ **NumLevels** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Greediness** (input_control) . . . . . real ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **Row** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinate of the found instances of the model.

▷ **Column** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinate of the found instances of the model.

▷ **Angle** (output_control) . . . . . angle.rad-array ⇝ *real*
Rotation angle of the found instances of the model.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the model.

Example

```
create_shape_model (ImageReduced, 0, rad(-45), rad(180), 0, \
                    'none', 'use_polarity', 30, 10, ModelID)
get_shape_model_contours (ModelXLD, ModelID, 1)
```

```
find_shape_model (SearchImage, ModelID, rad(-45), rad(180), \
                  0.5, 1, 0.5, 'interpolation', \
                  0, 0, Row, Column, Angle, Score)
* Create transformation matrix
vector_angle_to_rigid (0, 0, 0, Row, Column, Angle, HomMat2DObject)
* Display results
dev_display_shape_matching_results (ModelID, 'red', Row, Column, Angle, \
                                    1, 1, 0)
```

Result

If the parameter values are correct, the operator `find_shape_model` returns the value `2` (`H_MSG_TRUE`). If the input is empty (no input images are available) the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`create_shape_model`, `read_shape_model`, `set_shape_model_origin`, `set_shape_model_clutter`

Possible Successors

`clear_shape_model`

Alternatives

`find_generic_shape_model`

See also

`set_system`, `get_system`, `set_shape_model_param`

Module

Matching

find_shape_models (Image : : ModelIDs, AngleStart, AngleExtent, MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels, Greediness : Row, Column, Angle, Score, Model)

Find the best matches of multiple shape models.

The operator `find_shape_models` finds the best `NumMatches` instances of the shape models that are passed in the tuple `ModelIDs` in the input image `Image`. The models must have been created previously by calling `create_shape_model` or `read_shape_model`. In contrast to `find_shape_model`, multiple models can be searched in the same image in one call.

The position and rotation of the found instances of the model is returned in `Row`, `Column`, and `Angle`. The score of each found instance is returned in `Score`. The type of the found instances of the models is returned in `Model`. For details see respective sections below.

Characteristics of the parameter semantics

Compared to `find_shape_model`, the semantics of all input parameters have changed to some extent. All input parameters must either contain one element, in which case the parameter is used for all models, or must contain the same number of elements as `ModelIDs`, in which case each parameter element refers to the corresponding element in `ModelIDs`. (`NumLevels` may also contain either two or twice the number of elements as `ModelIDs`.) More details can be found below in the sections containing information for the respective parameters. Note that a call to `find_shape_models` with multiple values for `ModelIDs`, `NumMatches` and `MaxOverlap` has the same effect as multiple independent calls to `find_shape_model` with the respective parameters. However, a single call to `find_shape_models` is considerably more efficient.

Input parameters in detail

Image and its domain: The domain of the Image determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with create_shape_model. A different origin set with set_shape_model_origin is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than MinScore (see below). Note that, if for a certain pyramid level the model touches the image border, it might not be found even if it lies completely within the original image. As a rule of thumb, the model might not be found if its distance to an image border falls below $2^{NumLevels-1}$. This behavior can be changed with set_system('border_shape_models','true') for all models or with set_shape_model_param(ModelID,'border_shape_models','true') for a specific model, which will cause models that extend beyond the image border to be found if they achieve a score greater than MinScore. Here, points lying outside the image are regarded as being occluded, i.e., they lower the score. It should be noted that the runtime of the search will increase in this mode. When searching multiple models *'border_shape_models'* is treated as *'true'* for all models even if *'border_shape_models'* only evaluates to *'true'* for one of the models in a search. Note further, that in rare cases, which occur typically only for artificial images, the model might not be found also if for certain pyramid levels the model touches the border of the reduced domain. Then, it may help to enlarge the reduced domain by $2^{NumLevels-1}$ using, e.g., dilation_circle.

As usual, the domain of the input Image is used to restrict the search space for the reference point of the models ModelIDs. Consistent with the above semantics, the input Image can therefore contain a single image object or an image object tuple containing multiple image objects. If Image contains a single image object, its domain is used as the region of interest for all models in ModelIDs. If Image contains multiple image objects, each domain is used as the region of interest for the corresponding model in ModelIDs. In this case, the images have to be identical except for their domains, i.e., their pointers have to be identical (the pointers can be checked using get_image_pointer1). As a consequence, Image cannot be constructed in an arbitrary manner using concat_obj, but must be created from the same image using add_channels or equivalent calls. If this is not the case, an error message is returned.

AngleStart and AngleExtent: The parameters AngleStart and AngleExtent determine the range of rotations for which the model is searched. If necessary, the range of rotations is clipped to the range given when the model was created with create_shape_model. In particular, this means that the angle ranges of the model and the search must overlap.

Note that in some cases instances with a rotation that is slightly outside the specified range are found. This may happen if the specified range is smaller than the range given during the creation of the model. AngleStart and AngleExtent are checked only at the highest pyramid level. Matches that are found on the highest pyramid level are refined to the lowest pyramid level. For performance reasons, however, during the refinement it is no longer checked whether the matches are still within the specified range.

MinScore: The parameter MinScore determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger MinScore is chosen, the faster the search is. If the model can be expected never to be occluded in the images, MinScore may be set as high as 0.8 or even 0.9. If the matches are not tracked to the lowest pyramid level (see below) it might happen that instances with a score slightly below MinScore are found.

If a single value is passed in MinScore, this value is applied to found instances of all models. If, on the other hand, MinScore contains multiple values, the values are applied separately for the respective model.

In case that the shape models have been extended by clutter parameters with set_shape_model_clutter and thus *'use_clutter'* is enabled, MinScore expects for each minimum score an additional value which determines what clutter score a potential match must at most have to be regarded as an instance of the model in the image. The runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*. Changing this second value does not influence the runtime. Note that the different shape models must have the same value for *'use_clutter'*.

If the maximum clutter is specified separately for each model, which is needed if also the minimum score is set for each model individually, MinScore must contain twice the number of elements as ModelIDs. In this case, the minimum score and the maximum clutter must be specified interleaved in MinScore. If, for example, two models are specified in ModelIDs, the minimum score is 0.9 for the first model and 0.8 for the second model, and the maximum clutter is 0.1 for the first model and 0.2 for the second model, MinScore $= [0.9, 0.1, 0.8, 0.2]$ must be selected.

NumMatches: The maximum number of instances to be found can be determined with NumMatches. If more than NumMatches instances with a score greater than MinScore are found in the image, only the best

NumMatches instances are returned. If fewer than NumMatches are found, only that number is returned, i.e., the parameter MinScore takes precedence over NumMatches. If all model instances exceeding MinScore in the image should be found, NumMatches must be set to *0*.

When tracking the matches through the image pyramid, on each level (except the top level), some less promising matches are rejected based on NumMatches. Thus, it is possible that some matches are rejected that would have had a higher score on the lowest pyramid level. Due to this, for example, the found match for NumMatches set to *1* might be different from the match with the highest score returned when setting NumMatches to *0* or > *1*.

If multiple objects with a similar score are expected, but only the one with the highest score should be returned, it might be preferable to raise NumMatches, and then select the match with the highest score.

In case that the shape models have been extended by clutter parameters using set_shape_model_clutter, NumMatches also considers the second value passed in MinScore: If more than NumMatches instances with a score greater than the first entry of MinScore and a clutter score smaller than the second entry of MinScore are found in the image, only the best NumMatches instances with respect to clutter are returned. Still, MinScore takes precedence over NumMatches and NumMatches must be set to *0* if all model instances fulfilling the conditions imposed by MinScore should be found. Please note that using clutter parameters, when tracking the matches through the image pyramid, no matches are rejected. Thus the runtime using clutter parameters will be at least as high as the runtime without clutter parameters and NumMatches set to *0*.

If NumMatches contains one element, find_shape_models returns the best NumMatches instances of the model irrespective of the type of the model. If, for example, two models are passed in ModelIDs and NumMatches $= 2$ is selected, it can happen that two instances of the first model and no instances of the second model, one instance of the first model and one instance of the second model, or no instances of the first model and two instances of the second model are returned. If, on the other hand, NumMatches contains multiple values, the number of instances returned of the different models corresponds to the number specified in the respective entry in NumMatches. If, for example, NumMatches $= [1, 1]$ is selected, one instance of the first model and one instance of the second model is returned.

MaxOverlap: If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter MaxOverlap determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than MaxOverlap only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see smallest_rectangle2) of the found instances. If MaxOverlap $= 0$, the found instances may not overlap at all, while for MaxOverlap $= 1$ all instances are returned.

If a single value is passed in MaxOverlap, the overlap is computed for all found instances of the different models, irrespective of the model type, i.e., instances of the same or of different models that overlap too much are eliminated. If, on the other hand, multiple values are passed in MaxOverlap, the overlap is only computed for found instances of the model that have the same model type, i.e., only instances of the same model that overlap too much are eliminated. In this mode, models of different types may overlap completely.

SubPixel: The parameter SubPixel determines whether the instances should be extracted with subpixel accuracy. If SubPixel is set to *'none'* (or *'false'* for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle resolution that was specified with create_shape_model. If SubPixel is set to *'interpolation'* (or *'true'*) the position as well as the rotation are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast to *'interpolation'*, this mode requires additional computation time. The different modes for least-squares adjustment (*'least_squares'*, *'least_squares_high'*, and *'least_squares_very_high'*) can be used to determine the accuracy with which the minimum distance is being searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, SubPixel should be set to *'interpolation'*. If least-squares adjustment is desired, *'least_squares'* should be chosen because this results in the best trade-off between runtime and accuracy.

Objects that are slightly deformed with respect to the model, in some cases cannot be found or are found but only with a low accuracy. For such objects it is possible to additionally pass a maximal allowable object deformation in the parameter SubPixel. The deformation must be specified in pixels. This can be done by passing the optional parameter value *'max_deformation '* followed by an integer value between *0* and *32* (in the same string), which specifies the maximum deformation. For example, if the shape of the object may be

deformed by up to 2 pixels with respect to the shape that is stored in the model, the value *'max_deformation 2'* must be passed in SubPixel in addition to the above described mode for the subpixel extraction, i.e., for example *['least_squares', 'max_deformation 2']*. Passing the value *'max_deformation 0'* corresponds to a search without allowing deformations, i.e., the behavior is the same as if no *'max_deformation '* is passed. Note that higher values for the maximum deformation often result in an increased runtime. Furthermore, the higher the deformation value is chosen, the higher is the risk of finding wrong model instances. Both problems mainly arise when searching for small objects or for objects with fine structures. This is because such kinds of objects for higher deformations lose their characteristic shape, which is important for a robust search. Also note that for higher deformations the accuracy of partially occluded objects might decrease if clutter is present close to the object. Consequently, the maximum deformation should be chosen as small as possible and only as high as necessary. Approximately rotationally symmetric objects may not be found if *'max_deformation'* and AngleExtent are both set to a value greater than 0. In that case, ambiguities may occur that cannot be resolved, and the match is rejected as false. If this happens, try to set either *'max_deformation'* or AngleExtent to 0, or adjust the model such that symmetries are reduced. When specifying a deformation higher than *0* the computation of the score depends on the chosen value for the subpixel extraction. In most cases, the score of a match changes if *'least_squares'*, *'least_squares_high'*, or *'least_squares_very_high'* (see above) is chosen for the subpixel extraction (in comparison to *'none'* or *'interpolation'*). Furthermore, if one of the least-squares adjustments is selected the score might increase when increasing the maximum deformation because then for the model points more corresponding image points can be found. To get a meaningful score value and to avoid erroneous matches, we recommend to always combine the allowance of a deformation with a least-squares adjustment.

If the subpixel extraction and/or the maximum object deformation is specified separately for each model, for each model passed in ModelIDs exactly one value for the subpixel extraction must be passed in SubPixel. After each value for the subpixel extraction optionally a second value can be passed, which describes the maximum object deformation of the corresponding mode. If for a certain model no value for the maximum object deformation is passed, the model is searched without taking deformations into account. For example, if two models are passed in ModelIDs and for the first model the subpixel extraction is set to *'interpolation'* and no object deformations are allowed and for the second model the subpixel extraction is set to *'least_squares'* and a maximum object deformation of *3* pixels is allowed, then the tuple *['interpolation', 'least_squares', 'max_deformation 3']* must be passed in SubPixel. Alternatively, the equivalent tuple *['interpolation', 'max_deformation 0', 'least_squares', 'max_deformation 3']* may be passed.

NumLevels: The number of pyramid levels used during the search is determined with NumLevels. If necessary, the number of levels is clipped to the range given when the shape model was created with create_shape_model. If NumLevels is set to *0*, the number of pyramid levels specified in create_shape_model is used.

In certain cases, the number of pyramid levels that was determined automatically with, for example, create_shape_model may be too high. The consequence may be that some matches that may have a high final score are rejected on the highest pyramid level and thus are not found. Instead of setting MinScore to a very low value to find all matches, it may be better to query the value of NumLevels with get_shape_model_params and then use a slightly lower value in find_shape_models. This approach is often better regarding the speed and robustness of the matching.

Optionally, NumLevels can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of *[4,2]* for NumLevels means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, SubPixel should be set to at least *'least_squares'*. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

If the lowest pyramid level is specified separately for each model, NumLevels must contain twice the number of elements as ModelIDs. In this case, the number of pyramid levels and the lowest pyramid level must be specified interleaved in NumLevels. If, for example, two models are specified in ModelIDs, the number of pyramid levels is 5 for the first model and 4 for the second model, and the lowest pyramid level is 2 for the first model and 1 for the second model, NumLevels $= [5, 2, 4, 1]$ must be selected. If exactly two models are specified in ModelIDs, a special case occurs. If in this case the lowest pyramid level is to

be specified, the number of pyramid levels and the lowest pyramid level must be specified explicitly for both models, even if they are identical, because specifying two values in NumLevels is interpreted as the explicit specification of the number of pyramid levels for the two models.

In input images of poor quality, i.e., in images that are, e.g., defocused, deformed, or noisy, often no instances of the shape model can be found on the lowest pyramid level. The reason for this behavior is the missing or deformed edge information which is a result of the poor image quality. Nevertheless, the edge information may be sufficient on higher pyramid levels. But keep in mind the above mentioned restrictions on accuracy and robustness if instances that were found on higher pyramid levels are used. The selection of the suitable pyramid level, i.e., the lowest pyramid level on which at least one instance of the shape model can be found, depends on the model and on the input image. This pyramid level may vary from image to image. To facilitate the matching on images of poor quality, the lowest pyramid level on which at least one instance of the model can be found can be determined automatically during the matching. To activate this mechanism, i.e., to use the so-called 'increased tolerance mode', the lowest pyramid level must be specified negatively in NumLevels. If, e.g., NumLevels is set to *[5,2,4,-1]*, the lowest pyramid level for the first model is set to 2. If no instance of the first model can be found on this pyramid level, no result will be returned for this model. For the second shape model, the lowest pyramid level is set to *-1*. Therefore, an instance of the shape model is searched on the pyramid level 1. If no instance of the second model can be found on this pyramid level, the lowest pyramid level is determined on which at least one instance of the model can be found. The instances of this pyramid level will then be returned.

If a model was adapted with adapt_shape_model_high_noise the estimated lowest pyramid level will be used by default. However, the user can override the estimated lowest pyramid level by providing explicitly the lowest pyramid level as described above.

Greediness: The parameter Greediness determines how "greedily" the search should be carried out. If Greediness $= 0$, a safe search heuristic is used, which always finds the model if it is visible in the image and the other parameters are set appropriately. However, the search will be relatively time consuming in this case. If Greediness $= 1$, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For Greediness $= 1$, the maximum search speed is achieved. In almost all cases, the shape model will always be found for Greediness $= 0.9$.

Output parameters in detail

Row, Column, and Angle: The position and rotation of the found instances of the model is returned in Row, Column, and Angle. The coordinates Row and Column are the coordinates of the origin of the shape model in the search image. By default, the origin is the center of gravity of the domain (region) of the image that was used to create the shape model with create_shape_model. A different origin can be set with set_shape_model_origin.

Note that the coordinates Row and Column do not exactly correspond to the position of the model in the search image. Thus, you cannot directly use them. Instead, the values are optimized for creating the transformation matrix with which you can use the results of the matching for various tasks, e.g., to align ROIs for other processing steps. The example given for find_shape_model shows how to create this matrix and use it to display the model at the found position in the search image.

Note also that for visualizing the model at the found position, also the procedure dev_display_shape_matching_results can be used.

Score: The score of each found instance is returned in Score. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

In case that the shape models have been extended by clutter parameters using set_shape_model_clutter, following the above values Score also returns the clutter scores of each found instance. If, for example, half of the clutter region is filled by clutter edges, the clutter score will equal 0.5. If, e.g., two instances are found, the score is 0.9 for the first instance and 0.8 for the second instance, and the clutter score is 0.2 for the first instance and 0.1 for the second instance, Score $= [0.9, 0.8, 0.2, 0.1]$ is returned. Please note that of all shape-based matching results, clutter scores are affected the most when a variation of illumination occurs.

Model: The type of the found instances of the models is returned in Model. The elements of Model are indices into the tuple ModelIDs, i.e., they can contain values from 0 to $|\text{ModelIDs}| - 1$. Hence, a value of 0 in an element of Model corresponds to an instance of the first model in ModelIDs.

Specifying a timeout

Using the operator set_shape_model_param you can specify a *'timeout'* for find_shape_models. If the shape models referenced by ModelIDs hold different values for *'timeout'*, find_shape_models uses the smallest one. If find_shape_models reaches this *'timeout'*, it terminates without results and returns the error code 9400 (H_ERR_TIMEOUT).

MinContrast with multiple models

Please note, that the different models that are given with the parameter ModelIDs should have been created with the same value of MinContrast. If they were created with different values for MinContrast, find_shape_models will use the smallest of these values.

Visualization of the results

To display the results found by shape-based matching, we highly recommend the usage of the procedure dev_display_shape_matching_results.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Input image in which the models should be found.

▷ **ModelIDs** (input_control) . . . . . shape_model(-array) ⇝ *handle*
Handle of the models.

▷ **AngleStart** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Smallest rotation of the models.
Default: -0.39
Suggested values: AngleStart ∈ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}

▷ **AngleExtent** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Extent of the rotation angles.
Default: 0.79
Suggested values: AngleExtent ∈ {6.29, 3.14, 1.57, 0.79, 0.39, 0.0}
Restriction: AngleExtent >= 0

▷ **MinScore** (input_control) . . . . . real(-array) ⇝ *real*
Minimum score of the instances of the models to be found.
Default: 0.5
Suggested values: MinScore ∈ {0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MinScore ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **NumMatches** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of instances of the models to be found (or 0 for all matches).
Default: 1
Suggested values: NumMatches ∈ {0, 1, 2, 3, 4, 5, 10, 20}

▷ **MaxOverlap** (input_control) . . . . . real(-array) ⇝ *real*
Maximum overlap of the instances of the models to be found.
Default: 0.5
Suggested values: MaxOverlap ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ MaxOverlap ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **SubPixel** (input_control) . . . . . string(-array) ⇝ *string*
Subpixel accuracy if not equal to *'none'*.
Default: 'least_squares'
Suggested values: SubPixel ∈ {'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high', 'max_deformation 1', 'max_deformation 2', 'max_deformation 3', 'max_deformation 4', 'max_deformation 5', 'max_deformation 6'}

▷ **NumLevels** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default: 0
List of values: NumLevels ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

▷ **Greediness** (input_control) . . . . . real(-array) ⇝ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default: 0.9
Suggested values: Greediness ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Value range: 0 ≤ Greediness ≤ 1
Minimum increment: 0.01
Recommended increment: 0.05

▷ **Row** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinate of the found instances of the models.

▷ **Column** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinate of the found instances of the models.

▷ **Angle** (output_control) . . . . . angle.rad-array ⇝ *real*
Rotation angle of the found instances of the models.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Score of the found instances of the models.

▷ **Model** (output_control) . . . . . integer-array ⇝ *integer*
Index of the found instances of the models.

Example

```
read_image (Image, 'pcb_focus/pcb_focus_telecentric_061')
gen_rectangle1 (ROI_0, 236, 241, 313, 321)
gen_circle (ROI_1, 281, 653, 41)
reduce_domain (Image, ROI_0, ImageReduced1)
reduce_domain (Image, ROI_1, ImageReduced2)

create_shape_model (ImageReduced1, 0, rad(-45), rad(180), 0, \
                    'none', 'use_polarity', 30, 10, ModelID1)
create_shape_model (ImageReduced2, 0, rad(-45), rad(180), 0, \
                    'none', 'use_polarity', 30, 10, ModelID2)
ModelIDs:=[ModelID1, ModelID2]
find_shape_models (Image, ModelIDs, rad(-45), rad(90), 0.7, [1,1], 0.5, \
                   'least_squares', 0, 1, Row, Column, Angle, Score, Model)
* Display results
dev_display_shape_matching_results (ModelIDs, 'red', Row, Column, Angle, \
                                    1, 1, Model)
```

Result

If the parameter values are correct, the operator find_shape_models returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

add_channels, create_shape_model, read_shape_model, set_shape_model_origin, set_shape_model_param, set_shape_model_clutter

Possible Successors

clear_shape_model

Alternatives

find_generic_shape_model

See also

set_system, get_system

Module

Matching

get_generic_shape_model_object (: Object : ModelID, GenParamName :)

Get iconic objects of the shape model.

The operator get_generic_shape_model_object reads out objects of the shape model ModelID and returns them in Object.

GenParamName specifies which iconic object will be returned. The following values are possible:

'clutter_region': The clutter region of the shape model.

'contours': Representation of the shape model ModelID for the bottom-most pyramid level as XLD contours. Thereby the contour is translated so that the origin of the model with the coordinates *'origin_row'* and *'origin_column'* (see set_generic_shape_model_param) is located at the origin of the pixel centered HALCON standard coordinate system (see Transformations / 2D Transformations). The contours can be used, for example, to visualize the found instances of the model in an image.

Parameters

▷ **Object** (output_object) ..... (multichannel-)object(-array) ⇝ *object*
Iconic object to be returned.

▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
Handle of the shape model.

▷ **GenParamName** (input_control) ..... string-array ⇝ *string*
Parameter names.
Default: 'clutter_region'
List of values: GenParamName ∈ {'clutter_region', 'contours'}

Result

If the parameters are valid, the operator get_generic_shape_model_object returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_generic_shape_model, set_generic_shape_model_object, train_generic_shape_model

Module

Matching

get_generic_shape_model_param (: : ModelID, GenParamName : GenParamValue)

Return the parameters of a shape model.

The operator get_generic_shape_model_param returns the parameters values GenParamValue of GenParamName for the shape model ModelID.

For an explanation of the settable GenParamValue, see set_generic_shape_model_param. Additional parameters whose value can only be retrieved:

- *'has_samples'*:
  States, whether the model contains training samples for extended parameter estimation (*'true'*) or not (*'false'*).
- *'needs_training'*:
  States, whether the model must be trained using train_generic_shape_model before it can be used by find_generic_shape_model (*'true'*) or not (*'false'*).

- *'scale_type'*:
  A string which describes the type of scaling of the shape model.

The operator `get_generic_shape_model_param` returns the value as it is used by the shape model. As certain parameters may be modified after calling `train_generic_shape_model`, their value can differ from the set value. This applies particularly to parameters whose value is to be determined automatically, e.g., with values set to *'auto'*. The (unmodified) set value of a `GenParamName` can be retrieved by adding the suffix *'_param'* to its name.

Example: After calling `train_generic_shape_model`, *'num_levels'* returns the estimated value and *'num_levels_param'* returns the set value, e.g., its value set by default or the one set by the user.

Exceptions to this rule are:

- *'greediness'* when *'extended_parameter_estimation'* is set to *'per_level'*.
- *'min_score'* when *'extended_parameter_estimation'* is set to *'per_level'*.

Parameters

▷ **`ModelID`** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the shape model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name-array ⇝ *string*
Parameter name.
Default: 'min_score'
List of values: `GenParamName` ∈ {'angle_end', 'angle_start', 'angle_step', 'border_shape_models', 'clutter_border_mode', 'clutter_contrast', 'clutter_hom_mat_2d', 'contrast_high', 'contrast_low', 'extended_parameter_estimation', 'greediness', 'has_samples', 'iso_scale_max', 'iso_scale_min', 'iso_scale_step', 'max_clutter', 'max_deformation', 'max_overlap', 'max_overlap_global_enable', 'metric', 'min_contrast', 'min_score', 'min_size', 'model_cache', 'model_identifier', 'needs_training', 'num_levels', 'num_matches', 'optimization', 'origin_column', 'origin_row', 'prepare_contours_for_visualization', 'prepare_clutter_region_for_visualization', 'pyramid_level_highest', 'pyramid_level_lowest', 'pyramid_level_robust_tracking', 'restrict_iso_scale_max', 'restrict_iso_scale_min', 'restrict_scale_column_max', 'restrict_scale_column_min', 'restrict_scale_row_max', 'restrict_scale_row_min', 'scale_column_max', 'scale_column_min', 'scale_column_step', 'scale_row_max', 'scale_row_min', 'scale_row_step', 'scale_type', 'strict_boundaries', 'subpixel', 'time_measurement', 'timeout', 'use_clutter'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Parameter value.

Result

If the parameters are valid, the operator `get_generic_shape_model_param` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_generic_shape_model`, `set_generic_shape_model_param`, `train_generic_shape_model`

Module

Matching

get_generic_shape_model_result (: : MatchResultID, MatchSelector, GenParamName : GenParamValue)

Get alphanumerical values from a shape matching result.

get_generic_shape_model_result returns alphanumerical results of shape-based matching in GenParamValue. The matches are contained in MatchResultID and sorted by descending score.

The parameter MatchSelector is used to select from which matches the result values shall be returned. The sorting remains according to their score. Hence, it is independent of the order of the selection criteria passed in MatchSelector. Furthermore, a matching result is only returned once even if it meets more than one selection criterion.

MatchSelector processes selection criteria of the following types or combinations of them:

- *'all'*:
  All matching results are selected.
- *'best'*:
  The match with the highest score is selected.
- Result index:
  By setting MatchSelector to an integer, the index of the match in MatchResultID is determined.
  Value range: 0 to n-1 (with n as the total number of results).
  If the index is outside the valid range, an exception is raised.
- Shape model identifier:
  All matches found with a specific shape model are selected. For this you can pass the respective model identifier *'model_identifier'* as a string to MatchSelector.
- Shape model handle:
  All matches found with a specific shape model are selected. For this you can pass the respective shape model handle to MatchSelector.

If MatchSelector is no valid selection criterion, an exception is raised.

GenParamName specifies which alphanumerical result value will be returned. The following values are possible:

- *'num_match_result'*:
  Number of matches. The accumulated number of matches is returned in case multiple selections are made in MatchSelector.
- *'model_identifier'*:
  String identifying the model the match was found with.
- *'score'*:
  Score of the found match. It is a number between 0 and 1 and an approximate measure of how good the model is found in the image. For further information on scores, see "Solution Guide II-B - Matching", Chapter 'General Topics', section 'About the Score'.
- *'angle'*:
  Rotation angle of the found match.
  Value range: $-3.14 \leq$ *'angle'* ≤ 3.14 ($=\pi$).
- *'row'*:
  Row coordinate of the found match in the search image. The coordinate relates to the origin of the shape model.
- *'column'*:
  Column coordinate of the found match in the search image. The coordinate relates to the origin of the shape model.
- *'scale_row'*:
  Scale of the found match in row direction.
- *'scale_column'*:
  Scale of the found match in column direction.

- *'hom_mat_2d'*:
  Homographic transformation matrix to transform the contours of the shape model from its origin to the found location in the search image. Position, rotation, and scaling in row and column direction are taken into account. For more information on transformations with homographic matrices see `affine_trans_point_2d`.
- *'clutter_score'*:
  Clutter score of the found match. It is a number between 0 and 1 and an approximate measure of how much clutter edges are present in the clutter region.
- *'clutter_hom_mat_2d'*:
  Homographic transformation matrix to transform the clutter region of the shape model from its origin to the found location in the search image. Position, rotation, and scaling in row and column direction are taken into account.
- *'match'*:
  Dictionary containing the coordinate *'row'*, *'column'*, the rotation angle *'angle'*, the scale *'scale_row'*, *'scale_column'*, the score *'score'*, and the homographic transformation *'hom_mat_2d'* of the found match. Additionally, the two clutter-specific results *'clutter_score'* and *'clutter_hom_mat_2d'* are contained in the dictionary if clutter has been trained and used (see `set_generic_shape_model_object` with *'clutter_region'*).
- *'time_measurement'*:
  A dictionary containing the different time measurements that where conducted during the search. Note that their sum is less then the total time spent on the operator call. The dictionary contains time measurements for the four main steps of the search pipeline *'pipeline'*:
  - *'pyramid'*:
    Time spent constructing the image pyramid.
  - *'top_level'*:
    Time spent searching for candidates on highest pyramid level.
  - *'tracking'*:
    Time spent tracking candidates from the highest pyramid level to the lowest.
  - *'subpixel_refinement'*:
    Time spent conducting the refinement at the end of the search.

  Restriction: `MatchSelector` must be set to *'all'*.

Note that for values regarding the location of the matches coordinates are given as edge centered coordinates. For more information on edge centered coordinates see the chapter Transformations / 2D Transformations.

Parameters

▷ **`MatchResultID`** (input_control) . . . . . generic_shape_model_result ⇝ *handle*
Handle of the shape model matches.

▷ **`MatchSelector`** (input_control) . . . . . string-array ⇝ *string* / integer
Selector for the matching results to be queried.
Default: 'all'
List of values: `MatchSelector` ∈ {'all', 'best'}

▷ **`GenParamName`** (input_control) . . . . . attribute.name ⇝ *string*
Name of the queried result parameter.
Default: 'score'
List of values: `GenParamName` ∈ {'num_match_result', 'model_identifier', 'score', 'angle', 'row', 'column', 'scale_row', 'scale_column', 'hom_mat_2d', 'clutter_score', 'clutter_hom_mat_2d', 'match', 'time_measurement'}

▷ **`GenParamValue`** (output_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Value of the queried result parameter.

Result

If the parameters are valid, the operator `get_generic_shape_model_result` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`find_generic_shape_model`

Module

Matching

get_generic_shape_model_result_object (: Objects : MatchResultID, MatchSelector, GenParamName :)

Get objects from a shape matching result.

`get_generic_shape_model_result_object` returns iconic results of shape-based matching in `Objects`. The results are contained in `MatchResultID`.

The parameter `MatchSelector` is used to select from which matches the result values shall be returned. For more information and the list of supported values for `MatchSelector` see `get_generic_shape_model_result`.

`GenParamName` specifies which object result value will be returned. The following values are possible:

'contours': Model contours transformed according to the matching result.

'clutter_region': Region where no clutter should occur.

Parameters

▷ **Objects** (output_object) ..... (multichannel-)object(-array) ⇝ *object*
Objects of the queried result.

▷ **MatchResultID** (input_control) ..... generic_shape_model_result ⇝ *handle*
Handle of the shape model matches.

▷ **MatchSelector** (input_control) ..... string-array ⇝ *string* / integer
Selector for the matching results to be queried.
Default: 'all'
List of values: `MatchSelector` ∈ {'all', 'best'}

▷ **GenParamName** (input_control) ..... string ⇝ *string*
Name of the queried result object.
Default: 'contours'
List of values: `GenParamName` ∈ {'contours', 'clutter_region'}

Result

If the parameters are valid, the operator `get_generic_shape_model_result_object` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`find_generic_shape_model`

Module

Matching

get_shape_model_clutter (: ClutterRegion : ModelID, GenParamName : GenParamValue, HomMat2D, ClutterContrast)

Get the clutter parameters of a shape model.

The operator get_shape_model_clutter can be used to inspect clutter parameters of the shape model ModelID, which have been set previously using set_shape_model_clutter. Also, the value of *'use_clutter'*, which can be set using set_shape_model_param, can be queried by setting GenParamName to *'use_clutter'*. For a description of the parameters, please see set_shape_model_clutter and set_shape_model_param.

Parameters

- ▷ **ClutterRegion** (output_object) ..... region ⇝ *object*
  Region where no clutter should occur.
- ▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
  Handle of the model.
- ▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
  Parameter names.
  Default: 'use_clutter'
  List of values: GenParamName ∈ {'use_clutter', 'clutter_border_mode'}
- ▷ **GenParamValue** (output_control) ..... attribute.name(-array) ⇝ *string* / real / integer
  Parameter values.
- ▷ **HomMat2D** (output_control) ..... hom_mat2d ⇝ *real*
  Transformation matrix.
- ▷ **ClutterContrast** (output_control) ..... number ⇝ *integer*
  Minimum contrast of clutter in the search images.

Result

If the parameters are valid, the operator get_shape_model_clutter returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_shape_model_clutter, read_shape_model, deserialize_shape_model

Possible Successors

find_shape_model, find_scaled_shape_model, find_aniso_shape_model, find_shape_models, find_scaled_shape_models, find_aniso_shape_models

See also

create_shape_model, create_scaled_shape_model, create_aniso_shape_model, set_shape_model_param

Module

Matching

get_shape_model_contours (: ModelContours : ModelID, Level :)

Return the contour representation of a shape model.

The operator get_shape_model_contours returns a representation of the shape model ModelID as XLD contours in ModelContours. The parameter Level determines for which pyramid level of the model the contour representation should be returned. The contours can be used, for example, to visualize the found instances of the model in an image. It should be noted that the position of ModelContours is normalized such that the reference point of the model (see set_shape_model_origin) lies at the pixel position $(0, 0)$. Hence, the

contours simply need to be translated to the found position in the image (and possibly rotated and scaled around this point).

Parameters

▷ **ModelContours** (output_object) ..... xld_cont-array ⇝ *object*
Contour representation of the shape model.

▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
Handle of the model.

▷ **Level** (input_control) ..... integer ⇝ *integer*
Pyramid level for which the contour representation should be returned.
Default: 1
Suggested values: Level ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Restriction: Level >= 1

Result

If the handle of the model is valid, the operator get_shape_model_contours returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_shape_model, create_scaled_shape_model, create_aniso_shape_model, read_shape_model

Alternatives

get_generic_shape_model_object

See also

find_shape_model, find_scaled_shape_model, find_aniso_shape_model, find_shape_models, find_scaled_shape_models, find_aniso_shape_models

Module

Matching

get_shape_model_origin (: : ModelID : Row, Column)

Return the origin (reference point) of a shape model.

The operator get_shape_model_origin returns the origin (reference point) of the shape model ModelID. The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the shape model with create_shape_model, create_scaled_shape_model, or create_aniso_shape_model. Hence, an origin of (0,0) means that the center of gravity of the domain of the model image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity.

Parameters

▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
Handle of the model.

▷ **Row** (output_control) ..... point.y ⇝ *real*
Row coordinate of the origin of the shape model.

▷ **Column** (output_control) ..... point.x ⇝ *real*
Column coordinate of the origin of the shape model.

Result

If the handle of the model is valid, the operator get_shape_model_origin returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_generic_shape_model`, `read_shape_model`, `set_shape_model_origin`

Possible Successors

`find_generic_shape_model`

See also

`area_center`

Module

Matching

get_shape_model_params (: : ModelID : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleMin, ScaleMax, ScaleStep, Metric, MinContrast)

Return the parameters of a shape model.

The operator `get_shape_model_params` returns the parameters of the shape model `ModelID` that were used to create it using `create_shape_model`, `create_scaled_shape_model`, or `create_aniso_shape_model`. In particular, this output can be used to check the parameters `NumLevels`, `AngleStep`, `ScaleStep`, and `MinContrast` if they were determined automatically during the model creation with `create_shape_model`, `create_scaled_shape_model`, or `create_aniso_shape_model`.

If the `ModelID` was adapted with `adapt_shape_model_high_noise`, NumLevels contains two values: the highest pyramid level and the estimated lowest pyramid level.

If the shape model was created using `create_shape_model` or `create_scaled_shape_model` a single value is returned in `ScaleMin`, `ScaleMax`, and `ScaleStep`. This parameters corresponds to the isotropic scaling parameters of the shape model. If the shape model was created using `create_aniso_shape_model` two values each are returned in the above three parameters. Here, the first value of the respective parameter refers to the scaling in the row direction, while the second value refers to the scaling in the column direction.

Note that the parameters `Optimization` and `Contrast` that also can be determined automatically during the model creation cannot be queried by using `get_shape_model_params`. If their value is of interest `determine_shape_model_params` should be used instead.

Parameters

▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
Handle of the model.

▷ **NumLevels** (output_control) ..... integer(-array) ⇝ *integer*
Number of pyramid levels.

▷ **AngleStart** (output_control) ..... angle.rad ⇝ *real*
Smallest rotation of the pattern.

▷ **AngleExtent** (output_control) ..... angle.rad ⇝ *real*
Extent of the rotation angles.
Assertion: AngleExtent >= 0

▷ **AngleStep** (output_control) ..... angle.rad ⇝ *real*
Step length of the angles (resolution).
Assertion: AngleStep >= 0 && AngleStep <= pi / 2

▷ **ScaleMin** (output_control) ..... number(-array) ⇝ *real*
Minimum scale of the pattern.
Assertion: ScaleMin > 0

▷ **ScaleMax** (output_control) ..... number(-array) ⇝ *real*
Maximum scale of the pattern.
Assertion: ScaleMax >= ScaleMin

▷ **ScaleStep** (output_control) ..... number(-array) ⇝ *real*
Scale step length (resolution).
Assertion: ScaleStep >= 0

▷ **Metric** (output_control) ..... string ⇝ *string*
Match metric.

▷ **MinContrast** (output_control) ..... number ⇝ *integer*
Minimum contrast of the objects in the search images.

Result

If the handle of the model is valid, the operator `get_shape_model_params` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_shape_model`, `create_scaled_shape_model`, `create_aniso_shape_model`, `read_shape_model`

Alternatives

`get_generic_shape_model_param`

See also

`find_shape_model`, `find_scaled_shape_model`, `find_aniso_shape_model`, `find_shape_models`, `find_scaled_shape_models`, `find_aniso_shape_models`

Module

Matching

inspect_shape_model (Image : ModelImages, ModelRegions : NumLevels, Contrast :)

Create the representation of a shape model.

`inspect_shape_model` creates a representation of a shape model. The operator is particularly useful in order to determine the parameters `NumLevels` and `Contrast`, which are used in `create_shape_model`, `create_scaled_shape_model`, or `create_aniso_shape_model`, quickly and conveniently. The representation of the model is created on multiple image pyramid levels, where the number of levels is determined by `NumLevels`. In contrast to `create_shape_model`, `create_scaled_shape_model`, and `create_aniso_shape_model`, the model is only created for the rotation and scale of the object in the input image, i.e., 0° and 1. As output, `inspect_shape_model` creates an image object `ModelImages` containing the images of the individual levels of the image pyramid as well as a region in `ModelRegions` for each pyramid level that represents the model at the respective pyramid level. The individual objects can be accessed with `select_obj`. If the input image `Image` has one channel the representation of the model is created with the method that is used in `create_shape_model`, `create_scaled_shape_model` or `create_aniso_shape_model` for the metrics *'use_polarity'*, *'ignore_global_polarity'*, and *'ignore_local_polarity'*. If the input image has more than one channel the representation is created with the method that is used for the metric *'ignore_color_polarity'*. As described for `create_shape_model`, `create_scaled_shape_model`, and `create_aniso_shape_model`, the number of pyramid levels should be chosen as large as possible, while taking into account that the model must be recognizable on the highest pyramid level and must have enough model points. The parameter `Contrast` should be chosen such that only the significant features of the template object are used for the model. `Contrast` can also contain a tuple with two values. In this case, the model is segmented using a method similar to the hysteresis threshold method used in `edges_image`. Here, the first element of the tuple determines the lower threshold, while the second element determines the upper threshold. For more information about the hysteresis threshold method, see `hysteresis_threshold`. Optionally, `Contrast` can contain a third value as the last element of the tuple. This value determines a threshold for the selection of significant model components based

on the size of the components, i.e., components that have fewer points than the minimum size thus specified are suppressed. As the minimum size is applied on the extent of the components, the derived model contours can still be smaller than the specified minimum size. This threshold for the minimum size is divided by two for each successive pyramid level. If small model components should be suppressed, but hysteresis thresholding should not be performed, nevertheless three values must be specified in `Contrast`. In this case, the first two values can simply be set to identical values. In its typical use, `inspect_shape_model` is called interactively multiple times with different parameters for `NumLevels` and `Contrast`, until a satisfactory model is obtained. After this, `create_shape_model`, `create_scaled_shape_model`, or `create_aniso_shape_model` are called with the parameters thus obtained.

Parameters

- ▷ **`Image`** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2
  Input image.
- ▷ **`ModelImages`** (output_object) . . . . . (multichannel-)image-array ⇝ *object :* byte / uint2
  Image pyramid of the input image
- ▷ **`ModelRegions`** (output_object) . . . . . region-array ⇝ *object*
  Model region pyramid
- ▷ **`NumLevels`** (input_control) . . . . . integer ⇝ *integer*
  Number of pyramid levels.
  Default: 4
  List of values: `NumLevels` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
- ▷ **`Contrast`** (input_control) . . . . . number(-array) ⇝ *integer*
  Threshold or hysteresis thresholds for the contrast of the object in the image and optionally minimum size of the object parts.
  Default: 30
  Suggested values: `Contrast` ∈ {10, 20, 30, 40, 60, 80, 100, 120, 140, 160}

Result

If the parameters are valid, the operator `inspect_shape_model` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`reduce_domain`

Possible Successors

`select_obj`

See also

`set_generic_shape_model_param`

Module

Foundation

`read_shape_model ( : : FileName : ModelID )`

Read a shape model from a file.

The operator `read_shape_model` reads a shape model, which has been written with `write_shape_model`, from the file `FileName`. The default HALCON file extension for the shape model is 'shm'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .shm

▷ **ModelID** (output_control) . . . . . shape_model ⇝ *handle*
Handle of the model.

Result

If the file name is valid, the operator read_shape_model returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

find_generic_shape_model

Module

Matching

serialize_shape_model (: : ModelID : SerializedItemHandle)

Serialize a shape model.

serialize_shape_model serializes the data of a shape model (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_shape_model is converted to a serialized item. The shape model is defined by the handle ModelID. The serialized shape model is returned by the handle SerializedItemHandle and can be deserialized by deserialize_shape_model.

Parameters

▷ **ModelID** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the model.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_shape_model returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_generic_shape_model

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_shape_model

Module

Matching

set_generic_shape_model_object (Object : : ModelID, GenParamName :)

Set iconic objects to the shape model.

The operator set_generic_shape_model_object sets parameters of the shape model ModelID based on the given Object.

The parameter GenParamName determines for which type of setting the object is used. The following values are supported:

- *'high_noise_sample'*:
  Estimate and adapt the shape model parameters to cope better with high noise images. The parameter estimation is done based on the sample search image Object. As a consequence the domain of Object should contain the model and surrounding background showing noise that is typical for search images.
  The operator modifies the following parameters:
  - *'pyramid_level_lowest'*
- *'clutter_region'*:
  Set the clutter region for the shape model. If *'clutter_region'* is set, the value of *'use_clutter'* is automatically set to *'true'*.
  In case the clutter region is set before the training, it can be specified with regard to the template, and the transformation is estimated automatically. In case the clutter region is set after the training, it can be specified using a model instance found on a search image. This also requires setting its transformation using set_generic_shape_model_param with *'clutter_hom_mat_2d'*.
  Note that the *'clutter_region'* must not intersect with the final model contours that are generated by train_generic_shape_model.
  The clutter region may not contain negative coordinates. I.e., when defining clutter regions for shape models created from XLD contours, move the XLD template contour (and wanted clutter region) into the positive image area, e.g., using affine_trans_contour_xld. Alternatively use set_generic_shape_model_param with *'clutter_hom_mat_2d'* after training.

Parameters

▷ **Object** (input_object) . . . . . (multichannel-)object(-array) ⇝ *object*
Iconic object to be set.

▷ **ModelID** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the shape model.

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Parameter name.
Default: 'high_noise_sample'
List of values: GenParamName ∈ {'clutter_region', 'high_noise_sample'}

Result

If the parameters are valid, the operator set_generic_shape_model_object returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_generic_shape_model

Possible Successors

train_generic_shape_model, find_generic_shape_model

Module

Matching

set_generic_shape_model_param (: : ModelID, GenParamName, GenParamValue :)

Set selected parameters of the shape model.

The operator set_generic_shape_model_param sets the parameters GenParamName of the shape model ModelID to the values given in GenParamValue.

Various aspects of the matching process can be controlled by the parameters that can be selected by GenParamName. In the descriptions below they are grouped into the following categories (where the category represents the main purpose but not necessarily the only purpose of the parameter listed within):

Modifying the Model

Finding Model Instances

1. Modify the Instance
2. Sort out Found Matches
3. Gray Value Treatment
4. Refinement
5. Image Pyramid
6. Clutter
7. Regarding the Output

Computational Resources During a Search

Extended Parameter Estimation

In the following we list the settable parameters for the different categories and specify cases in which parameter values are modified automatically.

Modifying the Model

In this paragraph we list and explain parameters modifying the model ModelID itself. Changing a corresponding value makes it necessary to train the adjusted model before the matching (see train_generic_shape_model). The supported parameters GenParamName are:

Contrast The contrast is a measure for local gray value differences that are used to distinguish the object from the background or different parts of the object from each other. For the model only those pixels are selected which show the asked contrast, i.e., gray value difference to neighboring pixels. This asked contrast can be a simple threshold or a hysteresis threshold, depending on the values set for *'contrast_low'* and *'contrast_high'*:

- Simple threshold:
  - Both are set to the same value which is not *'auto'*.
  - One of the parameters is set to *'auto'* while the other one is set to an other value.
- Hysteresis threshold:
  - Both are set to different values which are both not *'auto'*.
  - Both are set to *'auto'*.

  In these cases the model is segmented using a method similar to the hysteresis threshold method used in edges_image. For more information about the hysteresis threshold method, see hysteresis_threshold.

Note, such a contrast is not used in case of training with XLD. For more information about contrast and its impact on shape model matching, we refer to the "Solution Guide II-B - Matching".

In certain cases, it might happen that the automatic determination of the contrast thresholds is not sufficient. For example, a manual setting of these parameters should be preferred if certain model components

should be included or suppressed due to application-specific reasons or if the object features several different contrasts. The contrast thresholds should be verified using `inspect_shape_model` before calling `train_generic_shape_model`.

- *'contrast_high'*:
  Determines together with *'contrast_low'* the gray value difference a pixel must have to be considered as being part of the model, see the explanation above. In case of a hysteresis threshold this parameter sets the maximum.
  Suggested values: *'auto'*, *90*, *80*.
  Restriction: Not used in case of training with XLD.
  Default: *'auto'*.
- *'contrast_low'*:
  Determines together with *'contrast_high'* the gray value difference a pixel must have to be considered as being part of the model, see the explanation above. In case of a hysteresis threshold this parameter sets the minimum.
  Suggested values: *'auto'*, *10*, *20*.
  Restriction: Not used in case of training with XLD.
  Default: *'auto'*.

Rotation Shape models are created with a rotation range starting at *0* and ending at *6.28* (=2π), which can be restricted during the search. The parameter space, thus discretization steps for the angles can be modified using:

- *'angle_step'*:
  Determines the step length within the selected range of angles. Its value should be chosen based on the size of the object. Smaller models do not have many different discrete rotations in the image, and hence it is recommended to choose larger values for smaller models. If set to *'auto'*, its value is estimated during the call of `train_generic_shape_model`. The set value is clipped to a maximal value of *0.20* (=$\frac{\pi}{16}$).
  Suggested values: *'auto'*, *0.1*, *0.2*.
  Attention: Modifications necessitate a model (re)training.
  Default: *'auto'*.

Scaling The parameter space, thus discretization steps for the scaling can be modified using scaling parameters. Which ones can be set depends on the type of scaling, see the overview in the following table. A scaling value of *1.0* corresponds to the original size of the model in the according direction.

GenParamName	isotropic scaling	anisotropic scaling
'iso_scale_max'	x	
'iso_scale_min'	x	
'iso_scale_step'	x	
'scale_row_max'		x
'scale_row_min'		x
'scale_row_step'		x
'scale_column_max'		x
'scale_column_min'		x
'scale_column_step'		x

- *'iso_scale_max'*:
  Determines the maximum of the range of possible isotropic scaling for which the model is searched.
  Suggested values: *1.0*, *1.1*, *1.2*.
  Default: *1.0*.
- *'iso_scale_min'*:
  Determines the minimum of the range of possible isotropic scaling for which the model is searched.
  Suggested values: *1.0*, *0.9*, *0.8*.
  Default: *1.0*.
- *'iso_scale_step'*:
  Determines the step length within the selected range of isotropic scales. The value *'auto'* means its value is estimated during the call of `train_generic_shape_model`.

- *'scale_row_max'*:
  Determines the maximum of the range of possible scaling in row direction for which the model is searched.
  Suggested values: *1.0*, *1.1*, *1.2*.
  Default: *1.0*.
- *'scale_row_min'*:
  Determines the minimum of the range of possible scaling in row direction for which the model is searched.
  Suggested values: *1.0*, *0.9*, *0.8*.
  Default: *1.0*.
- *'scale_row_step'*:
  Determines the step length within the selected range of scales in row direction. The value *'auto'* means its value is estimated during the call of `train_generic_shape_model`.
- *'scale_column_max'*:
  Determines the maximum of the range of possible scaling in column direction for which the model is searched.
  Suggested values: *1.0*, *1.1*, *1.2*.
  Default: *1.0*.
- *'scale_column_min'*:
  Determines the minimum of the range of possible scaling in column direction for which the model is searched.
  Suggested values: *1.0*, *0.9*, *0.8*.
  Default: *1.0*.
- *'scale_column_step'*:
  Determines the step length within the selected range of scales in column direction. The value *'auto'* means its value is estimated during the call of `train_generic_shape_model`.

Metric

- *'metric'*:
  Determines the conditions under which the model is recognized in the image. Supported values for *'metric'* are:
  - *'auto'*:
    The value is determined during the call of `train_generic_shape_model` depending if the training is done using an image (*'use_polarity'*) or XLDs (*'ignore_local_polarity'*).
  - *'use_polarity'*:
    An instance in the image and the model must have the same contrast.
    Example: The model has been created with a bright object on a dark background. Only instances are found if they are also brighter than the background.
    Restriction: Only applied to single-channel images. In case of multichannel images, only the first channel is used (without error message).
  - *'ignore_global_polarity'*:
    An instance is found in the image also if the contrast reverses globally. This mode increases the runtime of `find_generic_shape_model` slightly compared to *'use_polarity'*.
    Example: The model has been created with a bright object on a dark background. Instances are found if they are brighter or darker than the background.
    Restriction: Only applied to single-channel images. In case of multichannel images, only the first channel is used (without error message).
  - *'ignore_local_polarity'*:
    An instance is found even if the contrast changes locally. This mode increases the runtime of `find_generic_shape_model` significantly compared to *'use_polarity'*.
    It is usually recommendable to create several models that reflect the possible contrast variations of the object and to match them simultaneously.
    Example: The model or instance contains a part with medium gray values, within which either darker or brighter sub-objects may lie.
    Restriction: Only applied to single-channel images. In case of multichannel images, only the first channel is used (without error message).
  - *'ignore_color_polarity'*:

An instance is found even if the color contrast changes locally. This mode can increase the runtime of `find_generic_shape_model` significantly compared to *'use_polarity'*.
This metric can be used for images of an arbitrary number of channels, which do not need to contain a spectral subdivision of the light (like in an RGB image). For single-channel images it has the same effect as *'ignore_local_polarity'*. The number of channels is allowed to differ from the training with `train_generic_shape_model` to the search with `find_generic_shape_model`.
Example: Instances are found even if parts of the object can change their color, e.g., from red to green.

Restriction: Using XLDs, resetting the metric requires that the edge-direction information is available as XLD contour attribute. For more information see `set_shape_model_metric`.
Default: *'auto'*.

Size

- *'min_size'*:
  Determines the minimum number of points a model component must have in order to be considered. Thus, components having fewer points than *'min_size'* are suppressed. As *'min_size'* is applied on the extent of the components, the derived model contours can still be smaller than the specified minimum size. This threshold for the minimum size is divided by two for each successive pyramid level. The effect of this parameter can be checked in advance with `inspect_shape_model`.
  If set to *'auto'*, its value is estimated during the call of `train_generic_shape_model`.
  Restriction: Not used in case of training with XLD.
  Default: *'auto'*.
- *'optimization'*:
  Determines the number of points by which the model is reduced. This reduction can be useful for particularly large models. Possible values:
  - *'auto'*:
    The number of points is automatically determined by `train_generic_shape_model`.
  - *'none'*:
    All model points are stored.
  - *'point_reduction_low'*:
    The number of points is reduced to roughly $\frac{1}{2}$.
  - *'point_reduction_medium'*:
    The number of points is reduced to roughly $\frac{1}{3}$.
  - *'point_reduction_high'*:
    The number of points is reduced to roughly $\frac{1}{4}$.

  In case the number of model points is reduced it may be necessary to set the parameter *'greediness'* to a smaller value, e.g., *0.7* or *0.8*. Note, for small models reducing the number of model points usually does not speed up the search as more potential instances will be examined.
  Default: *'auto'*.

Image Pyramid

- *'num_levels'*:
  Number of considered image pyramid levels. Its value must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. This can be checked using the output of `inspect_shape_model`. It is advisable to set a value as large as possible as it reduces the runtime to find an instance significantly.
  If set to *'auto'*, its value is estimated during the call of `train_generic_shape_model`.
  It can happen that the automatically determined value does not suit. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or only with very low values for the parameters *'min_score'* or *'greediness'*. If the number of pyramid levels is chosen too small, the time required to find the model in `find_generic_shape_model` may increase. In both cases, the number of pyramid levels should be selected using the output of `inspect_shape_model`.
  Default: *'auto'*.

Naming

- *'model_identifier'*:
  Overwrites the current identifier string with GenParamValue. The given identifier should be unique to avoid ambiguities at e.g., a later match retrieval with get_generic_shape_model_result.
  Restriction: *'model_identifier'* must be a non-empty string. The following words are reserved and cannot be used as identifier names: *'all'*, *'best'*.
  Default: The string which is automatically set at the shape model creation.

Finding Model Instances

In this paragraph we list parameters modifying the search of a ModelID. Modifying these parameters does not necessitate a model training, except they are set from a not-estimated value to a value leading to an automatic estimation during train_generic_shape_model. The descriptions of these parameters contain a corresponding note. The supported parameters GenParamName are:

1. **Modify the Instance**

Rotation

- *'angle_start'*:
  Determines the start of the range of possible rotations for which the model is searched.
  Suggested values: *0.0*, *-3.14*, *3.14*.
  Default: *0.0*.
- *'angle_end'*:
  Determines the end of the range of possible rotations for which the model is searched.
  Suggested values: *0.0*, *3.14*, *6.28*.
  Default: *6.28* (= 2π).

Example: 'angle_start' = '-rad(10)', *'angle_end' = 'rad(10)'* searches for matches in the range from [rad(350), rad(360)] to [0, rad(10)].

Scaling The scaling range specified for the trained model can be restricted further for the search. For these parameters the value *'auto'* means that they do not restrict the according range.
Note, that these restrictions can only be applied if the parameter space is further limited. Thus, a smaller maximum and a higher minimum scaling parameter needs to be set. Otherwise, the model has to be trained anew with adjusted scaling parameters.

GenParamName	isotropic scaling	anisotropic scaling
'restrict_iso_scale_max'	x	
'restrict_iso_scale_min'	x	
'restrict_scale_row_max'		x
'restrict_scale_row_min'		x
'restrict_scale_column_max'		x
'restrict_scale_column_min'		x

- *'restrict_iso_scale_max'*:
  Determines a possibly restricted maximum of the range of possible scaling for which the model is searched.
  Suggested values: *'auto'*, *1.1*, *1.2*.
  Default: *'auto'*.
- *'restrict_iso_scale_min'*:
  Determines a possibly restricted minimum of the range of possible scaling for which the model is searched.
  Suggested values: *'auto'*, *0.9*, *0.8*.
  Default: *'auto'*.
- *'restrict_scale_row_max'*:
  Determines a possibly restricted maximum of the range of possible scaling in row direction for which the model is searched.
  Suggested values: *'auto'*, *1.1*, *1.2*.
  Default: *'auto'*.
- *'restrict_scale_row_min'*:

Determines a possibly restricted minimum of the range of possible scaling in row direction for which the model is searched.
Suggested values: *'auto'*, *0.9*, *0.8*.
Default: *'auto'*.

- *'restrict_scale_column_max'*:
  Determines a possibly restricted maximum of the range of possible scaling in column direction for which the model is searched.
  Suggested values: *'auto'*, *1.1*, *1.2*.
  Default: *'auto'*.
- *'restrict_scale_column_min'*:
  Determines a possibly restricted minimum of the range of possible scaling in column direction for which the model is searched.
  Suggested values: *'auto'*, *0.9*, *0.8*.
  Default: *'auto'*.

Deformation

- *'max_deformation'*:
  Determines by how much an object is allowed to deviate from the model in order to be considered as a match. The maximal allowable object deformation is specified in pixels. A value of *0* means no deformation is allowed during the search.
  Example: An object with a shape deformed by up to 2 pixels with respect to the shape of the model can be found with *'max_deformation'* = *2*.
  Increasing the value for *'max_deformation'* often results in an increased runtime.
  Furthermore, increasing the value for *'max_deformation'* increases the risk of obtaining unwanted matches. This is especially the case for small objects or objects with fine structures because they lose their characteristic shape, which is important for a robust search.
  Interplay with *'subpixel'*: When deformation is allowed (thus, a value larger *0* is set), the score computation depends on the possible subpixel refinement. In most cases the score changes if a least-squares adjustment is set. Using an adjustment the score might increase when increasing the maximum deformation because then for the model points more corresponding image points can be found.
  Default: *0*.

2. **Sort out Found Matches**

- *'min_score'*:
  Determines the minimum score a potential match must have to be regarded as model instance in the image. The larger *'min_score'* is chosen, the faster is the search. If the model can be expected never to be occluded in the images, *'min_score'* may be set as high as *0.8* or even *0.9*. If the matches are not tracked to the lowest pyramid level during `find_generic_shape_model` it might happen that instances with a score slightly below *'min_score'* are returned.
  The interplay with *'num_matches'* is explained in the description of the latter one.
  Value range: [*0.0*, *...*, *1.0*].
  Default: *0.5*.
- *'num_matches'*:
  Determines the maximum number of returned instances.
  The value *'all'* (or *0*) mean that every instance whose score is higher than *'min_score'* is returned. For the rest the interplay between *'num_matches'* and *'min_score'* as well as *'max_clutter'* (if set) is as follows:
  - No clutter set:
    In case there are more than *'num_matches'* instances found with a required score, only the *'num_matches'* instances with the highest scores are returned.
    In case there are less than *'num_matches'* instances found with a required score, only the instances fulfilling the score requirement are returned.
  - With clutter set:
    In case there are more than *'num_matches'* instances found with a required score and an allowed clutter score, only the *'num_matches'* instances with the best clutter score are returned.
    In case there are less than *'num_matches'* instances found with a required score and / or allowed clutter score, only the instances fulfilling the score and clutter score requirement are returned.

Tracking matches through the image pyramid depends whether clutter is set:

- No clutter set:
  On each level (except the top level) less promising matches are rejected based on *'num_matches'*. By doing so matches may be rejected although they would have obtained a higher score on the lowest pyramid level. As a result the returned instance for e.g., *'num_matches'* set to *1* can differ from the instance with the highest score returned when more matches are returned by setting *'num_matches'* to *'all'* or a value higher *1*. This explains why it might be preferable to return several instances and then select the instance with the highest score in case multiple objects with a similar score are expected, but only the one with the highest score is of interest.
- With clutter set:
  No matches are rejected during tracking. As a result the runtime using clutter parameters will be at least as high as the runtime without clutter parameters but returning all matches, thus *'num_matches'* set to *'all'*.

Default: *'all'*.

- *'strict_boundaries'*:
  Determines whether found instances are only returned when they are strictly within the given parameter space boundaries (*'true'*) or not (*'false'*). The parameter space includes all determined angle and scaling ranges (see above). Note that such a sort out appears after the reduction to the desired *'num_matches'* matches and as a consequence may lead in extreme cases to no returned match.
  Compatibility: Setting this value on a model not created by `create_generic_shape_model` will change internal model settings. As a consequence, setting for such models *'strict_boundaries'* to *'true'* and back to *'false'* may result in a different behavior regarding matches found outside the requested search range.
  Default: *'false'*.
- *'max_overlap'*:
  Determines by what fraction two instances may overlap at most in order to be considered as different instances and hence to be returned separately. If two instances overlap each other by more than *'max_overlap'* only the one with the higher score is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see `smallest_rectangle2`) of the found instances. *'max_overlap'* set to *0* means the returned instances may not overlap at all, while for *'max_overlap'* = *1* all instances are returned, independent of possible overlapping. For more information about *'max_overlap'*, we refer to the "`Solution Guide II-B - Matching`".
  Restriction: $0 \leq$ *'max_overlap'* ≤ 1.
  Default: *0.5*.
- *'max_overlap_global_enable'*:
  Determines whether the overlap is computed for all found instances regardless of the model (*'true'*) or not (*'false'*). I.e., the overlap of an instance is computed to the found instances of all models and a larger overlap leads to an elimination of this instance. For detailed information on the overlap computation see *'max_overlap'*.
  It is important to note that the global overlap is applied to all models that are used in a search even if only one model has enabled *'max_overlap_global_enable'*.
  The value used for the maximum allowed global overlap is determined as the minimum of the values set for *'max_overlap'* of all passed models.
  List of values: *'true'*, *'false'*.
  Default: *'false'*.
- *'max_clutter'*:
  Determines the maximal clutter score a potential match can have to be regarded as model instance in the image. Of course this applies only to models where clutter is considered, thus *'use_clutter'* is set to *'true'*.
  Default: *0.0*.

3. **Gray Value Treatment**

- *'min_contrast'*:
  Determines the minimal contrast (gray value difference to neighboring pixels) a point in a search image must at least have in order to be compared with the model. The main use of this parameter is to exclude noise, from the matching process. The operator `estimate_noise` can be used to assess the noise.
  Example: The gray values fluctuate within a range of 10 gray levels. In this case it is advisable to set *'min_contrast'* = *10*.

Multichannel images: For a search in a multichannel image using *'metric'* set to *'ignore_color_polarity'*, the noise in one channel must be multiplied by the square root of the number of channels to determine *'min_contrast'*.
Example: The gray values of a 3 channel image fluctuate within a range of 10 gray levels in a single channel. In this case it is advisable to set *'min_contrast'* = *17*.
Note, *'min_contrast'* must be smaller than the contrast threshold set using *'contrast_low'* and *'contrast_high'*, see above.
The value *'auto'* means its value is estimated during the call of `train_generic_shape_model` based on the noise in the model image. Consequently, for a meaningful automatic determination the noise in the model image must be similar to the one in the images used during recognition.
Training: Changing this value to *'auto'* necessitates a training of the shape model.
Suggested values: *'auto'*, *10*, *20*.
Default: *'auto'*.

- *'border_shape_models'*:
Determines whether the shape model `ModelID` to be found with, e.g., `find_generic_shape_model`, may lie partially outside the image (i.e., whether it may cross the image border, independent of the domain). Partially means that the model's origin still must be located within the image. A different origin set with `set_generic_shape_model_param` is not taken into account.
The value *'system'* means the model uses the system-wide set value set by `set_system` for the parameter *'border_shape_models'*.
List of values: *'system'*, *'true'*, *'false'*.
Default: *'system'*.

4. **Refinement**
   - *'subpixel'*:
   Determines which type of subpixel refinement should be carried out. Possible values:
     - *'none'*:
     The pose of the model is only determined with pixel accuracy and the angle resolution that was specified with *'angle_step'*.
     - *'interpolation'*:
     The pose of the model is interpolated from the score function. This mode features very low computation costs.
     - *'least_squares'*:
     The pose of the model is refined using least squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points.
     - *'least_squares_high'*:
     The pose of the model is refined using least squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. More minimization iterations are allowed than for *'least_squares'*, resulting in a possibly increased runtime.
     - *'least_squares_very_high'*:
     The pose of the model is refined using least squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. More minimization iterations are allowed than for *'least_squares_high'*, resulting in a possibly increased runtime.

   The interplay with *'max_deformation'* is explained in the description of latter one.
   Default: *'least_squares'*.
   - *'greediness'*:
   Determines how "greedily" the search should be carried out. A higher value results in a faster search but a less safe search heuristic.
   Restriction: $0 \leq$ *'greediness'* ≤ 1.
   Default: *0.9*.

5. **Image Pyramid**
   - *'pyramid_level_highest'*:
   Determines the highest level of the image pyramid used in the search. This allows to lower the highest image pyramid level set by the model uses during a search. Its value can not be higher than the highest image pyramid level of the model, which is set through *'num_levels'*. As a consequence, *'pyramid_level_highest'* is clipped. The value *'auto'* means no lowering is applied, thus the model uses the highest pyramid level set through *'num_levels'*.

Default: *'auto'*.

- *'pyramid_level_lowest'*:
  Determines the lowest level of the image pyramid to which the found matches are tracked. Not tracking the matches down to the lowest level decreases the runtime of the search. But in general the higher the lowest level, the lower the accuracy of the extracted pose parameters. In extreme cases this can even lead to finding wrong instances of that the desired accuracy cannot be achieved.
  Default: *1*.
- *'pyramid_level_robust_tracking'*:
  Determines whether the so-called "increased tolerance mode" is active (*'true'*) or not (*'false'*). In this mode the lowest image pyramid level on which at least one match can be found is automatically determined during the search. This mode can be helpful in case of input images of poor quality, e.g., defocused, deformed, or noisy images. In such cases the edge information on the lowest image pyramid level may be missing or deformed and as a consequence no instances of the shape model can be found. Nevertheless, the edge information may be sufficient on higher pyramid levels. The selection of the suitable pyramid level, i.e., the lowest pyramid level on which at least one instance of the shape model can be found, depends on the model and on the input image. Of course the restrictions on accuracy and robustness mentioned for *'pyramid_level_lowest'* occur. *Example: 'pyramid_level_highest'* is set to *4*, *'pyramid_level_lowest'* is set to *2*, and *'pyramid_level_robust_tracking'* is set to *'true'*. Now the matching starts at the fourth pyramid level and tracks the matches down to the second lowest pyramid level. If no instance of the model can be found on the pyramid level 2, but matches have been obtained at a higher level, the lowest pyramid level is determined on which at least one instance of the model is found. The instances of this pyramid level will then be returned.
  Default: *'false'*.

6. **Clutter**
   The following parameters are used to specify the clutter region. Note that for this the clutter region has to be set beforehand using `set_generic_shape_model_object`.

   - *'use_clutter'*:
     Determines whether the model `ModelID` considers clutter parameters (*'true'*) or not (*'false'*). The runtime using clutter parameters will be at least as high as the runtime without clutter parameters but returning all matches, thus *'num_matches'* set to *'all'*.
     List of values: *'false'*, *'true'*.
     Default: *'false'*.
   - *'clutter_contrast'*:
     Determines the minimal contrast which edges in the clutter region must have in order to be counted as clutter.
     The polarity of the found clutter edges is ignored, i.e., bright objects on a dark background will yield the same clutter score as dark objects on a bright background, independent of the parameter *'metric'* of the shape model.
     For the value *'auto'* the contrast will be estimated automatically. Note, a suitable estimation requires that the defining region is representative. We recommend to check if the estimated value is suitable and adapt it otherwise.
     Please note that clutter scores are strongly affected by illumination variations.
     Suggested values: *'auto'*, *5*, *10*.
     Restriction: *'clutter_contrast'* must be larger or equal to *'min_contrast'*.
     Default: *'auto'*.
     Attention: A suitable automatic estimation is not possible for shape models generated from XLD contours. Therefore, in such a case the estimated value is set to *'min_contrast'*, see also the section "Automatic modifications" below.
   - *'max_clutter'*:
     The value *'max_clutter'* specifies the maximum clutter score that a match can have in order to be accepted.
     Note, the value of this parameter has no effect on the runtime.
     Value range: [*0.0*, *1.0*].
     Default: *0.0*.
   - *'clutter_hom_mat_2d'*:
     The clutter region is set relative to the model. The transformation matrix *'clutter_hom_mat_2d'* maps the model origin to the position of the model instance (or rather its center of gravity) in the image when the clutter region has been set using `set_generic_shape_model_object`. Possible Values are:

  - *'auto'*:
    Value is estimated automatically. The clutter region is set relative to the model in the template image used by `set_generic_shape_model_object`.
  - Transformation matrix: Transformation is given by the user.

  Default: *'auto'*.

- *'clutter_border_mode'*:
  With *'clutter_border_mode'* the behavior of the clutter score can be influenced in cases where the clutter region of a found match is not entirely contained in the image domain. The corresponding values for *'clutter_border_mode'* are:
  - *'clutter_border_average'*:
    When *'clutter_border_average'* is set, the hidden part of the clutter region is assumed to be filled on average as is its visible part. If the clutter region is not visible at all, the clutter score of the found match is set to *0.0*.
  - *'clutter_border_empty'*:
    When *'clutter_border_empty'* is set, the clutter region is assumed to be clutter-free where it is not visible.

  Note that *'clutter_border_average'* may result in higher clutter scores than *'clutter_border_empty'* even if the match is not located at the border of the image domain.
  List of values: *'clutter_border_average'*, *'clutter_border_empty'*.
  Default: *'clutter_border_average'*.

7. **Regarding the Output**

- Origin:
  The origin (reference point) is specified relative to the center of gravity of the domain (region) of the image that was used to train the shape model with `train_generic_shape_model`. Hence, an origin of (0,0) means that the center of gravity of the domain of the template image is used as the origin. The origin can be set to a new value using the following parameters:
  - *'origin_row'*:
    Determines the column coordinate of a new, shifted center of origin.
    Default: *0*.
  - *'origin_column'*:
    Determines the column coordinate of a new, shifted center of origin.
    Default: *0*.

  Example: Setting *'origin_column'* = *-20* and *'origin_row'* = *-40* results in a new origin lying in the upper left of the center of gravity of the model. Please note that changing the origin influences the used transformation matrices, i.e., the *'clutter_hom_mat_2d'* set by the user and the calculated *'hom_mat_2d'* of a match.
- *'prepare_contours_for_visualization'*:
  Determines whether a found contour of a match should be prepared in `find_generic_shape_model` and returned in `MatchResultID` (*'true'*) or not (*'false'*). Deactivating *'prepare_contours_for_visualization'* decreases the runtime. For how to query the generated contour, see the documentation of `get_generic_shape_model_result_object`.
  List of values: *'true'*, *'false'*.
  Default: *'true'*.
- *'prepare_clutter_region_for_visualization'*:
  Determines whether the clutter region of a match should be prepared in `find_generic_shape_model` and returned in `MatchResultID` (*'true'*) or not (*'false'*). Deactivating *'prepare_clutter_region_for_visualization'* decreases the runtime. For how to query the generated clutter region, see the documentation of `get_generic_shape_model_result_object`.
  List of values: *'true'*, *'false'*.
  Default: *'true'*.
- *'time_measurement'*:
  Determines whether during the search, the time spent in individual parts should be measured. The individual times are returned in `MatchResultID`, see `get_generic_shape_model_result`.
  List of values: *'pipeline'*, *'false'*.
  Default: *'false'*.

Computational Resources During a Search

In this paragraph we list parameters concerning technical influences when searching the model ModelID. The supported parameters GenParamName are:

- *'model_cache'*:
  Determines whether an internal cache based on temporary memory is used (*'true'*) or not *'false'* during e.g., find_generic_shape_model. Switching the *'model_cache'* off (by setting *'false'*) sometimes results in slightly longer execution times but constantly smaller memory footprint.
  The size of the cache depends on the parameter space, thus if the parameter space contains many discretization steps. This means a fine search grid with small step values covering large ranges for start to end values results in a large memory consumption.
  Default: *'true'*.
- *'timeout'*:
  Determines the maximum runtime of the operators used to find the shape model ModelID (e.g., find_generic_shape_model). The temporal accuracy depends on several factors including the size of the model, the speed of the used computer, and the *'timer_mode'* set via set_system. Be aware that the runtime of the search increases by up to 10 percent with activated timeout.
  The value for *'timeout'* must be given in milliseconds. To disable the timeout you can either use a negative value or *'false'*.
  Suggested values: *'false'*, *-1*, *100*.
  Default: *'false'*.

Extended Parameter Estimation

Extended parameter estimation allows for improved estimation of specific parameters. It can be performed by providing representative samples before training the model using train_generic_shape_model in order to get an enhanced estimation for specific parameters. In this paragraph, we list parameters for extended parameter estimation of ModelID. Changing a corresponding value makes it necessary to train the model before the matching. The supported parameters GenParamName are:

- *'training_samples'*:
  Adds training samples to the model for extended parameter estimation. The training samples must be passed as a tuple of dictionaries, where each dictionary defines exactly one sample. The following keys and associated objects are required for a valid sample:
  - image: Training image. The image domain of the training image is disregarded.
  - region: Region(s), that define(s) the respective ground truth of an instance. In order to find the instance, the region must include the model origin as well as the model contours.

  All other keys in the dictionary are ignored. As soon as new training samples are added, training samples that have already been set are overwritten or, in the case of an empty tuple, the training samples are removed from the shape model.
  Restriction: *'training_samples'* cannot be set together with other parameters.
- *'extended_parameter_estimation'*:
  Controls which extended parameter estimation method is performed in the next call to train_generic_shape_model. After successful estimation, the estimated values are set into the model.
  In order to obtain good estimation results it is essential to set all parameters before the model training (thus, before the call of train_generic_shape_model).
  Possible values:
  - *'none'*:
    Deselects all extended parameter estimation methods.
  - *'linear'*:
    Determines a suitable value for *'min_score'*.
  - *'per_level'*:
    Determines suitable values replacing *'greediness'* and *'min_score'*.
    Attention: *'per_level'* depends on the number of used pyramid levels. Thus, it is important not to change the pyramid levels after the extended parameter estimation anymore. As a consequence, possible

changes of the number of pyramid levels whether directly (e.g., setting *'pyramid_level_highest'*) or indirectly (e.g., calling `adapt_shape_model_high_noise`) must be done before the extended parameter estimation (meaning, before the call of `train_generic_shape_model`). Furthermore, the parameter *'max_deformation'* must stay constant.

Restriction: *'extended_parameter_estimation'* cannot be set together with other parameters.

Restriction: All values except *'none'* require representative training samples. If training samples are not provided, extended parameter estimation is not executed and the affected parameters are set to the last user-provided values or the default.

Default: *'per_level'*.

Automatic Modifications

The following parameters are subject to automatic modifications without further notice:

- *'num_levels'*:
  If a number of levels is set such that on the asked levels not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level.
  If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, `train_generic_shape_model` returns with an error message.
- Parameters determining a *step_size*:
  In case the range is not a multiple of the *step_size*, latter one is modified accordingly.
  Examples: 'angle_step', *'iso_scale_step'*.
- `start` and `end` value of a specified range:
  To ensure that for model instances in the exact model position are returned by `find_generic_shape_model`, ranges may modified as follows: If there is no positive integer value n such that the *start* value plus n times the *step_size* gives 0.0 (for angles) and 1.0 (for scalings), respectively, the *start* value is decreased by up to one `step_size` and the *end* value increased such that the total range is increased by exactly one `step_size`.
  Examples: 'angle_start', *'restrict_iso_scale_max'*.
- *'clutter_hom_mat_2d'*:
  Changing the origin (*'origin_row'* and *'origin_column'*) influences the used transformation matrices, i.e., the *'clutter_hom_mat_2d'* set by the user.
- *'clutter_contrast'*:
  If *'min_contrast'* is changed such that it gets bigger than *'clutter_contrast'*, *'clutter_contrast'* is increased if the user set the value *'auto'*.

Note that the transformations are treated internally such that the scalings are applied first, followed by the rotation. Therefore, the model should usually be aligned such that it appears horizontally or vertically in the model image.

Parameters

▷ **`ModelID`** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the shape model.

▷ **`GenParamName`** (input_control) . . . . . attribute.name-array ⇝ *string*
Parameter name.
Default: 'min_score'
List of values: `GenParamName` ∈ {'angle_end', 'angle_start', 'angle_step', 'border_shape_models', 'clutter_border_mode', 'clutter_contrast', 'clutter_hom_mat_2d', 'contrast_high', 'contrast_low', 'extended_parameter_estimation', 'greediness', 'iso_scale_max', 'iso_scale_min', 'iso_scale_step', 'max_clutter', 'max_deformation', 'max_overlap', 'max_overlap_global_enable', 'metric', 'min_contrast', 'min_score', 'min_size', 'model_cache', 'model_identifier', 'num_levels', 'num_matches', 'optimization', 'origin_column', 'origin_row', 'prepare_contours_for_visualization', 'prepare_clutter_region_for_visualization', 'pyramid_level_highest', 'pyramid_level_lowest', 'pyramid_level_robust_tracking', 'restrict_iso_scale_max', 'restrict_iso_scale_min', 'restrict_scale_column_max', 'restrict_scale_column_min', 'restrict_scale_row_max', 'restrict_scale_row_min', 'scale_column_max', 'scale_column_min', 'scale_column_step', 'scale_row_max', 'scale_row_min', 'scale_row_step', 'strict_boundaries', 'subpixel', 'time_measurement', 'timeout', 'training_samples', 'use_clutter'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Parameter value.
Default: 0.5
Suggested values: GenParamValue ∈ {-3.14, -1.57, -0.79, -0.39, 0.0, 0.39, 0.79, 1.57, 3.14}

Result

If the parameters are valid, the operator set_generic_shape_model_param returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_generic_shape_model

Possible Successors

get_generic_shape_model_param, train_generic_shape_model, find_generic_shape_model

Module

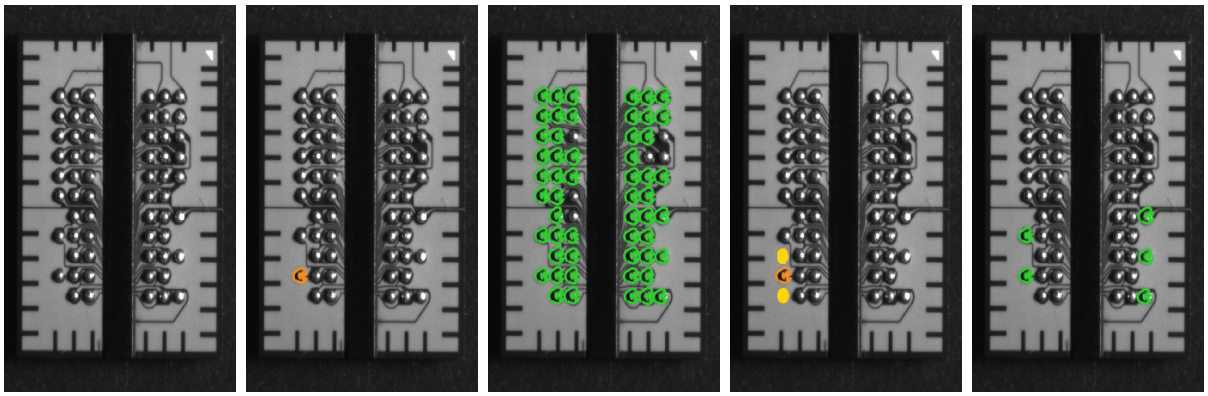
Matching

set_shape_model_clutter (ClutterRegion : : ModelID, HomMat2D, ClutterContrast, GenParamName, GenParamValue :)

Set the clutter parameters of a shape model.

set_shape_model_clutter sets the clutter parameters of the shape model ModelID. In particular, a region is defined relative to the model contours, in which no (or too faint) clutter edges should occur in the search image. This is helpful, e.g., if it is a special characteristic of the model that, in specific parts near to the model, structures are missing.

For example, within the HDevelop example program find_shape_model_clutter.hdev, the model corresponds to a ball in a ball grid array (BGA). As the ball grid array represents a repeating pattern, many matches are returned when matching is applied without considering the neighborhood of the model. By defining a clutter region, the search can be restricted to specific instances of the model.



(1) (2) (3) (4) (5)

(1) Image of ball grid array, (2) shape model of a single ball of the array (orange), (3) matching result without using clutter parameters (green), (4) defining the clutter region (yellow) relative to the model (orange), (5) matching result using clutter parameters (green).

For matches found in a search image, edges within the clutter region increase the resulting clutter scores. These are returned by find_shape_model, find_scaled_shape_model, find_aniso_shape_model, find_shape_models, find_scaled_shape_models, and find_aniso_shape_models in the parameter Score, following the common values measuring how much of the model is visible in the image. Note that the input parameter MinScore of those operators additionally expects information specifying the maximum clutter score up to which matches should be returned. For more information regarding the setting of MinScore, please refer to the respective operator reference.

To define the clutter region relative to the model contours, you need the region ClutterRegion and the transformation matrix HomMat2D that maps the model contours to the respective position where the object appears in an image. Typically, you obtain them by searching for a model instance in an image, e.g., with the operator find_shape_model. Then, the transformation matrix can be determined with the procedure get_hom_mat2d_from_matching_result using the respective output parameters of the search. The region ClutterRegion is specified within the same image. Note that ClutterRegion should contain the regions around the superfluous edges that are typical for all expected manifestations of the matches. We recommend that the clutter region is chosen larger than necessary when large scale ranges are searched. Furthermore, choosing the clutter region not too near to the expected model contours can increase the robustness.

The parameter ClutterContrast determines the contrast the edges in the clutter region must have in order to be counted as clutter. In many applications, the parameter Contrast, which has been used to create the shape model, is also a reasonable choice for ClutterContrast. ClutterContrast may not be smaller than the parameter MinContrast of the shape model, otherwise an error is raised at runtime. The polarity of the found clutter edges is ignored, i.e., bright objects on a dark background will yield the same clutter score as dark objects on a bright background, independent of the parameter Metric of the shape model. Please note that of all shape-based matching results, clutter scores are affected the most when a variation of illumination occurs.

With GenParamName set to *'clutter_border_mode'*, the behavior of the clutter score can be influenced in cases where the clutter region of a found match is not entirely contained in the image domain. The corresponding values for *'clutter_border_mode'* (GenParamValue) can be *'clutter_border_average'* or *'clutter_border_empty'*.

- *'clutter_border_average'*:
  When GenParamValue is set to *'clutter_border_average'* (default), the hidden part of the clutter region is assumed to be filled on average as is its visible part. If the clutter region is not visible at all, the clutter score of the found match is set to *0.0*.
- *'clutter_border_empty'*:
  When GenParamValue is set to *'clutter_border_empty'*, the clutter region is assumed to be empty where it is not visible.

Note that GenParamValue set to *'clutter_border_average'* results in higher clutter scores even if the match is not located at the border of the image domain.

The use of clutter parameters can be disabled and enabled by calling set_shape_model_param. For newly created shape models, the use of clutter parameters is disabled. After calling set_shape_model_clutter, the use of clutter parameters is enabled. Depending on the activation status for the use of clutter parameters, the find operators, e.g., find_shape_model, expect a different number of entries in the input parameter MinScore. The set clutter parameters and the value of *'use_clutter'* can be queried using the operator get_shape_model_clutter.

Parameters

▷ **ClutterRegion** (input_object) ..... region ⇝ *object*
Region where no clutter should occur.

▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
Handle of the model.

▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
Transformation matrix.

▷ **ClutterContrast** (input_control) ..... number ⇝ *integer*
Minimum contrast of clutter in the search images.
Default: 128

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Parameter names.
List of values: GenParamName ∈ {'clutter_border_mode'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / integer / string
Parameter values.
List of values: GenParamValue ∈ {'clutter_border_empty', 'clutter_border_average'}

Example

```
*
* Create a shape model.
read_image (ImageModel, '/bga_gap/bga_gap_01.png')
gen_circle (ROI, 753.869, 551.624, 28.4027)
reduce_domain (ImageModel, ROI, ImageReduced)
create_aniso_shape_model (ImageReduced, 'auto', rad(0), rad(0), 'auto', \
                          0.95, 1.05, 'auto', 0.95, 1.05, 'auto', 'auto', \
                          'use_polarity', 'auto', 'auto', ModelID)
*
* Specify the clutter parameters.
find_aniso_shape_model (ImageModel, ModelID, rad(0), rad(0), 0.95, 1.05, \
                        0.95, 1.05, 0.83, 0, 0.0, 'least_squares', 0, 0.0, \
                        Row, Column, Angle, ScaleR, ScaleC, Score)
get_hom_mat2d_from_matching_result (Row[0], Column[0], Angle[0], ScaleR[0], \
                                    ScaleC[0], HomMat2D)
*
gen_circle (ROI_0, 700.655, 548.666, 21.6273)
gen_circle (ROI_1_0, 810.655, 550.611, 21.6273)
union2 (ROI_0, ROI_1_0, ClutterRegion)
*
ClutterContrast := 12
*
* Set the clutter parameters into shape model.
set_shape_model_clutter (ClutterRegion, ModelID, HomMat2D, ClutterContrast, \
                         [], [])
*
* Use the shape model to detect objects with a small amount of clutter
read_image (Image,'/bga_gap/bga_gap_02.png')
MaxClutter := 0.09
find_aniso_shape_model (Image, ModelID, rad(0), rad(0), 0.95, 1.05, 0.95, \
                        1.05, [0.83, MaxClutter], 0, 0.0, 'least_squares', \
                        [4, 3],  0.0,  Row, Column, Angle, ScaleR, ScaleC, \
                        Score)
*
* Visualize the matches
dev_display_shape_matching_results (ModelID, ['green', 'red'], Row, Column, \
                                    Angle, ScaleR, ScaleC, 0)
Clutter := Score[|Score|/2:|Score|-1]
Score := Score[0:|Score|/2-1]
dev_inspect_ctrl ([Score, Clutter])
```

Result

If the parameters are valid, the operator set_shape_model_clutter returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_shape_model, create_scaled_shape_model, create_aniso_shape_model, create_shape_model_xld, create_scaled_shape_model_xld, create_aniso_shape_model_xld

Possible Successors

find_shape_model, find_scaled_shape_model, find_aniso_shape_model, find_shape_models, find_scaled_shape_models, find_aniso_shape_models, get_shape_model_clutter

Alternatives

set_generic_shape_model_param, set_generic_shape_model_object

See also

set_shape_model_param

Module

Matching

set_shape_model_metric (Image : : ModelID, HomMat2D, Metric :)

Set the metric of a shape model that was created from XLD contours.

set_shape_model_metric changes the value of the parameter Metric that was selected during the creation of the shape model ModelID from XLD contours. For this, the polarity of the model edges is determined based on the image Image. The transformation that maps the model edges to the image position where the respective object appears is given in HomMat2D.

The parameter Metric determines the conditions under which the model is recognized later in the search image. If Metric = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background. If Metric = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of find_generic_shape_model will increase slightly in this case.

It must be ensured that the object edges in the Image have the same (or inverse) polarity as the object edges in the image in which the object must be searched later. Especially, the object must not be occluded in the Image and the background must be either brighter than the object or darker. Otherwise, the determined polarity of the model edges will not represent the polarity of the object edges during the search. Note that only the polarity of the edges is determined, not their contrast. Note also that the polarity is determined from a single-channel image, only. If a multichannel image is passed in Image, only the first channel will be used (and no error message will be returned).

The transformation defined by the matrix HomMat2D maps the model contours to the respective position where the object appears in the Image. It can be obtained by finding the object in the Image, e.g., with the operator find_generic_shape_model and determining the respective transformation matrix with the procedure get_hom_mat2d_from_matching_result. The operator set_shape_model_metric ignores an origin that was set by set_shape_model_origin and always assumes (0,0) as the origin. Hence, when finding the object in the Image, all other operators must have the origin at the point (0,0). Therefore, set_shape_model_origin must be called with (0,0) as the origin.

A typical proceeding is to read the XLD contours from file. Since these XLD contours do not provide polarity information, the shape model must be created from the XLD contours by setting the parameter Metric to *'ignore_local_polarity'*. Then, in a first search image the model is recognized. The transformation that maps the model contours to the position of the object in the search image can be determined from the matching result. To verify the match interactively, the model contours can be mapped to this position. If the matching result is correct, the value of the parameter Metric can be changed, e.g., to *'use_polarity'*. This leads to a faster and more robust recognition in the following images.

Attention

set_shape_model_metric can only be used with shape models that were created from XLD contours.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
Input image used for the determination of the polarity.

▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
Handle of the model.

▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
Transformation matrix.

▷ **Metric** (input_control) ..... string ⇝ *string*
Match metric.
Default: 'use_polarity'
List of values: `Metric` ∈ {'use_polarity', 'ignore_global_polarity'}

Example

```
*
* Read the contours from file
read_contour_xld_dxf (Contours, 'metal-part-01', [], [], DxfStatus)
*
* Scale the contours such that they correspond to the image
hom_mat2d_identity (HomMat2DIdentity)
hom_mat2d_scale (HomMat2DIdentity, 3.35, 3.35, 0, 0, HomMat2DScale)
affine_trans_contour_xld (Contours, ContoursAffineTrans, HomMat2DScale)
*
* Create the shape model with Metric='ignore_local_polarity'
create_generic_shape_model (ModelID)
train_generic_shape_model (ContoursAffineTrans, ModelID)
*
* Get an image of the object and try to find the object in this image
read_image (Image, 'metal-parts/metal-parts-01')
find_generic_shape_model (Image, ModelID, MatchResultID, NumMatchResult)
*
* Visualize the match
get_generic_shape_model_result_object (Objects, MatchResultID, 'all', \
                                       'contours')
stop()
get_generic_shape_model_result (MatchResultID, 'all', 'hom_mat_2d', \
                                HomMat2D)
*
* Set the matching metric to 'use_polarity', based on the position
* of the found object
set_shape_model_metric (Image, ModelID, HomMat2D, 'use_polarity')
*
* Use the shape model to detect the object faster and more robust
read_image (Image, 'metal-parts/metal-parts-02')
find_generic_shape_model (Image, ModelID, MatchResultID, NumMatchResult)
*
* Visualize the matches
get_generic_shape_model_result_object (Objects, MatchResultID, 'all', \
                                       'contours')
```

Result

If the parameters are valid, the operator `set_shape_model_metric` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_generic_shape_model

Possible Successors

find_generic_shape_model

See also

set_generic_shape_model_param

Module

Matching

set_shape_model_origin (: : ModelID, Row, Column :)

Set the origin (reference point) of a shape model.

The operator set_shape_model_origin sets the origin (reference point) of the shape model ModelID to a new value. The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the shape model with create_shape_model, create_scaled_shape_model, or create_aniso_shape_model. Hence, an origin of (0,0) means that the center of gravity of the domain of the model image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity.

Parameters

▷ **ModelID** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the model.

▷ **Row** (input_control) . . . . . point.y ⇝ *real*
Row coordinate of the origin of the shape model.

▷ **Column** (input_control) . . . . . point.x ⇝ *real*
Column coordinate of the origin of the shape model.

Result

If the handle of the model is valid, the operator set_shape_model_origin returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_generic_shape_model

Possible Successors

train_generic_shape_model

See also

area_center

Module

Matching

set_shape_model_param (: : ModelID, GenParamName, GenParamValue :)

Set selected parameters of the shape model.

The operator set_shape_model_param sets the selected parameters GenParamName in the shape model ModelID. The following parameters can be modified:

'min_contrast'

Sets the minimum contrast of the object in the search images for the shape model ModelID. Thereby, the value of *'min_contrast'* that was originally set, e.g., with create_shape_model, is overwritten for the shape model ModelID. Note that if the shape model ModelID was read from file and if this file should be changed as well, the shape model ModelID must again be written to file after the execution of the operator set_shape_model_param.

'timeout'

Sets the maximum runtime of the operators used to find the shape model ModelID (e.g., find_shape_model). This is especially useful in cases where a maximum cycle time has to be ensured. The *'timeout'* must be given in milliseconds. The temporal accuracy depends on several factors including the size of the model, the speed of your computer, and the *'timer_mode'* set via set_system. Be aware that the runtime of the search increases by up to 10 percent with activated timeout. To disable the timeout you can either use a negative value or *'false'*.

'border_shape_models'

This parameter determines whether the shape model ModelID to be found with, e.g., find_shape_model, may lie partially outside the image (i.e., whether they may cross the image border). Partially means that the model's origin still must be located within the image. A different origin set with set_shape_model_origin is not taken into account.

The value of *'border_shape_models'* can be *'true'*, *'false'*, or *'system'*. The value *'system'* is the default behavior and uses the system-wide used value that was set by set_system for the parameter *'border_shape_models'*.

'use_clutter'

Disables or enables the use of clutter parameters for the shape model ModelID, which have been set previously using set_shape_model_clutter. The value of *'use_clutter'* can be *'true'* or *'false'*. Note that the value of *'use_clutter'* affects the expected number of entries in the input parameter MinScore of the find operators. Please refer to the documentation of the different find operators, e.g., find_shape_model.

For newly created shape models, the use of clutter parameters is disabled. After calling set_shape_model_clutter, the use of clutter parameters is enabled. Clutter parameters and the value of *'use_clutter'* can be queried using get_shape_model_clutter.

'model_cache'

Disables or enables the use of an internal cache based on temporary memory that is used by a shape model when it executes, e.g., find_shape_model. The size of the cache depends on whether the find operation needs a big parameter space with many discretization steps. This means that a small AngleStep and ScaleStep and a big AngleExtent and a big range given by ScaleMin and ScaleMax result in a big memory consumption. Switching the *'model_cache'* off (by setting *'false'*) sometimes results in slightly longer execution times but constantly small memory footprint, particularly important in embedded applications. By default, this cache is switched on (*'true'*).

Parameters

▷ **ModelID** (input_control) ..... shape_model ⇝ *handle*
Handle of the model.

▷ **GenParamName** (input_control) ..... attribute.name-array ⇝ *string*
Parameter names.
List of values: GenParamName ∈ {'min_contrast', 'timeout', 'border_shape_models', 'use_clutter', 'model_cache'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / integer / string
Parameter values.
List of values: GenParamValue ∈ {'true', 'false', 'system'}

Result

If the parameters are valid, the operator set_shape_model_param returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_shape_model_xld, create_scaled_shape_model_xld, create_aniso_shape_model_xld

Possible Successors

find_shape_model, find_scaled_shape_model, find_aniso_shape_model

Alternatives

set_generic_shape_model_param

See also

create_shape_model_xld, create_scaled_shape_model_xld, create_aniso_shape_model_xld, set_shape_model_clutter, get_shape_model_clutter

Module

Matching

train_generic_shape_model (Template : : ModelID :)

Train a shape model for matching.

The operator train_generic_shape_model trains the shape model ModelID in order to find the training pattern given in Template.

Template accepts the training pattern to be of one of the following types:

Image: The domain of the image given in Template sets the ROI of the model. Which channels of a multichannel image are used depends on the model parameter *'metric'* (see set_generic_shape_model_param).

The model origin (reference point) is defined by the center of gravity of the image domain. The origin can be reset to a different point using set_generic_shape_model_param.

XLD contour: XLD contours passed in Template should represent the gray value edges of the pattern to be searched.

The model origin (reference point) is defined by the center of gravity of the smallest axis-parallel rectangle enclosing the contours. The origin can be reset to a different point using set_generic_shape_model_param.

Properties of the model ModelID can be retrieved using get_generic_shape_model_param and modified using set_generic_shape_model_param. Note that after modifying certain parameters the model needs to be (re)trained, see set_generic_shape_model_param.

Added training samples: In case training samples have been added they are freed from the model after a successful call of train_generic_shape_model.

Parameters

▷ **Template** (input_object) . . . . . (multichannel-)object(-array) ⇝ *object*
Training image or xld.

▷ **ModelID** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the shape model.

Result

If the parameters are valid, the operator `train_generic_shape_model` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ModelID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_generic_shape_model`, `set_generic_shape_model_param`

Possible Successors

`find_generic_shape_model`

Module

Matching

write_shape_model (: : ModelID, FileName :)

Write a shape model to a file.

The operator `write_shape_model` writes a shape model to the file `FileName`. The model can be read again with `read_shape_model`. The default HALCON file extension for the shape model is 'shm'.

Parameters

▷ **ModelID** (input_control) . . . . . shape_model ⇝ *handle*
Handle of the model.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: `.shm`

Result

If the file name is valid (write permission), the operator `write_shape_model` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`set_generic_shape_model_param`, `set_generic_shape_model_object`

Module

Matching

Chapter 19

Matrix

19.1 Access

get_diagonal_matrix (: : MatrixID, Diagonal : VectorID)

Get the diagonal elements of a matrix.

The operator get_diagonal_matrix generates a new matrix Vector and copies the diagonal elements of the Matrix to this new matrix. The Matrix is defined by the matrix handle MatrixID. The matrix Vector has one column and n rows, where n = number of diagonal elements. The operator returns the matrix handle VectorID of the matrix Vector. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

If Diagonal = 0, the output of the Vector is the main diagonal of the Matrix.

Example:

Diagonal = *0*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \texttt{Vector} = \begin{bmatrix} 3.0 \\ 7.0 \\ 1.0 \end{bmatrix}$$

If Diagonal is positive, the output Vector is the Diagonal-th super-diagonal of the Matrix.

Example:

Diagonal = *2*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 & 1.0 \\ -5.0 & 7.0 & 2.0 & 6.0 \\ -9.0 & -4.0 & 1.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Vector} = \begin{bmatrix} -2.0 \\ 6.0 \end{bmatrix}$$

If Diagonal is negative, the Diagonal-th sub-diagonal of the Matrix is copied.

Example:

Diagonal = *-1*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 & 1.0 \\ -5.0 & 7.0 & 2.0 & 6.0 \\ -9.0 & -4.0 & 1.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Vector} = \begin{bmatrix} -5.0 \\ -4.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Diagonal** (input_control) . . . . . integer ⇝ *integer*
Number of the desired diagonal.
Default: 0
Suggested values: Diagonal ∈ {-20, -10, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20}

▷ **VectorID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle containing the diagonal elements.

Result

If the parameters are valid, the operator get_diagonal_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

See also

set_diagonal_matrix

Module

Foundation

get_full_matrix (: : MatrixID : Values)

Return all values of a matrix.

The operator get_full_matrix returns the values of all elements of the Matrix given by the matrix handle MatrixID. The output parameter Values is a tuple of floating point numbers and contains all values in a row-major order, i.e., line by line.

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \rightarrow \quad \texttt{Values} = [3.0, 1.0, -2.0, -5.0, 7.0, 2.0, -9.0, -4.0, 1.0]$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Values** (output_control) . . . . . real(-array) ⇝ *real*
Values of the matrix elements.

Result

If the parameters are valid, the operator `get_full_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`clear_matrix`

See also

`set_full_matrix`

Module

Foundation

get_sub_matrix (: : MatrixID, Row, Column, RowsSub, ColumnsSub : MatrixSubID)

Get a sub-matrix of a matrix.

The operator `get_sub_matrix` generates a new matrix `MatrixSub` and copies to this matrix a part of the input `Matrix` defined by the matrix handle `MatrixID`. The part of the `Matrix` is determined by the upper left corner (`Row`,`Column`) and the sub-matrix dimensions (`RowsSub`, `ColumnsSub`). The operator returns the matrix handle `MatrixSubID` of the matrix `MatrixSub`. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`.

Example:

`Row` = *0*, `Column` = *1*, `RowsSub` = *3*, `ColumnsSub` = *2*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 & 1.0 \\ -5.0 & 7.0 & 2.0 & 6.0 \\ -9.0 & -4.0 & 1.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixSub} = \begin{bmatrix} 1.0 & -2.0 \\ 7.0 & 2.0 \\ -4.0 & 1.0 \end{bmatrix}$$

Attention

The conditions $0 \leq$ `Row` $<$ size of `Matrix` in row direction, `Row` + `RowsSub` $\leq$ size of `Matrix` in the row direction, $0 \leq$ `Column` $<$ size of `Matrix` in the column direction, and `Column` + `ColumnsSub` $\leq$ size of matrix `Matrix` in the column direction must be satisfied.

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Row** (input_control) . . . . . integer ⇝ *integer*
Upper row position of the sub-matrix in the input matrix.
Default: 0
Suggested values: Row ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Row >= 0

▷ **Column** (input_control) . . . . . integer ⇝ *integer*
Left column position of the sub-matrix in the input matrix.
Default: 0
Suggested values: Column ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Column >= 0

▷ **RowsSub** (input_control) . . . . . integer ⇝ *integer*
Number of rows of the sub-matrix.
Default: 1
Suggested values: RowsSub ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: RowsSub >= 1

▷ **ColumnsSub** (input_control) . . . . . integer ⇝ *integer*
Number of columns of the sub-matrix.
Default: 1
Suggested values: ColumnsSub ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: ColumnsSub >= 1

▷ **MatrixSubID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle of the sub-matrix.

Result

If the parameters are valid, the operator get_sub_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

See also

set_sub_matrix

Module

Foundation

get_value_matrix (: : MatrixID, Row, Column : Value)

Return one ore more elements of a matrix.

The operator get_value_matrix returns the values of one ore more elements of the Matrix as a tuple of floating point numbers. The Matrix is given by the matrix handle MatrixID. The row coordinates of the elements of the Matrix are determined by the tuple Row, the column coordinates by the tuple Column.

Example:

Row = *[0,2,1]*, Column = *[1,0,3]*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 & 1.0 \\ -5.0 & 7.0 & 2.0 & 6.0 \\ -9.0 & -4.0 & 1.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Value} = [1.0, -9.0, 6.0]$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Row** (input_control) . . . . . integer(-array) ⇝ *integer*
Row numbers of matrix elements to be returned.
Default: 0
Suggested values: Row ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Row >= 0

▷ **Column** (input_control) . . . . . integer(-array) ⇝ *integer*
Column numbers of matrix elements to be returned.
Default: 0
Suggested values: Column ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Column >= 0

▷ **Value** (output_control) . . . . . real(-array) ⇝ *real*
Values of indicated matrix elements.

Result

If the parameters are valid, the operator get_value_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

clear_matrix

See also

set_value_matrix

Module

Foundation

set_diagonal_matrix (: : MatrixID, VectorID, Diagonal :)

Set the diagonal elements of a matrix.

The operator set_diagonal_matrix *overwrites* the diagonal elements of the Matrix with the elements of the matrix Vector. The matrices are defined by their matrix handles MatrixID and VectorID. The matrix Vector must have one column and one row, n columns and one row or one column and n rows. n is the number of elements to be set in the Matrix (see below). If the matrix Vector has one column and one row, i.e., the matrix has one value, each element of the diagonal of the Matrix is overwritten by this value. Otherwise, the diagonal is overwritten by the elements of the matrix Vector.

If Diagonal = 0, the main diagonal of the Matrix is overwritten. The number n = min(number of rows of Matrix, number of columns of Matrix).

Example 1:

$$\texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$\texttt{Vector} = \begin{bmatrix} 3.0 & 7.0 & 1.0 \end{bmatrix} \qquad \texttt{Diagonal} = 0$$

$$\rightarrow \qquad \texttt{Matrix} = \begin{bmatrix} 3.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 7.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}$$

Example 2:

$$\texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$\texttt{Vector} = \begin{bmatrix} 3.0 \end{bmatrix} \qquad \texttt{Diagonal} = 0$$

$$\rightarrow \qquad \texttt{Matrix} = \begin{bmatrix} 3.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & 0.0 \end{bmatrix}$$

If `Diagonal` is positive, the `Diagonal`-th super-diagonal of `Matrix` is overwritten. For the example 1 the number n = min(number of rows of `Matrix`, parameter `Diagonal`). For the example 2 the number n = min(number of rows of `Matrix`, number of columns of `Matrix`).

Example 1:

$$\texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$\texttt{Vector} = \begin{bmatrix} -2.0 & 6.0 \end{bmatrix} \qquad \texttt{Diagonal} = 2$$

$$\rightarrow \qquad \texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & -2.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 6.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Example 2:

$$\texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$\texttt{Vector} = \begin{bmatrix} -2.0 \end{bmatrix} \qquad \texttt{Diagonal} = 2$$

$$\rightarrow \qquad \texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & -2.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -2.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

If Diagonal is negative, the Diagonal-th sub-diagonal of Matrix is overwritten. For the example 1 the number n = min(number of columns of Matrix, parameter Diagonal). For the example 2 the number n = min(number of rows of Matrix, number of columns of Matrix).

Example 1:

$$\texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$\texttt{Vector} = \begin{bmatrix} -5.0 & -4.0 \end{bmatrix} \qquad \texttt{Diagonal} = -1$$

$$\rightarrow \qquad \texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ -5.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -4.0 & 0.0 & 0.0 \end{bmatrix}$$

Example 2:

$$\texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$\texttt{Vector} = \begin{bmatrix} -5.0 \end{bmatrix} \qquad \texttt{Diagonal} = -1$$

$$\rightarrow \qquad \texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ -5.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -5.0 & 0.0 & 0.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **VectorID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle containing the diagonal elements to be set.

▷ **Diagonal** (input_control) . . . . . integer ⇝ *integer*
Position of the diagonal.
Default: 0
Suggested values: Diagonal ∈ {-20, -10, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20}

Result

If the parameters are valid, the operator set_diagonal_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

See also

get_diagonal_matrix

Module

Foundation

set_full_matrix (: : MatrixID, Values :)

Set all values of a matrix.

The operator set_full_matrix *sets* all elements of the input Matrix defined by the matrix handle MatrixID. The values of the parameter Values can be a tuple of floating point or integer numbers. Integer numbers are converted to floating point numbers automatically. The parameter Values must contain all values in a row-major order, i.e., stored line by line. In addition, the number of elements in Values must be 1 or identical to the number of all elements of the matrix.

Note: The same result can be reached with the operator create_matrix. The advantage by using the operator set_full_matrix is to recycle a matrix that is no longer needed. Thus, the runtime of the operation takes fewer time.

Example 1:

$$\texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \qquad \texttt{Values} = [3, 1, -2, -5, 7, 2, -9, -4, 1]$$

$$\rightarrow \qquad \texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix}$$

Example 2:

$$\texttt{Matrix} = \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix} \qquad \texttt{Values} = [7]$$

$$\rightarrow \qquad \texttt{Matrix} = \begin{bmatrix} 7.0 & 7.0 & 7.0 \\ 7.0 & 7.0 & 7.0 \\ 7.0 & 7.0 & 7.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Values** (input_control) . . . . . number(-array) ⇝ *real* / integer
Values to be set.

Result

If the parameters are valid, the operator set_full_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

clear_matrix

See also

get_full_matrix

Module

Foundation

set_sub_matrix (: : MatrixID, MatrixSubID, Row, Column :)

Set a sub-matrix of a matrix.

The operator set_sub_matrix *overwrites* a part of the Matrix with the matrix MatrixSub. The input matrices are defined by the matrix handles MatrixID and MatrixSubID. The parameters Row and Column determine the position of the upper left corner of the sub-matrix MatrixSub in Matrix.

Example:

Row = *0*, Column = *1*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 0.0 & 9.0 & 1.0 \\ -5.0 & 3.0 & 1.0 & 6.0 \\ -9.0 & 0.0 & 1.0 & -1.0 \end{bmatrix} \qquad \texttt{MatrixSub} = \begin{bmatrix} 1.0 & -2.0 \\ 7.0 & 2.0 \\ -4.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 & 1.0 \\ -5.0 & 7.0 & 2.0 & 6.0 \\ -9.0 & -4.0 & 1.0 & -1.0 \end{bmatrix}$$

Attention

The conditions $0 \leq$ Row $<$ size of matrix Matrix in the row direction, Row $+$ size of matrix MatrixSub in the row direction $\leq$ size of matrix Matrix in the row direction, $0 \leq$ Column $<$ size of Matrix in the column direction, and Column $+$ size of matrix MatrixSub in the columns direction $\leq$ size of Matrix in the column direction must be satisfied.

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixSubID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input sub-matrix.

▷ **Row** (input_control) ..... integer ⇝ *integer*
Upper row position of the sub-matrix in the matrix.
Default: 0
Suggested values: Row ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Row >= 0

▷ **Column** (input_control) ..... integer ⇝ *integer*
Left column position of the sub-matrix in the matrix.
Default: 0
Suggested values: Column ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Column >= 0

Result

If the parameters are valid, the operator set_sub_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

See also

get_sub_matrix

Module

Foundation

set_value_matrix (: : MatrixID, Row, Column, Value :)

Set one or more elements of a matrix.

The operator set_value_matrix *sets* the values of the elements of the input Matrix at the positions (Row,Column) to the values specified by Value. The values can be a tuple of floating point or integer numbers. Integer numbers are converted to floating point numbers automatically. The number of values of Value must match the number of elements of Row and Column. In addition, the conditions $0 \leq$ Row $<$ size of Matrix in the row direction and $0 \leq$ Column $<$ size of Matrix in the column direction must be satisfied. The Matrix is defined by the matrix handle MatrixID.

Example:

Row = *[0,2,1]*, Column = *[1,0,3]*, Value = *[1,-9,6]*

$$\text{Matrix} = \begin{bmatrix} 3.0 & 0.0 & -2.0 & 1.0 \\ -5.0 & 7.0 & 2.0 & -2.0 \\ 8.0 & -4.0 & 1.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \quad \text{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 & 1.0 \\ -5.0 & 7.0 & 2.0 & 6.0 \\ -9.0 & -4.0 & 1.0 & -1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Row** (input_control) . . . . . integer(-array) ⇝ *integer*
Row numbers of the matrix elements to be modified.
Default: 0
Suggested values: Row ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Row >= 0

▷ **Column** (input_control) . . . . . integer(-array) ⇝ *integer*
Column numbers of the matrix elements to be modified.
Default: 0
Suggested values: Column ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Column >= 0

▷ **Value** (input_control) . . . . . number(-array) ⇝ *real* / integer
Values to be set in the indicated matrix elements.
Default: 0
Suggested values: Value ∈ {0, 1, -1}

Result

If the parameters are valid, the operator set_value_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

clear_matrix

See also

get_value_matrix

Module

Foundation

19.2 Arithmetic

abs_matrix (: : MatrixID : MatrixAbsID)

Compute the absolute values of the elements of a matrix.

The operator abs_matrix computes the absolute values of all elements of the input Matrix given by the matrix handle MatrixID. A new matrix MatrixAbs is generated with the result. The operator returns the matrix handle MatrixAbsID of the matrix MatrixAbs. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\mathtt{MatrixAbs}_{ij} = |\mathtt{Matrix}_{ij}|.$$

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \rightarrow \quad \texttt{MatrixAbs} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixAbsID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the absolute values of the input matrix.

Result

If the parameters are valid, the operator `abs_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

Alternatives

`abs_matrix_mod`

Module

Foundation

abs_matrix_mod (: : MatrixID :)

Compute the absolute values of the elements of a matrix.

The operator `abs_matrix_mod` computes the absolute values of all elements of the input `Matrix` given by the matrix handle `MatrixID`. The input matrix is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`. The formula for the calculation of the result is:

$$\texttt{Matrix}_{ij} \quad = \quad |\texttt{Matrix}_{ij}|.$$

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

Result

If the parameters are valid, the operator `abs_matrix_mod` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

abs_matrix

Module

Foundation

add_matrix (: : MatrixAID, MatrixBID : MatrixSumID)

Add two matrices.

The operator add_matrix computes the sum of the input matrices MatrixA and MatrixB given by the matrix handles MatrixAID and MatrixBID. Both matrices must have identical dimensions. A new matrix MatrixSum is generated with the result. The operator returns the matrix handle MatrixSumID of the matrix MatrixSum. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\texttt{MatrixSum} = \texttt{MatrixA} + \texttt{MatrixB}.$$

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \texttt{MatrixB} = \begin{bmatrix} 2.0 & 8.0 & -3.0 \\ -4.0 & -1.0 & 5.0 \\ 2.0 & -4.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixSum} = \begin{bmatrix} 5.0 & 9.0 & -5.0 \\ -9.0 & 6.0 & 7.0 \\ -7.0 & -8.0 & 8.0 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix B.

▷ **MatrixSumID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the sum of the input matrices.

Result

If the parameters are valid, the operator add_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

add_matrix_mod

See also

sub_matrix, sub_matrix_mod

Module

Foundation

add_matrix_mod (: : MatrixAID, MatrixBID :)

Add two matrices.

The operator add_matrix_mod computes the sum of the input matrices MatrixA and MatrixB given by the matrix handles MatrixAID and MatrixBID. Both matrices must have identical dimensions. The input matrix MatrixA is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\texttt{MatrixA} = \texttt{MatrixA} + \texttt{MatrixB}.$$

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \texttt{MatrixB} = \begin{bmatrix} 2.0 & 8.0 & -3.0 \\ -4.0 & -1.0 & 5.0 \\ 2.0 & -4.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixA} = \begin{bmatrix} 5.0 & 9.0 & -5.0 \\ -9.0 & 6.0 & 7.0 \\ -7.0 & -8.0 & 8.0 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix B.

Result

If the parameters are valid, the operator add_matrix_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixAID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

add_matrix

See also

sub_matrix, sub_matrix_mod

Module

Foundation

div_element_matrix (: : MatrixAID, MatrixBID : MatrixDivID)

Divide matrices element-by-element.

The operator div_element_matrix divides the two matrices MatrixA and MatrixB element-by-element. The matrices are given by the matrix handles MatrixAID and MatrixBID. It is required that both input matrices have identical dimensions. The new created output MatrixDiv contains the result. The operator returns the matrix handle MatrixDivID of the matrix MatrixDiv. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\texttt{MatrixDiv}_{ij} \;=\; \texttt{MatrixA}_{ij} \;/\; \texttt{MatrixB}_{ij}.$$

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 4.0 & 8.0 & 0.0 \\ -6.0 & 7.0 & 2.0 \\ -9.0 & 4.0 & 1.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 2.0 & 8.0 & -3.0 \\ -4.0 & -1.0 & 4.0 \\ -2.0 & -4.0 & 2.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixDiv} = \begin{bmatrix} 2.0 & 1.0 & 0.0 \\ 1.5 & -7.0 & 0.5 \\ 4.5 & -1.0 & 0.5 \end{bmatrix}$$

Parameters

- ▷ **MatrixAID** (input_control) ..... matrix ⇝ *handle*
  Matrix handle of the input matrix A.
- ▷ **MatrixBID** (input_control) ..... matrix ⇝ *handle*
  Matrix handle of the input matrix B.
- ▷ **MatrixDivID** (output_control) ..... matrix ⇝ *handle*
  Matrix handle with the divided values of input matrices.

Result

If the parameters are valid, the operator div_element_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

div_element_matrix_mod

See also

mult_element_matrix, mult_element_matrix_mod, scale_matrix, scale_matrix_mod

Module

Foundation

div_element_matrix_mod (: : MatrixAID, MatrixBID :)

Divide matrices element-by-element.

The operator div_element_matrix_mod divides the two matrices MatrixA and MatrixB element-by-element. The matrices are given by the matrix handles MatrixAID and MatrixBID. It is required that both input matrices have identical dimensions. The input matrix MatrixA is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\texttt{MatrixA}_{ij} \quad = \quad \texttt{MatrixA}_{ij} \quad / \quad \texttt{MatrixB}_{ij}.$$

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 4.0 & 8.0 & 0.0 \\ -6.0 & 7.0 & 2.0 \\ -9.0 & 4.0 & 1.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 2.0 & 8.0 & -3.0 \\ -4.0 & -1.0 & 4.0 \\ -2.0 & -4.0 & 2.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixA} = \begin{bmatrix} 2.0 & 1.0 & 0.0 \\ 1.5 & -7.0 & 0.5 \\ 4.5 & -1.0 & 0.5 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix B.

Result

If the parameters are valid, the operator div_element_matrix_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixAID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

div_element_matrix

See also

mult_element_matrix, mult_element_matrix_mod, scale_matrix, scale_matrix_mod

Module

Foundation

invert_matrix (: : MatrixID, MatrixType, Epsilon : MatrixInvID)

Invert a matrix.

The operator invert_matrix computes the inverse of the Matrix defined by the matrix handle MatrixID. A new matrix MatrixInv is generated with the result and the matrix handle MatrixInvID of this matrix is returned. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

For Epsilon = 0, the inverse is computed. The type of the Matrix can be selected via MatrixType. The following values are supported: *'general'* for general, *'symmetric'* for symmetric, *'positive_definite'* for symmetric positive definite, *'tridiagonal'* for tridiagonal, *'upper_triangular'* for upper triangular, *'permuted_upper_triangular'* for permuted upper triangular, *'lower_triangular'* for lower triangular, and *'permuted_lower_triangular'* for permuted lower triangular matrices.

Example 1:

MatrixType = *'general'*, Epsilon = *0*

$$\text{Matrix} = \begin{bmatrix} 1.0 & 3.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 5.0 & 5.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \text{MatrixInv} = \begin{bmatrix} -1.25 & 1.50 & -0.75 \\ -0.50 & 2.00 & -1.50 \\ 1.25 & -2.50 & 1.75 \end{bmatrix}$$

Example 2:

MatrixType = *'upper_triangular'*, Epsilon = *0*

$$\text{Matrix} = \begin{bmatrix} 1.0 & 3.0 & 3.0 \\ 0 & 2.0 & 6.0 \\ 0 & 0 & 10.0 \end{bmatrix}$$

$$\rightarrow \quad \text{MatrixInv} = \begin{bmatrix} 1.00 & -1.50 & 0.60 \\ 0 & 0.50 & -0.30 \\ 0 & 0 & 0.10 \end{bmatrix}$$

Example 3:

MatrixType = *'permuted_upper_triangular'*, Epsilon = *0*

$$\text{Matrix} = \begin{bmatrix} 1.0 & 3.0 & 3.0 \\ 0 & 0 & 10.0 \\ 0 & 2.0 & 6.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixInv} = \begin{bmatrix} 1.00 & -1.50 & 0.60 \\ 0 & 0.50 & -0.30 \\ 0 & 0 & 0.10 \end{bmatrix}$$

For Epsilon > 0, the pseudo inverse is computed using a singular value decomposition (SVD). During the computation, all singular values less than the value Epsilon × the largest singular value are set to 0. For these values no internal division is done to prevent a division by zero. If a square matrix is computed with the SVD algorithm the computation takes more time. The type of the matrix must be set to MatrixType = *'general'*.

Example:

MatrixType = *'general'*, Epsilon = *2.2204e-16*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 & 5.0 \\ -5.0 & 7.0 & 2.0 & -6.0 \\ -9.0 & -4.0 & 1.0 & 4.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixInv} = \begin{bmatrix} -0.0021 & -0.0482 & -0.0813 \\ 0.1435 & 0.1137 & -0.0137 \\ -0.0519 & -0.0015 & 0.0028 \\ 0.1518 & 0.0056 & 0.0526 \end{bmatrix}$$

Note: The relative accuracy of the floating point representation of the used data type (double) is Epsilon = 2.2204e-16.

It should be also noted that in the examples there are differences in the meaning of the numbers of the output matrices: The results of the elements are per definition a certain value if the number of this value is shown as an integer number, e.g., 0 or 1. If the number is shown as a floating point number, e.g., 0.0 or 1.0, the value is computed.

Attention

For MatrixType = *'symmetric'*, *'positive_definite'*, or *'upper_triangular'* the upper triangular part of the input Matrix must contain the relevant information of the matrix. The strictly lower triangular part of the matrix is not referenced. For MatrixType = *'lower_triangular'* the lower triangular part of the input Matrix must contain the relevant information of the matrix. The strictly upper triangular part of the matrix is not referenced. For MatrixType = *'tridiagonal'*, only the main diagonal, the superdiagonal, and the subdiagonal of the input Matrix are used. The other parts of the matrix are not referenced. If the referenced part of the input Matrix is not of the specified type, an exception is raised.

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixType** (input_control) . . . . . string ⇝ *string*
The type of the input matrix.
Default: 'general'
List of values: MatrixType ∈ {'general', 'symmetric', 'positive_definite', 'tridiagonal', 'upper_triangular', 'permuted_upper_triangular', 'lower_triangular', 'permuted_lower_triangular'}

▷ **Epsilon** (input_control) . . . . . real ⇝ *real*
Type of inversion.
Default: 0.0
Suggested values: Epsilon ∈ {0.0, 2.2204e-16}

▷ **MatrixInvID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the inverse matrix.

Result

If the parameters are valid, the operator invert_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

invert_matrix_mod

See also

transpose_matrix, transpose_matrix_mod

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

invert_matrix_mod (: : MatrixID, MatrixType, Epsilon :)

Invert a matrix.

The operator invert_matrix_mod computes the inverse of the Matrix defined by the matrix handle MatrixID. The input matrix is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

For Epsilon = 0, the inverse is computed. The type of the Matrix can be selected via MatrixType. The following values are supported: *'general'* for general, *'symmetric'* for symmetric, *'positive_definite'* for symmetric positive definite, *'tridiagonal'* for tridiagonal, *'upper_triangular'* for upper triangular, *'permuted_upper_triangular'* for permuted upper triangular, *'lower_triangular'* for lower triangular, and *'permuted_lower_triangular'* for permuted lower triangular matrices.

Example 1:

MatrixType = *'general'*, Epsilon = *0*

$$\mathtt{Matrix} = \begin{bmatrix} 1.0 & 3.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 5.0 & 5.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{Matrix} = \begin{bmatrix} -1.25 & 1.50 & -0.75 \\ -0.50 & 2.00 & -1.50 \\ 1.25 & -2.50 & 1.75 \end{bmatrix}$$

Example 2:

MatrixType = *'upper_triangular'*, Epsilon = *0*

$$\mathtt{Matrix} = \begin{bmatrix} 1.0 & 3.0 & 3.0 \\ 0 & 2.0 & 6.0 \\ 0 & 0 & 10.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{Matrix} = \begin{bmatrix} 1.00 & -1.50 & 0.60 \\ 0 & 0.50 & -0.30 \\ 0 & 0 & 0.10 \end{bmatrix}$$

Example 3:

MatrixType = *'permuted_upper_triangular'*, Epsilon = *0*

$$\texttt{Matrix} = \begin{bmatrix} 1.0 & 3.0 & 3.0 \\ 0 & 0 & 10.0 \\ 0 & 2.0 & 6.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 1.00 & -1.50 & 0.60 \\ 0 & 0.50 & -0.30 \\ 0 & 0 & 0.10 \end{bmatrix}$$

For Epsilon > 0, the pseudo inverse is computed using a singular value decomposition (SVD). During the computation, all singular values less than the value Epsilon $\times$ the largest singular value are set to 0. For these values no internal division is done to prevent a division by zero. If a square matrix is computed with the SVD algorithm the computation takes more time. The type of the matrix must be set to MatrixType = *'general'*.

Example:

MatrixType = *'general'*, Epsilon = *2.2204e-16*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 & 5.0 \\ -5.0 & 7.0 & 2.0 & -6.0 \\ -9.0 & -4.0 & 1.0 & 4.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Matrix} = \begin{bmatrix} -0.0021 & -0.0482 & -0.0813 \\ 0.1435 & 0.1137 & -0.0137 \\ -0.0519 & -0.0015 & 0.0028 \\ 0.1518 & 0.0056 & 0.0526 \end{bmatrix}$$

Note: The relative accuracy of the floating point representation of the used data type (double) is Epsilon = 2.2204e-16.

It should be also noted that in the examples there are differences in the meaning of the numbers of the output matrices: The results of the elements are per definition a certain value if the number of this value is shown as an integer number, e.g., 0 or 1. If the number is shown as a floating point number, e.g., 0.0 or 1.0, the value is computed.

Attention

For MatrixType = *'symmetric'*, *'positive_definite'*, or *'upper_triangular'* the upper triangular part of the input Matrix must contain the relevant information of the matrix. The strictly lower triangular part of the matrix is not referenced. For MatrixType = *'lower_triangular'* the lower triangular part of the input Matrix must contain the relevant information of the matrix. The strictly upper triangular part of the matrix is not referenced. For MatrixType = *'tridiagonal'*, only the main diagonal, the superdiagonal, and the subdiagonal of the input Matrix are used. The other parts of the matrix are not referenced. If the referenced part of the input Matrix is not of the specified type, an exception is raised.

invert_matrix_mod modifies the content of an already existing matrix.

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixType** (input_control) . . . . . string ⇝ *string*
The type of the input matrix.
Default: 'general'
List of values: MatrixType ∈ {'general', 'symmetric', 'positive_definite', 'tridiagonal', 'upper_triangular', 'permuted_upper_triangular', 'lower_triangular', 'permuted_lower_triangular'}

▷ **Epsilon** (input_control) . . . . . real ⇝ *real*
Type of inversion.
Default: 0.0
Suggested values: Epsilon ∈ {0.0, 2.2204e-16}

Result

If the parameters are valid, the operator `invert_matrix_mod` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

Alternatives

`invert_matrix`

See also

`transpose_matrix`, `transpose_matrix_mod`

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

`mult_element_matrix` `( : : MatrixAID, MatrixBID : MatrixMultID )`

Multiply matrices element-by-element.

The operator `mult_element_matrix` multiplies the two matrices `MatrixA` and `MatrixB` element-by-element. The matrices are defined by the matrix handles `MatrixAID` and `MatrixBID`. It is required that both input matrices have identical dimensions. A new matrix `MatrixMult` is generated with the result. The operator returns the matrix handle `MatrixMultID` of the matrix `MatrixMult`. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`. The formula for the calculation of the result is:

$$\texttt{MatrixMult}_{ij} = \texttt{MatrixA}_{ij} \cdot \texttt{MatrixB}_{ij}.$$

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 4.0 & 8.0 & 0.0 \\ -6.0 & 7.0 & 2.0 \\ -9.0 & 4.0 & 1.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 2.0 & 8.0 & -3.0 \\ -4.0 & -1.0 & 4.0 \\ -2.0 & -4.0 & 2.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixMult} = \begin{bmatrix} 8.0 & 64.0 & 0.0 \\ 24.0 & -7.0 & 8.0 \\ 18.0 & -16.0 & 2.0 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix B.

▷ **MatrixMultID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the multiplied values of the input matrices.

Result

If the parameters are valid, the operator `mult_element_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

Alternatives

`mult_element_matrix_mod`

See also

`div_element_matrix`, `div_element_matrix_mod`, `scale_matrix`, `scale_matrix_mod`

Module

Foundation

mult_element_matrix_mod (: : MatrixAID, MatrixBID :)

Multiply matrices element-by-element.

The operator `mult_element_matrix_mod` multiplies the two matrices `MatrixA` and `MatrixB` element-by-element. The matrices are defined by the matrix handles `MatrixAID` and `MatrixBID`. It is required that both input matrices have identical dimensions. The input matrix `MatrixA` is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`. The formula for the calculation of the result is:

$$\texttt{MatrixA}_{ij} = \texttt{MatrixA}_{ij} \cdot \texttt{MatrixB}_{ij}.$$

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 4.0 & 8.0 & 0.0 \\ -6.0 & 7.0 & 2.0 \\ -9.0 & 4.0 & 1.0 \end{bmatrix} \quad \texttt{MatrixB} = \begin{bmatrix} 2.0 & 8.0 & -3.0 \\ -4.0 & -1.0 & 4.0 \\ -2.0 & -4.0 & 2.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixA} = \begin{bmatrix} 8.0 & 64.0 & 0.0 \\ 24.0 & -7.0 & 8.0 \\ 18.0 & -16.0 & 2.0 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix B.

Result

If the parameters are valid, the operator `mult_element_matrix_mod` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixAID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

Alternatives

`mult_element_matrix`

See also

`div_element_matrix`, `div_element_matrix_mod`, `scale_matrix`, `scale_matrix_mod`

Module

Foundation

mult_matrix (: : MatrixAID, MatrixBID, MultType : MatrixMultID)

Multiply two matrices.

The operator `mult_matrix` computes the product of the input matrices `MatrixA` and `MatrixB` defined by the matrix handles `MatrixAID` and `MatrixBID`. A new matrix `MatrixMult` is generated with the result. The operator returns the matrix handle `MatrixMultID` of the matrix `MatrixMult`. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`. If desired, one or both input matrices will be transposed for the multiplication.

The type of multiplication can be selected via `MultType`:

'AB': The matrices `MatrixA` and `MatrixB` will not be transposed. Therefore, the formula for the calculation of the result is:

$$\texttt{MatrixMult} = \texttt{MatrixA} \cdot \texttt{MatrixB}.$$

The number of columns of the matrix `MatrixA` must be identical to the number of rows of the matrix `MatrixB`.

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & -3.0 \\ 2.0 & -5.0 \\ -3.0 & 2.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 3.0 & -3.0 & 1.0 & 1.0 \\ 2.0 & -1.0 & -2.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMult} = \begin{bmatrix} 3.0 & -6.0 & 9.0 & 6.0 \\ -4.0 & -1.0 & 12.0 & 7.0 \\ -5.0 & 7.0 & -7.0 & -5.0 \end{bmatrix}$$

'ATB': The matrix `MatrixA` will be transposed. The matrix `MatrixB` will not be transposed. Therefore, the formula for the calculation of the result is:

$$\texttt{MatrixMult} = \texttt{MatrixA}^T \cdot \texttt{MatrixB}.$$

The number of rows of the matrix `MatrixA` must be identical to the number of rows of the matrix `MatrixB`.

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & 2.0 & -3.0 \\ -3.0 & -5.0 & 2.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 3.0 & -3.0 & 1.0 & 1.0 \\ 2.0 & -1.0 & -2.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMult} = \begin{bmatrix} 3.0 & -6.0 & 9.0 & 6.0 \\ -4.0 & -1.0 & 12.0 & 7.0 \\ -5.0 & 7.0 & -7.0 & -5.0 \end{bmatrix}$$

'ABT': The matrix `MatrixA` will not be transposed. The matrix `MatrixB` will be transposed. Therefore, the formula for the calculation of the result is:

$$\texttt{MatrixMult} = \texttt{MatrixA} \cdot \texttt{MatrixB}^T.$$

The number of columns of the matrix `MatrixA` must be identical to the number of columns of the matrix `MatrixB`.

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & -3.0 \\ 2.0 & -5.0 \\ -3.0 & 2.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 3.0 & 2.0 \\ -3.0 & -1.0 \\ 1.0 & -2.0 \\ 1.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMult} = \begin{bmatrix} 3.0 & -6.0 & 9.0 & 6.0 \\ -4.0 & -1.0 & 12.0 & 7.0 \\ -5.0 & 7.0 & -7.0 & -5.0 \end{bmatrix}$$

'ATBT': The matrix `MatrixA` and the matrix `MatrixB` will be transposed. Therefore, the formula for the calculation of the result is:

$$\texttt{MatrixMult} = \texttt{MatrixA}^T \cdot \texttt{MatrixB}^T.$$

The number of rows of the matrix `MatrixA` must be identical to the number of columns of the matrix `MatrixB`.

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & 2.0 & -3.0 \\ -3.0 & -5.0 & 2.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 3.0 & 2.0 \\ -3.0 & -1.0 \\ 1.0 & -2.0 \\ 1.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMult} = \begin{bmatrix} 3.0 & -6.0 & 9.0 & 6.0 \\ -4.0 & -1.0 & 12.0 & 7.0 \\ -5.0 & 7.0 & -7.0 & -5.0 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix B.

▷ **MultType** (input_control) . . . . . string ⇝ *string*
Type of the input matrices.
Default: 'AB'
List of values: MultType ∈ {'AB', 'ATB', 'ABT', 'ATBT'}

▷ **MatrixMultID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle of the multiplied matrices.

Result

If the parameters are valid, the operator mult_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

mult_matrix_mod

See also

mult_element_matrix, mult_element_matrix_mod, div_element_matrix, div_element_matrix_mod, transpose_matrix, transpose_matrix_mod

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

mult_matrix_mod (: : MatrixAID, MatrixBID, MultType :)

Multiply two matrices.

The operator mult_matrix computes the product of the input matrices MatrixA and MatrixB defined by the matrix handles MatrixAID and MatrixBID. The input matrix MatrixA is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. If desired, one or both input matrices will be transposed for the multiplication.

The type of multiplication can be selected via MultType:

'AB': The matrices MatrixA and MatrixB will not be transposed. Therefore, the formula for the calculation of the result is:

$$\text{MatrixA} = \text{MatrixA} \cdot \text{MatrixB}.$$

The number of columns of the matrix MatrixA must be identical to the number of rows of the matrix MatrixB.

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & -3.0 \\ 2.0 & -5.0 \\ -3.0 & 2.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 3.0 & -3.0 & 1.0 & 1.0 \\ 2.0 & -1.0 & -2.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixA} = \begin{bmatrix} 3.0 & -6.0 & 9.0 & 6.0 \\ -4.0 & -1.0 & 12.0 & 7.0 \\ -5.0 & 7.0 & -7.0 & -5.0 \end{bmatrix}$$

'ATB': The matrix `MatrixA` will be transposed. The matrix `MatrixB` will not be transposed. Therefore, the formula for the calculation of the result is:

$$\texttt{MatrixA} \quad = \quad \texttt{MatrixA}^T \quad \cdot \quad \texttt{MatrixB}.$$

The number of rows of the matrix `MatrixA` must be identical to the number of rows of the matrix `MatrixB`.
Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & 2.0 & -3.0 \\ -3.0 & -5.0 & 2.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 3.0 & -3.0 & 1.0 & 1.0 \\ 2.0 & -1.0 & -2.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixA} = \begin{bmatrix} 3.0 & -6.0 & 9.0 & 6.0 \\ -4.0 & -1.0 & 12.0 & 7.0 \\ -5.0 & 7.0 & -7.0 & -5.0 \end{bmatrix}$$

'ABT': The matrix `MatrixA` will not be transposed. The matrix `MatrixB` will be transposed. Therefore, the formula for the calculation of the result is:

$$\texttt{MatrixA} \quad = \quad \texttt{MatrixA} \quad \cdot \quad \texttt{MatrixB}^T.$$

The number of columns of the matrix `MatrixA` must be identical to the number of columns of the matrix `MatrixB`.
Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & -3.0 \\ 2.0 & -5.0 \\ -3.0 & 2.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 3.0 & 2.0 \\ -3.0 & -1.0 \\ 1.0 & -2.0 \\ 1.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixA} = \begin{bmatrix} 3.0 & -6.0 & 9.0 & 6.0 \\ -4.0 & -1.0 & 12.0 & 7.0 \\ -5.0 & 7.0 & -7.0 & -5.0 \end{bmatrix}$$

'ATBT': The matrix `MatrixA` and the matrix `MatrixB` will be transposed. Therefore, the formula for the calculation of the result is:

$$\texttt{MatrixA} \quad = \quad \texttt{MatrixA}^T \quad \cdot \quad \texttt{MatrixB}^T.$$

The number of rows of the matrix `MatrixA` must be identical to the number of columns of the matrix `MatrixB`.
Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & 2.0 & -3.0 \\ -3.0 & -5.0 & 2.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 3.0 & 2.0 \\ -3.0 & -1.0 \\ 1.0 & -2.0 \\ 1.0 & -1.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixA} = \begin{bmatrix} 3.0 & -6.0 & 9.0 & 6.0 \\ -4.0 & -1.0 & 12.0 & 7.0 \\ -5.0 & 7.0 & -7.0 & -5.0 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix B.

▷ **MultType** (input_control) . . . . . string ⇝ *string*
Type of the input matrices.
Default: 'AB'
List of values: MultType ∈ {'AB', 'ATB', 'ABT', 'ATBT'}

Result

If the parameters are valid, the operator mult_matrix_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixAID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

mult_matrix

See also

mult_element_matrix, mult_element_matrix_mod, div_element_matrix, div_element_matrix_mod, transpose_matrix, transpose_matrix_mod

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

pow_element_matrix (: : MatrixID, MatrixExpID : MatrixPowID)

Compute the power functions of the elements of a matrix.

The operator pow_element_matrix computes the power of all elements of the input Matrix with the elements of the input matrix MatrixExp. The input matrices Matrix and MatrixExp are defined by the matrix handles MatrixID and MatrixExpID. Both matrices must have identical dimensions. A new matrix MatrixPow is generated with the result. The operator returns the matrix handle MatrixPowID of the matrix MatrixPow. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\mathtt{MatrixPow}_{ij} \quad = \quad \mathtt{Matrix}_{ij}^{\mathtt{MatrixExp}_{ij}}.$$

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix} \qquad \texttt{MatrixExp} = \begin{bmatrix} 1.0 & 2.0 & 2.0 \\ 2.0 & 0.0 & 3.0 \\ 0.5 & 3.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixPow} = \begin{bmatrix} 3.0 & 1.0 & 4.0 \\ 25.0 & 1.0 & 8.0 \\ 3.0 & 64.0 & 1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix of the base.

▷ **MatrixExpID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix with exponents.

▷ **MatrixPowID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the raised power of the input matrix.

Result

If the parameters are valid, the operator `pow_element_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

Alternatives

`pow_element_matrix_mod`, `pow_scalar_element_matrix`, `pow_scalar_element_matrix_mod`

See also

`sqrt_matrix`, `sqrt_matrix_mod`

Module

Foundation

pow_element_matrix_mod (: : MatrixID, MatrixExpID :)

Compute the power functions of the elements of a matrix.

The operator `pow_element_matrix_mod` computes the power of all elements of the input `Matrix` with the elements of the input matrix `MatrixExp`. The input matrices `Matrix` and `MatrixExp` are defined by the matrix handles `MatrixID` and `MatrixExpID`. Both matrices must have identical dimensions. The input `Matrix` is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`. The formula for the calculation of the result is:

$$\texttt{Matrix}_{ij} \quad = \quad \texttt{Matrix}_{ij}{}^{\texttt{MatrixExp}_{ij}}.$$

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix} \qquad \texttt{MatrixExp} = \begin{bmatrix} 1.0 & 2.0 & 2.0 \\ 2.0 & 0.0 & 3.0 \\ 0.5 & 3.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 4.0 \\ 25.0 & 1.0 & 8.0 \\ 3.0 & 64.0 & 1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix of the base.

▷ **MatrixExpID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix with exponents.

Result

If the parameters are valid, the operator pow_element_matrix_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

pow_element_matrix, pow_scalar_element_matrix, pow_scalar_element_matrix_mod

See also

sqrt_matrix, sqrt_matrix_mod

Module

Foundation

pow_matrix (: : MatrixID, MatrixType, Power : MatrixPowID)

Compute the power functions of a matrix.

The operator pow_matrix computes the power of the input Matrix by a constant value. The input Matrix is given by the matrix handle MatrixID. The power value is given by the parameter Power. A new matrix MatrixPow is generated with the result. The operator returns the matrix handle MatrixPowID of the matrix MatrixPow. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

The type of the Matrix can be selected via MatrixType. The following values are supported: *'general'* for general, *'symmetric'* for symmetric, *'positive_definite'* for symmetric positive definite, *'upper_triangular'* for upper triangular, *'permuted_upper_triangular'* for permuted upper triangular, *'lower_triangular'* for lower triangular, and *'permuted_lower_triangular'* for permuted lower triangular matrices. The formula for the calculation of the result is:

$$\texttt{MatrixPow} \quad = \quad \texttt{Matrix}^{\texttt{Power}}.$$

Example:

Power = *[2.0]*, MatrixType = *'general'*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixPow} = \begin{bmatrix} 32.0 & 18.0 & 10.0 \\ 68.0 & 62.0 & 26.0 \\ 56.0 & 41.0 & 27.0 \end{bmatrix}$$

Attention

For MatrixType = *'symmetric'*, *'positive_definite'*, or *'upper_triangular'* the upper triangular part of the input Matrix must contain the relevant information of the matrix. The strictly lower triangular part of the matrix is not referenced. For MatrixType = *'lower_triangular'* the lower triangular part of the input Matrix must contain the relevant information of the matrix. The strictly upper triangular part of the matrix is not referenced. If the referenced part of the input Matrix is not of the specified type, an exception is raised.

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixType** (input_control) . . . . . string ⇝ *string*
The type of the input matrix.
Default: 'general'
List of values: MatrixType ∈ {'general', 'symmetric', 'positive_definite', 'upper_triangular', 'permuted_upper_triangular', 'lower_triangular', 'permuted_lower_triangular'}

▷ **Power** (input_control) . . . . . number ⇝ *real* / integer
The power.
Default: 2.0
Suggested values: Power ∈ {0.1, 0.2, 0.3, 0.5, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0, 10.0}

▷ **MatrixPowID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the raised powered matrix.

Result

If the parameters are valid, the operator pow_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

pow_matrix_mod, eigenvalues_symmetric_matrix, eigenvalues_general_matrix

See also

sqrt_matrix, sqrt_matrix_mod

Module

Foundation

pow_matrix_mod (: : MatrixID, MatrixType, Power :)

Compute the power functions of a matrix.

The operator pow_matrix_mod computes the power of the input Matrix by a constant value. The input Matrix is given by the matrix handle MatrixID. The power value is given by the parameter Power. The input matrix is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

The type of the Matrix can be selected via MatrixType. The following values are supported: *'general'* for general, *'symmetric'* for symmetric, *'positive_definite'* for symmetric positive definite, *'upper_triangular'* for upper triangular, *'permuted_upper_triangular'* for permuted upper triangular, *'lower_triangular'* for lower triangular, and *'permuted_lower_triangular'* for permuted lower triangular matrices. The formula for the calculation of the result is:

$$\mathtt{Matrix} \quad = \quad \mathtt{Matrix}^{\mathtt{Power}}.$$

Example:

Power = *[2.0]*, MatrixType = *'general'*

$$\mathtt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \qquad \mathtt{Matrix} = \begin{bmatrix} 32.0 & 18.0 & 10.0 \\ 68.0 & 62.0 & 26.0 \\ 56.0 & 41.0 & 27.0 \end{bmatrix}$$

Attention

For MatrixType = *'symmetric'*, *'positive_definite'*, or *'upper_triangular'* the upper triangular part of the input Matrix must contain the relevant information of the matrix. The strictly lower triangular part of the matrix is not referenced. For MatrixType = *'lower_triangular'* the lower triangular part of the input Matrix must contain the relevant information of the matrix. The strictly upper triangular part of the matrix is not referenced. If the referenced part of the input Matrix is not of the specified type, an exception is raised.

pow_matrix_mod modifies the content of an already existing matrix.

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixType** (input_control) . . . . . string ⇝ *string*
The type of the input matrix.
Default: 'general'
List of values: MatrixType ∈ {'general', 'symmetric', 'positive_definite', 'upper_triangular', 'permuted_upper_triangular', 'lower_triangular', 'permuted_lower_triangular'}

▷ **Power** (input_control) . . . . . number ⇝ *real* / integer
The power.
Default: 2.0
Suggested values: Power ∈ {0.1, 0.2, 0.3, 0.5, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0, 10.0}

Result

If the parameters are valid, the operator pow_matrix_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

pow_matrix, eigenvalues_symmetric_matrix, eigenvalues_general_matrix

See also

sqrt_matrix, sqrt_matrix_mod

Module

Foundation

pow_scalar_element_matrix (: : MatrixID, Power : MatrixPowID)

Compute the power functions of the elements of a matrix.

The operator pow_scalar_element_matrix computes the power of all elements of the input Matrix by a constant value. The input Matrix is given by the matrix handle MatrixID. The power value Power is given by the parameter Power. A new matrix MatrixPow is generated with the result. The operator returns the matrix handle MatrixPowID of the matrix MatrixPow. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\mathtt{MatrixPow}_{ij} \quad = \quad \mathtt{Matrix}_{ij}{}^{\mathtt{Power}}.$$

Example:

Power = *[2.0]*

$$\mathtt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{MatrixPow} = \begin{bmatrix} 9.0 & 1.0 & 4.0 \\ 25.0 & 49.0 & 4.0 \\ 81.0 & 16.0 & 1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Power** (input_control) . . . . . number ⇝ *real* / integer
The power.
Default: 2.0
Suggested values: Power ∈ {0.1, 0.2, 0.3, 0.5, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0, 10.0}

▷ **MatrixPowID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the raised power of the input matrix.

Result

If the parameters are valid, the operator pow_scalar_element_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

pow_scalar_element_matrix_mod, pow_element_matrix, pow_element_matrix_mod

See also

sqrt_matrix, sqrt_matrix_mod

Module

Foundation

pow_scalar_element_matrix_mod (: : MatrixID, Power :)

Compute the power functions of the elements of a matrix.

The operator pow_scalar_element_matrix_mod computes the power of all elements of the input Matrix by a constant value. The input Matrix is given by the matrix handle MatrixID. The power value is given by the parameter Power. The input matrix is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\mathtt{Matrix}_{ij} \quad = \quad \mathtt{Matrix}_{ij}^{\mathtt{Power}}.$$

Example:

Power = *[2.0]*

$$\mathtt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \qquad \mathtt{Matrix} = \begin{bmatrix} 9.0 & 1.0 & 4.0 \\ 25.0 & 49.0 & 4.0 \\ 81.0 & 16.0 & 1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Power** (input_control) . . . . . number ⇝ *real* / integer
The power.
Default: 2.0
Suggested values: Power ∈ {0.1, 0.2, 0.3, 0.5, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0, 10.0}

Result

If the parameters are valid, the operator pow_scalar_element_matrix_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

pow_scalar_element_matrix, pow_element_matrix, pow_element_matrix_mod

See also

sqrt_matrix, sqrt_matrix_mod

Module

Foundation

scale_matrix (: : MatrixID, Factor : MatrixScaledID)

Scale a matrix.

The operator scale_matrix scales the Matrix by a constant factor. The matrix is defined by the matrix handle MatrixID. A new matrix MatrixScaled is generated with the result. The operator returns the matrix handle MatrixScaledID. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\mathtt{MatrixScaled}_{ij} = \mathtt{Factor} \cdot \mathtt{Matrix}_{ij}.$$

Example:

Factor = *1.5*

$$\mathtt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -2.0 & 6.0 & 2.0 \\ -5.0 & -4.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{MatrixScaled} = \begin{bmatrix} 4.5 & 1.5 & -3.0 \\ -3.0 & 9.0 & 3.0 \\ -7.5 & -6.0 & 1.5 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Factor** (input_control) ..... number ⇝ *real* / integer
Scale factor.
Default: 2.0
Suggested values: Factor ∈ {0.1, 0.2, 0.3, 0.5, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0, 10.0}

▷ **MatrixScaledID** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the scaled elements.

Result

If the parameters are valid, the operator scale_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

scale_matrix_mod

See also

mult_element_matrix, mult_element_matrix_mod, div_element_matrix, div_element_matrix_mod

Module

Foundation

scale_matrix_mod (: : MatrixID, Factor :)

Scale a matrix.

The operator scale_matrix scales the Matrix by a constant factor. The matrix is defined by the matrix handle MatrixID. The input matrix is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\mathtt{Matrix}_{ij} = \mathtt{Factor} \cdot \mathtt{Matrix}_{ij}.$$

Example:

Factor = *1.5*

$$\mathtt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -2.0 & 6.0 & 2.0 \\ -5.0 & -4.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{Matrix} = \begin{bmatrix} 4.5 & 1.5 & -3.0 \\ -3.0 & 9.0 & 3.0 \\ -7.5 & -6.0 & 1.5 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Factor** (input_control) . . . . . number ⇝ *real* / integer
Scale factor.
Default: 2.0
Suggested values: Factor ∈ {0.1, 0.2, 0.3, 0.5, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0, 10.0}

Result

If the parameters are valid, the operator scale_matrix_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

scale_matrix

See also

mult_element_matrix, mult_element_matrix_mod, div_element_matrix, div_element_matrix_mod

Module

Foundation

solve_matrix (: : MatrixLHSID, MatrixLHSType, Epsilon, MatrixRHSID : MatrixResultID)

Compute the solution of a system of equations.

The operator solve_matrix computes the solution of a system of linear equations or of a linear least squares problem. The input matrices MatrixLHS and MatrixRHS are defined by the matrix handles MatrixLHSID and MatrixRHSID. The number of rows of matrices MatrixLHS and MatrixRHS must be identical. The operator returns the matrix handle MatrixResultID of the matrix MatrixResult. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

For linear equation systems, the equations

$$\mathtt{MatrixLHS} \cdot \mathtt{MatrixResult} = \mathtt{MatrixRHS}$$

are solved. Therefore, the matrix MatrixLHS must be a square matrix and the parameter Epsilon must be 0. The type of the matrix MatrixLHS can be selected via the parameter MatrixLHSType. The following values are supported: *'general'* for general, *'symmetric'* for symmetric, *'positive_definite'* for symmetric positive definite, *'tridiagonal'* for tridiagonal, *'upper_triangular'* for upper triangular, *'permuted_upper_triangular'* for permuted upper triangular, *'lower_triangular'* for lower triangular, and *'permuted_lower_triangular'* for permuted lower triangular matrices.

Example:

MatrixLHSType = *'positive_definite'*, Epsilon = *0*

$$\mathtt{MatrixLHS} = \begin{bmatrix} 6.0 & 5.0 & 3.0 \\ 5.0 & 7.0 & 3.0 \\ 3.0 & 3.0 & 4.0 \end{bmatrix} \quad \mathtt{MatrixRHS} = \begin{bmatrix} -1.0 & 6.0 \\ 3.0 & -3.0 \\ 5.0 & 4.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{MatrixResult} = \begin{bmatrix} -2.0 & 3.0 \\ 1.0 & -3.0 \\ 2.0 & 1.0 \end{bmatrix}$$

For linear least squares problems or if Epsilon is not 0, the matrix MatrixLHS need not be a square matrix. The linear least squares problem is solved using the singular value decomposition (SVD) of the matrix MatrixLHS by minimizing

$$||\mathtt{MatrixRHS} - \mathtt{MatrixLHS} \cdot \mathtt{MatrixResult}||.$$

All singular values less than the value Epsilon $\times$ the largest singular value are set to 0. For these values no internal division is done to prevent a division by zero. Also, the matrix MatrixLHS may be rank-deficient. The type of matrix must be selected via MatrixLHSType = *'general'*.

Example:

MatrixLHSType = *'general'*, Epsilon = *2.2204e-16*

$$\texttt{MatrixLHS} = \begin{bmatrix} 6.0 & 5.0 & 3.0 \\ 3.0 & 7.0 & -3.0 \\ 5.0 & 12.0 & 4.0 \\ 5.0 & 4.0 & 12.0 \\ 4.0 & 6.0 & 8.0 \end{bmatrix} \qquad \texttt{MatrixRHS} = \begin{bmatrix} 29.0 \\ 10.0 \\ 35.0 \\ 43.0 \\ 25.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixResult} = \begin{bmatrix} 3.4914 \\ 0.7114 \\ 1.6213 \end{bmatrix}$$

Note: The relative accuracy of the floating point representation of the used data type (double) is Epsilon = 2.2204e-16.

Attention

For MatrixLHSType = *'symmetric'*, *'positive_definite'*, or *'upper_triangular'* the upper triangular part of the input MatrixLHS must contain the relevant information of the matrix. The strictly lower triangular part of the matrix is not referenced. For MatrixLHSType = *'lower_triangular'* the lower triangular part of the input MatrixLHS must contain the relevant information of the matrix. The strictly upper triangular part of the matrix is not referenced. For MatrixLHSType = *'tridiagonal'*, only the main diagonal, the superdiagonal, and the subdiagonal of the input MatrixLHS are used. The other parts of the matrix are not referenced. If the referenced part of the input MatrixLHS is not of the specified type, an exception is raised.

Parameters

- ▷ **MatrixLHSID** (input_control) . . . . . matrix ⇝ *handle*
  Matrix handle of the input matrix of the left hand side.
- ▷ **MatrixLHSType** (input_control) . . . . . string ⇝ *string*
  The type of the input matrix of the left hand side.
  Default: 'general'
  List of values: MatrixLHSType ∈ {'general', 'symmetric', 'positive_definite', 'tridiagonal', 'upper_triangular', 'permuted_upper_triangular', 'lower_triangular', 'permuted_lower_triangular'}
- ▷ **Epsilon** (input_control) . . . . . real ⇝ *real*
  Type of solving and limitation to set singular values to be 0.
  Default: 0.0
  Suggested values: Epsilon ∈ {0.0, 2.2204e-16}
- ▷ **MatrixRHSID** (input_control) . . . . . matrix ⇝ *handle*
  Matrix handle of the input matrix of right hand side.
- ▷ **MatrixResultID** (output_control) . . . . . matrix ⇝ *handle*
  New matrix handle with the solution.

Result

If the parameters are valid, the operator solve_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: “Matrix Computations”; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

sqrt_matrix (: : MatrixID : MatrixSqrtID)

Compute the square root values of the elements of a matrix.

The operator sqrt_matrix computes the square root values of all elements of the input Matrix given by the matrix handle MatrixID. A new matrix MatrixSqrt is generated with the result. The operator returns the matrix handle MatrixSqrtID of the matrix MatrixSqrt. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\mathtt{MatrixSqrt}_{ij} \quad = \quad \sqrt{\mathtt{Matrix}_{ij}}.$$

Example:

$$\mathtt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix} \quad \rightarrow \quad \mathtt{MatrixSqrt} = \begin{bmatrix} 1.7321 & 1.0000 & 1.4142 \\ 2.2361 & 2.6458 & 1.4142 \\ 3.0000 & 2.0000 & 1.0000 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixSqrtID** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the square root values of the input matrix.

Result

If the parameters are valid, the operator sqrt_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

sqrt_matrix_mod

See also

pow_scalar_element_matrix, pow_scalar_element_matrix_mod

Module

Foundation

sqrt_matrix_mod (: : MatrixID :)

Compute the square root values of the elements of a matrix.

The operator sqrt_matrix_mod computes the square root values of all elements of the input Matrix given by the matrix handle MatrixID. The input matrix is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\texttt{Matrix}_{ij} \quad = \quad \sqrt{\texttt{Matrix}_{ij}}.$$

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ 5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix} \quad \rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 1.7321 & 1.0000 & 1.4142 \\ 2.2361 & 2.6458 & 1.4142 \\ 3.0000 & 2.0000 & 1.0000 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

Result

If the parameters are valid, the operator sqrt_matrix_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

sqrt_matrix

Module

Foundation

sub_matrix (: : MatrixAID, MatrixBID : MatrixSubID)

Subtract two matrices.

The operator sub_matrix computes the difference of the input matrices MatrixA and MatrixB given by the matrix handles MatrixAID and MatrixBID. Both matrices must have identical dimensions. A new matrix MatrixSub is generated with the result. The operator returns the matrix handle MatrixSubID of the matrix MatrixSub. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The formula for the calculation of the result is:

$$\texttt{MatrixSub} \quad = \quad \texttt{MatrixA} \quad - \quad \texttt{MatrixB}.$$

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \texttt{MatrixB} = \begin{bmatrix} 2.0 & 8.0 & -3.0 \\ -4.0 & -1.0 & 5.0 \\ 2.0 & -4.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixSub} = \begin{bmatrix} 1.0 & -7.0 & 1.0 \\ -1.0 & 8.0 & -3.0 \\ -11.0 & 0.0 & -6.0 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix B.

▷ **MatrixSubID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the difference of the input matrices.

Result

If the parameters are valid, the operator `sub_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

Alternatives

`sub_matrix_mod`

See also

`add_matrix`, `add_matrix_mod`

Module

Foundation

sub_matrix_mod (: : MatrixAID, MatrixBID :)

Subtract two matrices.

The operator `sub_matrix_mod` computes the difference of the input matrices `MatrixA` and `MatrixB` given by the matrix handles `MatrixAID` and `MatrixBID`. Both matrices must have identical dimensions. The input matrix `MatrixA` is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`. The formula for the calculation of the result is:

$$\texttt{MatrixA} = \texttt{MatrixA} - \texttt{MatrixB}.$$

Example:

$$\texttt{MatrixA} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \texttt{MatrixB} = \begin{bmatrix} 2.0 & 8.0 & -3.0 \\ -4.0 & -1.0 & 5.0 \\ 2.0 & -4.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixA} = \begin{bmatrix} 1.0 & -7.0 & 1.0 \\ -1.0 & 8.0 & -3.0 \\ -11.0 & 0.0 & -6.0 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix B.

Result

If the parameters are valid, the operator `sub_matrix_mod` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixAID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

Alternatives

`sub_matrix`

See also

`add_matrix`, `add_matrix_mod`

Module

Foundation

`transpose_matrix ( : : MatrixID : MatrixTransposedID )`

Transpose a matrix.

The operator `transpose_matrix` returns the transpose of the input `Matrix`. The input matrix is defined by the matrix handle `MatrixID`. A new matrix `MatrixTransposed` is generated with the result and the matrix handle `MatrixTransposedID` of this matrix is returned. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`.

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \rightarrow \quad \texttt{MatrixTransposed} = \begin{bmatrix} 3.0 & -5.0 & -9.0 \\ 1.0 & 7.0 & -4.0 \\ -2.0 & 2.0 & 1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixTransposedID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the transpose of the input matrix.

Result

If the parameters are valid, the operator `transpose_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

transpose_matrix_mod

See also

invert_matrix, invert_matrix_mod

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

transpose_matrix_mod (: : MatrixID :)

Transpose a matrix.

The operator transpose_matrix_mod returns the transpose of the input Matrix. The input matrix is defined by the matrix handle MatrixID. The input matrix is overwritten with the result. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 3.0 & -5.0 & -9.0 \\ 1.0 & 7.0 & -4.0 \\ -2.0 & 2.0 & 1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

Result

If the parameters are valid, the operator transpose_matrix_mod returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

transpose_matrix

See also

invert_matrix, invert_matrix_mod

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

19.3 Creation

clear_matrix (: : MatrixID :)

Free the memory of a matrix.

The operator clear_matrix frees the memory of the matrix Matrix given by the matrix handle MatrixID. After calling clear_matrix, the Matrix can no longer be used. The matrix handle MatrixID becomes invalid.

Parameters

▷ **MatrixID** (input_control) ..... matrix(-array) ⇝ *handle*
Matrix handle.

Result

If the parameters are valid, the operator clear_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MatrixID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_matrix

Module

Foundation

copy_matrix (: : MatrixID : MatrixCopyID)

Copy a matrix.

The operator copy_matrix creates the new matrix MatrixCopy and copies all elements of the input Matrix to this new matrix. The input Matrix is defined by the matrix handle MatrixID. The operator returns the matrix handle MatrixCopyID of the matrix MatrixCopy. Access to the elements of the matrix is possible e.g., with

the operator get_full_matrix. With this operator a matrix can be saved before the matrix is modified by the operators set_value_matrix, set_full_matrix, set_sub_matrix, or set_diagonal_matrix.

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \rightarrow \quad \texttt{MatrixCopy} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixCopyID** (output_control) ..... matrix ⇝ *handle*
Matrix handle of the copied matrix.

Result

If the parameters are valid, the operator copy_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

repeat_matrix

Module

Foundation

create_matrix (: : Rows, Columns, Value : MatrixID)

Create a matrix.

The operator create_matrix creates a new Matrix with Rows rows and Columns columns and returns the matrix handle MatrixID. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix. The parameter Value is a string or a tuple of floating point or integer numbers. Integer numbers are converted to floating point numbers automatically.

If Value = *'identity'*, Rows and Columns must have the identical values and an identity matrix is created.

Example:

Rows = *3*, Columns = *3*, Value = *'identity'*

$$\rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If the parameter Value contains a single value, all elements of the matrix are filled with this value.

Example:

Rows = *3*, Columns = *3*, Value = *7*

$$\rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 7.0 & 7.0 & 7.0 \\ 7.0 & 7.0 & 7.0 \\ 7.0 & 7.0 & 7.0 \end{bmatrix}$$

If Value contains as many values as the main diagonal, all elements of the main diagonal are set to the values of the parameter Value (i.e., the number of elements in Value is identical).

Example:

Rows = *3*, Columns = *4*, Value = *[3,7,1]*

$$\rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 3.0 & 0 & 0 & 0 \\ 0 & 7.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \end{bmatrix}$$

It is also possible to set all elements of the matrix with different values. In this case the parameter Value must contain all values in a row-major order, i.e., stored line by line. In addition, the number of elements of Value must be identical to the number of all elements of the matrix, i.e., Rows × Columns.

Example:

Rows = *3*, Columns = *3*, Value = *[3,1,-2,-5,7,2,-9,-4,1]*

$$\rightarrow \quad \texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix}$$

It should be noted that in the examples there are differences in the meaning of the values of the output matrices: If a value is shown as an integer number, e.g., 0 or 1, the value of this element is per definition this certain value. If the number is shown as a floating point number, e.g., 0.0 or 1.0, the value is computed by the operator.

Parameters

▷ **Rows** (input_control) . . . integer ⇝ *integer*
Number of rows of the matrix.
Default: 3
Suggested values: Rows ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Rows >= 1

▷ **Columns** (input_control) . . . integer ⇝ *integer*
Number of columns of the matrix.
Default: 3
Suggested values: Columns ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 50, 100}
Restriction: Columns >= 1

▷ **Value** (input_control) . . . number(-array) ⇝ *real* / integer / string
Values for initializing the elements of the matrix.
Default: 0
Suggested values: Value ∈ {0, 1, 'identity'}

▷ **MatrixID** (output_control) . . . matrix ⇝ *handle*
Matrix handle.

Result

If the parameters are valid, the operator create_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_full_matrix, clear_matrix

Module

Foundation

repeat_matrix (: : MatrixID, Rows, Columns : MatrixRepeatedID)

Repeat a matrix.

The operator repeat_matrix creates the new matrix MatrixRepeated and copies all elements of the input Matrix n times to this new matrix, where n = Rows · Columns. The new matrix has dimensions Rows × rows of the input Matrix and Columns × columns of the input Matrix. The input Matrix is defined by the matrix handle MatrixID. The operator returns the matrix handle MatrixRepeatedID of the matrix MatrixRepeated. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

Example:

Rows = *2*, Columns = *3*

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & -1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixRepeated} = \begin{bmatrix} 3.0 & -1.0 & -2.0 & 3.0 & -1.0 & -2.0 & 3.0 & -1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 & -5.0 & 7.0 & 2.0 & -5.0 & 7.0 & 2.0 \\ 3.0 & -1.0 & -2.0 & 3.0 & -1.0 & -2.0 & 3.0 & -1.0 & -2.0 \\ -5.0 & 7.0 & 2.0 & -5.0 & 7.0 & 2.0 & -5.0 & 7.0 & 2.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Rows** (input_control) ..... integer ⇝ *integer*
Number of copies of input matrix in row direction.
Default: 2
Suggested values: Rows ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20}
Restriction: Rows >= 1

▷ **Columns** (input_control) ..... integer ⇝ *integer*
Number of copies of input matrix in column direction.
Default: 2
Suggested values: Columns ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20}
Restriction: Columns >= 1

▷ **MatrixRepeatedID** (output_control) ..... matrix ⇝ *handle*
Matrix handle of the repeated copied matrix.

Result

If the parameters are valid, the operator repeat_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

See also

copy_matrix

Module

Foundation

19.4 Decomposition

decompose_matrix (: : MatrixID, MatrixType : Matrix1ID, Matrix2ID)

Decompose a matrix.

The operator decompose_matrix decomposes the square input Matrix given by the matrix handle MatrixID. The results are stored in two generated matrices Matrix1 and Matrix2. The operator returns the matrix handles Matrix1ID and Matrix2ID. Access to the elements of the matrices is possible e.g., with the operator get_full_matrix.

The type of the input Matrix can be selected via the parameter MatrixType. The following values are supported: *'general'* for general, *'symmetric'* for symmetric, *'positive_definite'* for symmetric positive definite, and *'tridiagonal'* for tridiagonal matrices.

The decomposition MatrixType = *'general'* or *'tridiagonal'* is a LU factorization (Lower/Upper) with the form

$$\mathtt{Matrix} = \mathtt{Matrix1} \cdot \mathtt{Matrix2}.$$

The output Matrix1 is a lower triangular matrix with unit diagonal elements and interchanged rows. The output Matrix2 is an upper triangular matrix.

Example for a factorization of a general matrix:

$$\mathtt{Matrix} = \begin{bmatrix} 5.0 & -3.0 & 1.0 \\ 0.0 & 2.0 & -1.0 \\ -5.0 & -1.0 & 5.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{Matrix1} = \begin{bmatrix} 1 & 0 & 0 \\ 0.0 & -0.5 & 1 \\ -1.0 & 1 & 0 \end{bmatrix} \quad \mathtt{Matrix2} = \begin{bmatrix} 5.0 & -3.0 & 1.0 \\ 0 & -4.0 & 6.0 \\ 0 & 0 & 2.0 \end{bmatrix}$$

Example for a factorization of a tridiagonal matrix:

$$\mathtt{Matrix} = \begin{bmatrix} -8.0 & -8.0 & 0.0 & 0.0 \\ -8.0 & 6.0 & -4.0 & 0.0 \\ 0.0 & 7.0 & -2.0 & 7.0 \\ 0.0 & 0.0 & 5.0 & -3.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{Matrix1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1.0 & 1 & 0 & 0 \\ 0.0 & 0.5 & 0.0 & 1 \\ 0.0 & 0.0 & 1 & 0 \end{bmatrix} \quad \mathtt{Matrix2} = \begin{bmatrix} -8.0 & -8.0 & 0.0 & 0.0 \\ 0 & 14.0 & -4.0 & 0.0 \\ 0 & 0 & 5.0 & -3.0 \\ 0 & 0 & 0 & 7.0 \end{bmatrix}$$

For MatrixType = *'symmetric'* the factorization is a UDU^T decomposition (Upper/Diagonal/Upper) with the form

$$\mathtt{Matrix} = \mathtt{Matrix1} \cdot \mathtt{Matrix2} \cdot \mathtt{Matrix1}^T$$

where the output Matrix1 is an upper triangular matrix with interchanged columns. The output matrix Matrix2 is a symmetric block diagonal matrix with 1×1 and 2×2 diagonal blocks.

Example for a factorization of a symmetric matrix:

$$\mathtt{Matrix} = \begin{bmatrix} 3.0 & -2.0 & 7.0 & -1.0 \\ -2.0 & -2.0 & 4.0 & 0.0 \\ 7.0 & 4.0 & 8.0 & 1.0 \\ -1.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Matrix1} = \begin{bmatrix} 0 & 0 & 1 & 0.0 \\ 0 & 1 & 0.0 & 2.0 \\ 1 & -1.0 & -1.0 & -10.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \texttt{Matrix2} = \begin{bmatrix} 27.0 & 0 & 0 & 0 \\ 0 & -2.0 & 0 & 0 \\ 0 & 0 & 3.0 & -1.0 \\ 0 & 0 & -1.0 & 0.0 \end{bmatrix}$$

For MatrixType = *'positive_definite'* a Cholesky factorization is computed with the form

$$\texttt{Matrix} \quad = \quad \texttt{Matrix1} \quad \cdot \quad \texttt{Matrix2}.$$

where the output Matrix1 is a lower triangular matrix and the output matrix Matrix2 is an upper triangular matrix. Furthermore, the Matrix2 is the transpose of the matrix Matrix1.

Example for a factorization of a positive definite matrix:

$$\texttt{Matrix} = \begin{bmatrix} 9.0 & 12.0 & -6.0 \\ 12.0 & 17.0 & -7.0 \\ -6.0 & -7.0 & 14.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Matrix1} = \begin{bmatrix} 3.0 & 0 & 0 \\ 4.0 & 1.0 & 0 \\ -2.0 & 1.0 & 3.0 \end{bmatrix} \qquad \texttt{Matrix2} = \begin{bmatrix} 3.0 & 4.0 & -2.0 \\ 0 & 1.0 & 1.0 \\ 0 & 0 & 3.0 \end{bmatrix}$$

It should be noted that in the examples there are differences in the meaning of the values of the output matrices: If a value is shown as an integer number, e.g., 0 or 1, the value of this element is per definition this certain value. If the number is shown as a floating point number, e.g., 0.0 or 1.0, the value is computed by the operator.

Attention

For MatrixType = *'symmetric'* or *'positive_definite'*, the upper triangular part of the input Matrix must contain the relevant information of the matrix. The strictly lower triangular part of the matrix is not referenced. For MatrixType = *'tridiagonal'*, only the main diagonal, the superdiagonal, and the subdiagonal of the input Matrix are used. The other parts of the matrix are not referenced. If the referenced part of the input Matrix is not of the specified type, an exception is raised.

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MatrixType** (input_control) ..... string ⇝ *string*
Type of the input matrix.
Default: 'general'
List of values: MatrixType ∈ {'general', 'symmetric', 'positive_definite', 'tridiagonal'}

▷ **Matrix1ID** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the output matrix 1.

▷ **Matrix2ID** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the output matrix 2.

Result

If the parameters are valid, the operator decompose_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

orthogonal_decompose_matrix, solve_matrix

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

orthogonal_decompose_matrix (: : MatrixID, DecompositionType, OutputMatricesType, ComputeOrthogonal : MatrixOrthogonalID, MatrixTriangularID)

Perform an orthogonal decomposition of a matrix.

The operator orthogonal_decompose_matrix decomposes the Matrix defined by the matrix handle MatrixID. The results are stored in the two generated matrices MatrixOrthogonal and MatrixTriangular. The operator returns the matrix handles MatrixOrthogonalID and MatrixTriangularID of these two matrices. Access to the elements of the matrices is possible e.g., with the operator get_full_matrix.

For OutputMatricesType = *'full'* all results of the decomposition are stored in the matrices MatrixOrthogonal and MatrixTriangular. For OutputMatricesType = *'reduced'* only a part of result elements of the matrices MatrixOrthogonal and MatrixTriangular are stored. Therefore the sizes of these matrices are smaller than for OutputMatricesType = *'full'*.

For the parameter ComputeOrthogonal = *'true'* both output matrices are computed. For the ComputeOrthogonal = *'false'* only the matrix MatrixTriangular is computed. Thus, the runtime of the operation takes fewer time.

The type of the Matrix can be selected via the parameter DecompositionType. For DecompositionType = *'qr'* a QR decomposition (Quadratic/Right) or for DecompositionType = *'ql'* a QL decomposition (Quadratic/Left) is computed. The decomposition is written as

$$\texttt{Matrix} = \texttt{MatrixOrthogonal} \cdot \texttt{MatrixTriangular}.$$

Example:

DecompositionType = *'qr'*, OutputMatricesType = *'full'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & -3.0 & 4.0 \\ 5.0 & 5.0 & -2.0 \\ -3.0 & 12.0 & 5.0 \\ 3.0 & 4.0 & 7.0 \\ 7.0 & 5.0 & 3.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixOrthogonal} = \begin{bmatrix} -0.5303 & 0.2613 & -0.4277 & -0.5566 & -0.3972 \\ -0.4419 & -0.2920 & -0.6383 & 0.1727 & -0.5312 \\ 0.2652 & -0.8443 & -0.2059 & -0.3961 & -0.1326 \\ -0.2652 & -0.2432 & -0.6008 & 0.7000 & -0.1400 \\ -0.6187 & -0.2729 & 0.0799 & -0.1160 & 0.7231 \end{bmatrix}$$

$$\texttt{MatrixTriangular} = \begin{bmatrix} -11.3137 & -1.5910 & -3.6239 \\ 0 & -14.7129 & -5.1135 \\ 0 & 0 & -7.9824 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Example:

DecompositionType = *'qr'*, OutputMatricesType = *'reduced'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & -3.0 & 4.0 \\ 5.0 & 5.0 & -2.0 \\ -3.0 & 12.0 & 5.0 \\ 3.0 & 4.0 & 7.0 \\ 7.0 & 5.0 & 3.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixOrthogonal} = \begin{bmatrix} -0.5303 & 0.2613 & -0.4277 \\ -0.4419 & -0.2920 & -0.6383 \\ 0.2652 & -0.8443 & -0.2059 \\ -0.2652 & -0.2432 & -0.6008 \\ -0.6187 & -0.2729 & 0.0799 \end{bmatrix}$$

$$\texttt{MatrixTriangular} = \begin{bmatrix} -11.3137 & -1.5910 & -3.6239 \\ 0 & -14.7129 & -5.1135 \\ 0 & 0 & -7.9824 \end{bmatrix}$$

Example:

DecompositionType = *'ql'*, OutputMatricesType = *'full'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & -3.0 & 4.0 \\ 5.0 & 5.0 & -2.0 \\ -3.0 & 12.0 & 5.0 \\ 3.0 & 4.0 & 7.0 \\ 7.0 & 5.0 & 3.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixOrthogonal} = \begin{bmatrix} 0.6806 & 0.0657 & 0.3659 & -0.4932 & -0.3941 \\ 0.1161 & 0.5464 & 0.6091 & 0.5274 & 0.1971 \\ 0.4093 & -0.0832 & -0.4046 & 0.6474 & -0.4927 \\ -0.5362 & 0.4713 & 0.0072 & -0.1208 & -0.6897 \\ -0.2611 & -0.6842 & 0.5757 & 0.2119 & -0.2956 \end{bmatrix}$$

$$\texttt{MatrixTriangular} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 10.5059 & 0 & 0 \\ -1.1429 & 12.4620 & 0 \\ -4.0399 & -7.9812 & -10.1489 \end{bmatrix}$$

Example:

DecompositionType = *'ql'*, OutputMatricesType = *'reduced'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & -3.0 & 4.0 \\ 5.0 & 5.0 & -2.0 \\ -3.0 & 12.0 & 5.0 \\ 3.0 & 4.0 & 7.0 \\ 7.0 & 5.0 & 3.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixOrthogonal} = \begin{bmatrix} 0.3659 & -0.4932 & -0.3941 \\ 0.6091 & 0.5274 & 0.1971 \\ -0.4046 & 0.6474 & -0.4927 \\ 0.0072 & -0.1208 & -0.6897 \\ 0.5757 & 0.2119 & -0.2956 \end{bmatrix}$$

$$\texttt{MatrixTriangular} = \begin{bmatrix} 10.5059 & 0 & 0 \\ -1.1429 & 12.4620 & 0 \\ -4.0399 & -7.9812 & -10.1489 \end{bmatrix}$$

For DecompositionType = *'rq'* a RQ decomposition (Right/Quadratic) or for DecompositionType = *'lq'* a LQ decomposition (Left/Quadratic) is computed. The decomposition is written as

$$\texttt{Matrix} \;=\; \texttt{MatrixTriangular} \;\cdot\; \texttt{MatrixOrthogonal}.$$

Example:

DecompositionType = *'rq'*, OutputMatricesType = *'full'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & 5.0 & -3.0 & 3.0 & 7.0 \\ -3.0 & 5.0 & 12.0 & 4.0 & 5.0 \\ 4.0 & -2.0 & 5.0 & 7.0 & 3.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixOrthogonal} = \begin{bmatrix} 0.6806 & 0.1161 & 0.4093 & -0.5362 & -0.2611 \\ 0.0657 & 0.5464 & -0.0832 & 0.4713 & -0.6842 \\ 0.3659 & 0.6091 & -0.4046 & 0.0072 & 0.5757 \\ -0.4932 & 0.5274 & 0.6474 & -0.1208 & 0.2119 \\ -0.3941 & 0.1971 & -0.4927 & -0.6897 & -0.2956 \end{bmatrix}$$

$$\texttt{MatrixTriangular} = \begin{bmatrix} 0 & 0 & 10.5059 & -1.1429 & -4.0399 \\ 0 & 0 & 0 & 12.4620 & -7.9812 \\ 0 & 0 & 0 & 0 & -10.1489 \end{bmatrix}$$

Example:

DecompositionType = *'rq'*, OutputMatricesType = *'reduced'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & 5.0 & -3.0 & 3.0 & 7.0 \\ -3.0 & 5.0 & 12.0 & 4.0 & 5.0 \\ 4.0 & -2.0 & 5.0 & 7.0 & 3.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixOrthogonal} = \begin{bmatrix} 0.3659 & 0.6091 & -0.4046 & 0.0072 & 0.5757 \\ -0.4932 & 0.5274 & 0.6474 & -0.1208 & 0.2119 \\ -0.3941 & 0.1971 & -0.4927 & -0.6897 & -0.2956 \end{bmatrix}$$

$$\texttt{MatrixTriangular} = \begin{bmatrix} 10.5059 & -1.1429 & -4.0399 \\ 0 & 12.4620 & -7.9812 \\ 0 & 0 & -10.1489 \end{bmatrix}$$

Example:

DecompositionType = *'lq'*, OutputMatricesType = *'full'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & 5.0 & -3.0 & 3.0 & 7.0 \\ -3.0 & 5.0 & 12.0 & 4.0 & 5.0 \\ 4.0 & -2.0 & 5.0 & 7.0 & 3.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixOrthogonal} = \begin{bmatrix} -0.5303 & -0.4419 & 0.2652 & -0.2652 & -0.6187 \\ 0.2613 & -0.2920 & -0.8443 & -0.2432 & -0.2729 \\ -0.4277 & 0.6383 & -0.2059 & -0.6008 & 0.0799 \\ -0.5566 & 0.1727 & -0.3961 & 0.7000 & -0.1160 \\ -0.3972 & -0.5312 & -0.1326 & -0.1400 & 0.7231 \end{bmatrix}$$

$$\texttt{MatrixTriangular} = \begin{bmatrix} -11.3137 & 0 & 0 & 0 & 0 \\ -1.5910 & -14.7129 & 0 & 0 & 0 \\ -3.6239 & -5.1135 & -7.9824 & 0 & 0 \end{bmatrix}$$

Example:

DecompositionType = *'lq'*, OutputMatricesType = *'reduced'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & 5.0 & -3.0 & 3.0 & 7.0 \\ -3.0 & 5.0 & 12.0 & 4.0 & 5.0 \\ 4.0 & -2.0 & 5.0 & 7.0 & 3.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixOrthogonal} = \begin{bmatrix} -0.5303 & -0.4419 & 0.2652 & -0.2652 & -0.6187 \\ 0.2613 & -0.2920 & -0.8443 & -0.2432 & -0.2729 \\ -0.4277 & 0.6383 & -0.2059 & -0.6008 & 0.0799 \end{bmatrix}$$

$$\texttt{MatrixTriangular} = \begin{bmatrix} -11.3137 & 0 & 0 \\ -1.5910 & -14.7129 & 0 \\ -3.6239 & -5.1135 & -7.9824 \end{bmatrix}$$

It should be noted that in the examples there are differences in the meaning of the numbers of the output matrices: The results of the elements are per definition a certain value if the number of this value is shown as an integer number, e.g., 0 or 1. If the number is shown as a floating point number, e.g., 0.0 or 1.0, the value is computed.

Parameters

- ▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
  Matrix handle of the input matrix.
- ▷ **DecompositionType** (input_control) ..... string ⇝ *string*
  Method of decomposition.
  Default: 'qr'
  List of values: DecompositionType ∈ {'qr', 'rq', 'ql', 'lq'}
- ▷ **OutputMatricesType** (input_control) ..... string ⇝ *string*
  Type of output matrices.
  Default: 'full'
  List of values: OutputMatricesType ∈ {'full', 'reduced'}
- ▷ **ComputeOrthogonal** (input_control) ..... string ⇝ *string*
  Computation of the orthogonal matrix.
  Default: 'true'
  List of values: ComputeOrthogonal ∈ {'true', 'false'}
- ▷ **MatrixOrthogonalID** (output_control) ..... matrix ⇝ *handle*
  Matrix handle with the orthogonal part of the decomposed input matrix.
- ▷ **MatrixTriangularID** (output_control) ..... matrix ⇝ *handle*
  Matrix handle with the triangular part of the decomposed input matrix.

Result

If the parameters are valid, the operator orthogonal_decompose_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

See also

decompose_matrix, solve_matrix

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

svd_matrix (: : MatrixID, SVDType, ComputeSingularVectors : MatrixUID, MatrixSID, MatrixVID)

Compute the singular value decomposition of a matrix.

The operator svd_matrix computes a full or reduced singular value decomposition (SVD) of the Matrix defined by the matrix handle MatrixID. The operator returns the matrix handle MatrixSID of the matrix MatrixS with singular values in descending order. Optionally, the matrices MatrixU with the left and MatrixV with the right singular vectors are computed and the matrix handles MatrixUID and MatrixVID are returned. Access to the elements of the matrices is possible e.g., with the operator get_full_matrix. The SVD is written

$$\mathtt{Matrix} = \mathtt{MatrixU} \cdot \mathtt{MatrixS} \cdot \mathtt{MatrixV}^T.$$

For SVDType = *'full'*, a full SVD is computed.

Example:

SVDType = *'full'*, ComputeSingularVectors = *'both'*

$$\mathtt{Matrix} = \begin{bmatrix} 6.0 & -5.0 \\ 10.0 & 4.0 \\ -3.0 & 5.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{MatrixU} = \begin{bmatrix} -0.5228 & 0.5691 & 0.6346 \\ -0.8070 & -0.5702 & -0.1535 \\ 0.2745 & -0.5924 & 0.7574 \end{bmatrix}$$

$$\mathtt{MatrixS} = \begin{bmatrix} 12.0547 & 0 \\ 0 & 8.1046 \\ 0 & 0 \end{bmatrix}$$

$$\mathtt{MatrixV} = \begin{bmatrix} -0.9980 & -0.0629 \\ 0.0629 & -0.9980 \end{bmatrix}$$

For SVDType = *'reduced'*, a reduced SVD is computed.

Example:

SVDType = *'reduced'*, ComputeSingularVectors = *'both'*

$$\mathtt{Matrix} = \begin{bmatrix} 6.0 & -5.0 \\ 10.0 & 4.0 \\ -3.0 & 5.0 \end{bmatrix}$$

$$\rightarrow \quad \mathtt{MatrixU} = \begin{bmatrix} -0.5228 & 0.5691 \\ -0.8070 & -0.5702 \\ 0.2745 & -0.5924 \end{bmatrix}$$

$$\mathtt{MatrixS} = \begin{bmatrix} 12.0547 & 0 \\ 0 & 8.1046 \end{bmatrix}$$

$$\mathtt{MatrixV} = \begin{bmatrix} -0.9980 & -0.0629 \\ 0.0629 & -0.9980 \end{bmatrix}$$

For ComputeSingularVectors = *'left'*, the matrix MatrixU with the left singular vectors is computed. For ComputeSingularVectors = *'right'*, the matrix MatrixV with the right singular vectors is computed. For ComputeSingularVectors = *'both'*, the matrices MatrixU and MatrixV with the left and right singular vectors are computed.

For ComputeSingularVectors = *'none'*, no matrices with the singular vectors are computed. The matrix MatrixS is a matrix with n rows and one column, where the number n = min(number of rows of the input Matrix, number of columns of the input Matrix).

Example:

SVDType = *'reduced'* or *'full'*, ComputeSingularVectors = *'none'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & -5.0 \\ 10.0 & 4.0 \\ -3.0 & 5.0 \end{bmatrix}$$

$$\rightarrow \qquad \texttt{MatrixS} = \begin{bmatrix} 12.0547 \\ 8.1046 \end{bmatrix}$$

It should be noted that in the examples there are differences in the meaning of the values of the output matrices: If a value is shown as an integer number, e.g., 0 or 1, the value of this element is per definition this certain value. If the number is shown as a floating point number, e.g., 0.0 or 1.0, the value is computed by the operator.

Parameters

- ▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
  Matrix handle of the input matrix.
- ▷ **SVDType** (input_control) ..... string ⇝ *string*
  Type of computation.
  Default: 'full'
  List of values: SVDType ∈ {'full', 'reduced'}
- ▷ **ComputeSingularVectors** (input_control) ..... string ⇝ *string*
  Computation of singular values.
  Default: 'both'
  List of values: ComputeSingularVectors ∈ {'none', 'left', 'right', 'both'}
- ▷ **MatrixUID** (output_control) ..... matrix ⇝ *handle*
  Matrix handle with the left singular vectors.
- ▷ **MatrixSID** (output_control) ..... matrix ⇝ *handle*
  Matrix handle with singular values.
- ▷ **MatrixVID** (output_control) ..... matrix ⇝ *handle*
  Matrix handle with the right singular vectors.

Result

If the parameters are valid, the operator svd_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

19.5 Eigenvalues

eigenvalues_general_matrix (: : MatrixID, ComputeEigenvectors : EigenvaluesRealID, EigenvaluesImagID, EigenvectorsRealID, EigenvectorsImagID)

Compute the eigenvalues and optionally the eigenvectors of a general matrix.

The operator eigenvalues_general_matrix computes all eigenvalues and, optionally, the left or right eigenvectors of a square, general Matrix. The matrix is defined by the matrix handle MatrixID. The computed eigenvectors have the norm 1.

The operator generates the new matrices EigenvaluesReal and EigenvaluesImag with the real and the imaginary parts of the computed eigenvalues. Each matrix has one column and n rows, where n is the number of rows of the input Matrix. In contrast to the operator eigenvalues_symmetric_matrix, the order of the eigenvalues is not defined. The operator returns the matrix handles EigenvaluesRealID and EigenvaluesImagID. If desired, the real and imaginary parts of the computed eigenvectors are stored in the new matrices EigenvectorsReal and EigenvectorsImag. For this, the operator returns valid matrix handles EigenvectorsRealID and EigenvectorsImagID. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

The computation type of eigenvectors can be selected via the parameter ComputeEigenvectors. If ComputeEigenvectors = *'none'*, no eigenvectors are computed. If *'left'* is selected, the left eigenvalues are computed. If *'right'* is selected, the right eigenvalues are computed.

Example:

ComputeEigenvectors = *'right'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & 4.0 & -8.0 \\ 5.0 & 7.0 & 3.0 \\ 4.0 & -1.0 & 4.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{EigenvaluesReal} = \begin{bmatrix} 3.3110 \\ 3.3110 \\ 10.3781 \end{bmatrix}$$

$$\texttt{EigenvaluesImag} = \begin{bmatrix} 5.4143 \\ -5.4143 \\ 0.0 \end{bmatrix}$$

$$\texttt{EigenvectorsReal} = \begin{bmatrix} 0.6353 & 0.6353 & 0.4813 \\ -0.4764 & -0.4764 & 0.8605 \\ -0.0246 & -0.0246 & 0.1669 \end{bmatrix}$$

$$\texttt{EigenvectorsImag} = \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ -0.2485 & 0.2485 & 0.0 \\ -0.5542 & 0.5542 & 0.0 \end{bmatrix}$$

Parameters

- ▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
  Matrix handle of the input matrix.
- ▷ **ComputeEigenvectors** (input_control) ..... string ⇝ *string*
  Computation of the eigenvectors.
  Default: 'none'
  List of values: ComputeEigenvectors ∈ {'none', 'left', 'right'}
- ▷ **EigenvaluesRealID** (output_control) ..... matrix ⇝ *handle*
  Matrix handle with the real parts of the eigenvalues.
- ▷ **EigenvaluesImagID** (output_control) ..... matrix ⇝ *handle*
  Matrix handle with the imaginary parts of the eigenvalues.
- ▷ **EigenvectorsRealID** (output_control) ..... matrix ⇝ *handle*
  Matrix handle with the real parts of the eigenvectors.

▷ **EigenvectorsImagID** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the imaginary parts of the eigenvectors.

Result

If the parameters are valid, the operator `eigenvalues_general_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`, `get_diagonal_matrix`

See also

`eigenvalues_symmetric_matrix`, `generalized_eigenvalues_symmetric_matrix`, `generalized_eigenvalues_general_matrix`

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

eigenvalues_symmetric_matrix (: : MatrixID, ComputeEigenvectors : EigenvaluesID, EigenvectorsID)

Compute the eigenvalues and optionally eigenvectors of a symmetric matrix.

The operator `eigenvalues_symmetric_matrix` computes all eigenvalues and, optionally, eigenvectors of the symmetric `Matrix`. The matrix is defined by the matrix handle `MatrixID`. On output, a new matrix `Eigenvalues` with the eigenvalues in ascending order and, optionally, a new matrix `Eigenvectors` with the eigenvectors is created. The operator returns the matrix handles `EigenvaluesID` and `EigenvectorsID` of the matrices `Eigenvalues` and `Eigenvectors`. Access to the elements of the matrices is possible e.g., with the operator `get_full_matrix`.

The computation of eigenvectors can be selected via `ComputeEigenvectors` = *'true'* or `ComputeEigenvectors` = *'false'*.

Example:

`ComputeEigenvectors` = *'true'*

$$\texttt{Matrix} = \begin{bmatrix} 6.0 & 5.0 & 3.0 \\ 5.0 & 7.0 & 3.0 \\ 3.0 & 3.0 & 4.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Eigenvalues} = \begin{bmatrix} 1.4195 \\ 2.1507 \\ 13.4298 \end{bmatrix} \qquad \texttt{Eigenvectors} = \begin{bmatrix} 0.7842 & 0.0626 & 0.6174 \\ -0.5667 & 0.4776 & 0.6714 \\ -0.2529 & -0.8763 & 0.4100 \end{bmatrix}$$

Attention

The upper triangular part of the input `Matrix` must contain the relevant information of the matrix. The strictly lower triangular part of the matrix is not referenced. If the referenced part of the input `Matrix` is not of the specified type, an exception is raised.

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **ComputeEigenvectors** (input_control) ..... string ⇝ *string*
Computation of the eigenvectors.
Default: 'false'
List of values: ComputeEigenvectors ∈ {'true', 'false'}

▷ **EigenvaluesID** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the eigenvalues.

▷ **EigenvectorsID** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the eigenvectors.

Result

If the parameters are valid, the operator eigenvalues_symmetric_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

Alternatives

eigenvalues_general_matrix

See also

generalized_eigenvalues_symmetric_matrix,
generalized_eigenvalues_general_matrix

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

generalized_eigenvalues_general_matrix (: : MatrixAID, MatrixBID, ComputeEigenvectors : EigenvaluesRealID, EigenvaluesImagID, EigenvectorsRealID, EigenvectorsImagID)

Compute the generalized eigenvalues and optionally the generalized eigenvectors of general matrices.

The operator generalized_eigenvalues_general_matrix computes all generalized eigenvalues and, optionally, the left or right generalized eigenvectors of the square, general matrices MatrixA and MatrixB. Both matrices must have identical dimensions. The matrices are defined by the matrix handles MatrixAID and MatrixBID. The computed eigenvectors have the norm 1.

The operator generates the new matrices EigenvaluesReal and EigenvaluesImag with the real and the imaginary parts of the computed eigenvalues. Each matrix has one column and n rows, where n is the number of rows or columns of the input matrices. In contrast to the operator generalized_eigenvalues_symmetric_matrix, the order of the generalized eigenvalues is not defined. The operator returns the matrix handles EigenvaluesRealID and EigenvaluesImagID. If desired, the real and imaginary parts of the respective eigenvectors are stored in the new matrices EigenvectorsReal and EigenvectorsImag. Here, the jth column of the matrices of eigenvectors contains the related eigenvector

to the jth eigenvalue. For this, the operator returns additionally the matrix handles EigenvectorsRealID and EigenvectorsImagID. Access to the elements of the matrix is possible, e.g., with the operator get_full_matrix or get_sub_matrix.

The computation type of eigenvectors can be selected via the parameter ComputeEigenvectors. If ComputeEigenvectors = *'none'*, no eigenvectors are computed and the operator is faster. For this, the matrix handles EigenvectorsRealID and EigenvectorsImagID are invalid. If *'right'* is selected, the right generalized eigenvalues are computed. The formula for the calculation of the result is

$$\texttt{MatrixA} \cdot x_j = \lambda_j \cdot \texttt{MatrixB} \cdot x_j,$$

with λ_j representing the jth (complex) eigenvalue and x_j represents the corresponding (complex) eigenvector.

If *'left'* is selected, the left generalized eigenvalues are computed. The formula for the calculation of the result is

$$x^H{}_j \ \texttt{MatrixA} \cdot = x^H{}_j \cdot \lambda_j \cdot \texttt{MatrixB},$$

with $x^H{}_j$ represents the conjugate-transposed of x_j.

Example:

ComputeEigenvectors = *'right'*

$$\texttt{MatrixA} = \begin{bmatrix} 6.0 & 4.0 & -8.0 \\ 5.0 & 7.0 & 3.0 \\ 4.0 & -1.0 & 4.0 \end{bmatrix} \qquad \texttt{MatrixB} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ -5.0 & 7.0 & 2.0 \\ 9.0 & 4.0 & 1.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{EigenvaluesReal} = \begin{bmatrix} 0.5363 \\ 0.5363 \\ -6.1616 \end{bmatrix}$$

$$\texttt{EigenvaluesImag} = \begin{bmatrix} 0.4208 \\ -0.4208 \\ 0.0 \end{bmatrix}$$

$$\texttt{EigenvectorsReal} = \begin{bmatrix} 0.3500 & 0.3500 & 0.0410 \\ -0.9565 & -0.9565 & 0.3267 \\ -0.2757 & -0.2757 & -1.0000 \end{bmatrix}$$

$$\texttt{EigenvectorsImag} = \begin{bmatrix} -0.4644 & 0.4644 & 0.0 \\ 0.0435 & -0.0435 & 0.0 \\ -0.1869 & 0.1869 & 0.0 \end{bmatrix}$$

Parameters

▷ **MatrixAID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix A.

▷ **MatrixBID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix B.

▷ **ComputeEigenvectors** (input_control) . . . . . string ⇝ *string*
Computation of the eigenvectors.
Default: 'none'
List of values: ComputeEigenvectors ∈ {'none', 'left', 'right'}

▷ **EigenvaluesRealID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the real parts of the eigenvalues.

▷ **EigenvaluesImagID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the imaginary parts of the eigenvalues.

▷ **EigenvectorsRealID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the real parts of the eigenvectors.

▷ **EigenvectorsImagID** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the imaginary parts of the eigenvectors.

Result

If the parameters are valid, the operator `generalized_eigenvalues_general_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`, `get_diagonal_matrix`

See also

`generalized_eigenvalues_symmetric_matrix`, `eigenvalues_symmetric_matrix`, `eigenvalues_general_matrix`

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

generalized_eigenvalues_symmetric_matrix (: : MatrixAID, MatrixBID, ComputeEigenvectors : EigenvaluesID, EigenvectorsID)

Compute the generalized eigenvalues and optionally generalized eigenvectors of symmetric input matrices.

The operator `generalized_eigenvalues_symmetric_matrix` computes all generalized eigenvalues and, optionally, generalized eigenvectors of the symmetric matrix `MatrixA` and the symmetric positive definite matrix `MatrixB`. Both matrices must have identical dimensions. The matrices are defined by the matrix handles `MatrixAID` and `MatrixBID`. On output, a new matrix `Eigenvalues` with the generalized eigenvalues in ascending order and, optionally, a new matrix `Eigenvectors` with the generalized eigenvectors is created. Each jth column of the matrix `Eigenvectors` contains the related eigenvector to the jth eigenvalue. The operator returns the matrix handles `EigenvaluesID` and `EigenvectorsID` of the matrices `Eigenvalues` and `Eigenvectors`. Access to the elements of the matrices is possible, e.g., with the operator `get_full_matrix` or `get_sub_matrix`.

The computation of generalized eigenvectors can be selected via `ComputeEigenvectors` = *'true'*. The formula for the calculation of the result is

$$\texttt{MatrixA} \cdot x_j = \lambda_j \cdot \texttt{MatrixB} \cdot x_j,$$

with λ_j representing the jth eigenvalue and x_j represents the corresponding eigenvector.

If `ComputeEigenvectors` = *'false'*, no generalized eigenvectors are computed. For this, the matrix handle `EigenvectorsID` is invalid.

Example:

`ComputeEigenvectors` = *'true'*

$$\texttt{MatrixA} = \begin{bmatrix} 4.0 & -3.0 & 2.0 \\ -3.0 & 1.0 & 4.0 \\ 2.0 & 4.0 & 8.0 \end{bmatrix} \quad \texttt{MatrixB} = \begin{bmatrix} 6.0 & 5.0 & 3.0 \\ 5.0 & 7.0 & 3.0 \\ 3.0 & 3.0 & 4.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{Eigenvalues} = \begin{bmatrix} -0.4463 \\ 1.8747 \\ 4.5472 \end{bmatrix} \quad \texttt{Eigenvectors} = \begin{bmatrix} -0.1579 & 0.1851 & -0.6358 \\ -0.3377 & -0.2568 & 0.4311 \\ 0.2382 & 0.5065 & 0.3185 \end{bmatrix}$$

Attention

The upper triangular parts of the input matrices `MatrixA` and `MatrixB` must contain the relevant information of the matrices. The strictly lower triangular parts of the matrices are not referenced. If the referenced parts of the input matrices `MatrixA` or `MatrixB` are not of the specified type, an exception is raised.

Parameters

- ▷ **`MatrixAID`** (input_control) . . . . . matrix ⇝ *handle*
  Matrix handle of the symmetric input matrix A.
- ▷ **`MatrixBID`** (input_control) . . . . . matrix ⇝ *handle*
  Matrix handle of the symmetric positive definite input matrix B.
- ▷ **`ComputeEigenvectors`** (input_control) . . . . . string ⇝ *string*
  Computation of the eigenvectors.
  Default: 'false'
  List of values: `ComputeEigenvectors` ∈ {'true', 'false'}
- ▷ **`EigenvaluesID`** (output_control) . . . . . matrix ⇝ *handle*
  Matrix handle with the eigenvalues.
- ▷ **`EigenvectorsID`** (output_control) . . . . . matrix ⇝ *handle*
  Matrix handle with the eigenvectors.

Result

If the parameters are valid, the operator `generalized_eigenvalues_symmetric_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

Alternatives

`generalized_eigenvalues_general_matrix`

See also

`eigenvalues_symmetric_matrix`, `eigenvalues_general_matrix`

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

19.6 Features

`determinant_matrix` (: : `MatrixID`, `MatrixType` : `Value`)

Compute the determinant of a matrix.

The operator `determinant_matrix` computes the determinant of the input `Matrix` given by the matrix handle `MatrixID`. The type of the input `Matrix` can be selected via the parameter `MatrixType`. The following values are supported: *'general'* for general, *'symmetric'* for symmetric, *'positive_definite'* for symmetric positive definite, *'tridiagonal'* for tridiagonal, *'upper_triangular'* for upper triangular, *'permuted_upper_triangular'* for permuted upper triangular, *'lower_triangular'* for lower triangular, and *'permuted_lower_triangular'* for permuted lower triangular matrices. The formula for the calculation of the result is:

$$\texttt{Value} \quad = \quad \det \texttt{Matrix}.$$

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 & 2.0 \\ -5.0 & 7.0 & 2.0 \\ -9.0 & -4.0 & 1.0 \end{bmatrix} \quad \rightarrow \quad \texttt{Value} = -134.0$$

Attention

For MatrixType = *'symmetric'*, *'positive_definite'*, or *'upper_triangular'* the upper triangular part of the input Matrix must contain the relevant information of the matrix. The strictly lower triangular part of the matrix is not referenced. For MatrixType = *'lower_triangular'* the lower triangular part of the input Matrix must contain the relevant information of the matrix. The strictly upper triangular part of the matrix is not referenced. For MatrixType = *'tridiagonal'*, only the main diagonal, the superdiagonal, and the subdiagonal of the input Matrix are used. The other parts of the matrix are not referenced. If the referenced part of the input Matrix is not of the specified type, an exception is raised.

Parameters

- ▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
  Matrix handle of the input matrix.
- ▷ **MatrixType** (input_control) . . . . . string ⇝ *string*
  The type of the input matrix.
  Default: 'general'
  List of values: MatrixType ∈ {'general', 'symmetric', 'positive_definite', 'tridiagonal', 'upper_triangular', 'permuted_upper_triangular', 'lower_triangular', 'permuted_lower_triangular'}
- ▷ **Value** (output_control) . . . . . real ⇝ *real*
  Determinant of the input matrix.

Result

If the parameters are valid, the operator determinant_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

get_size_matrix (: : MatrixID : Rows, Columns)

Get the size of a matrix.

The operator get_size_matrix returns the number of rows Rows and columns Columns of the input Matrix. The input Matrix is defined by the matrix handle MatrixID.

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **Rows** (output_control) . . . . . integer ⇝ *integer*
Number of rows of the matrix.

▷ **Columns** (output_control) . . . . . integer ⇝ *integer*
Number of columns of the matrix.

Result

If the parameters are valid, the operator `get_size_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`clear_matrix`

Module

Foundation

max_matrix (: : MatrixID, MaxType : MatrixMaxID)

Returns the elementwise maximum of a matrix.

The operator `max_matrix` returns the maximum values of the elements of the `Matrix` defined by the matrix handle `MatrixID`. A new matrix `MatrixMax` is generated with the result and the matrix handle `MatrixMaxID` of this matrix is returned. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`.

The type of maximum determination of the matrix can be selected via the parameter `MaxType`:

'columns': The maximum is returned for each column of the `Matrix` separately. The resulting matrix `MatrixMax` has one row and the identical number of columns as the input matrix.

Example:

`MaxType` = *'columns'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMax} = \begin{bmatrix} 8.0 & 4.0 & 7.0 \end{bmatrix}$$

'rows': The maximum is returned for each row of the `Matrix` separately. The resulting matrix `MatrixMax` has the identical number of rows as the input matrix and one column.

Example:

`MaxType` = *'rows'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMax} = \begin{bmatrix} 8.0 \\ 7.0 \end{bmatrix}$$

'full': The maximum is returned using all elements of the `Matrix`. The resulting matrix `MatrixMax` has one row and one column.

Example:

`MaxType` = *'full'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMax} = \begin{bmatrix} 8.0 \end{bmatrix}$$

Parameters

▷ **`MatrixID`** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **`MaxType`** (input_control) ..... string ⇝ *string*
Type of maximum determination.
Default: 'columns'
List of values: `MaxType` ∈ {'columns', 'rows', 'full'}

▷ **`MatrixMaxID`** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the maximum values of the input matrix.

Result

If the parameters are valid, the operator `max_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

See also

`min_matrix`

Module

Foundation

`mean_matrix` (: : `MatrixID`, `MeanType` : `MatrixMeanID`)

Returns the elementwise mean of a matrix.

The operator `mean_matrix` returns the mean values of the elements of the `Matrix` defined by the matrix handle `MatrixID`. A new matrix `MatrixMean` is generated with the result and the matrix handle `MatrixMeanID` of this matrix is returned. Access to the elements of the matrix is possible e.g., with the operator `get_full_matrix`.

The type of mean determination of the matrix can be selected via the parameter `MeanType`:

'columns': The mean is returned for each column of the `Matrix` separately. The resulting matrix `MatrixMean` has one row and the identical number of columns as the input matrix.

Example:

`MeanType` = *'columns'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMean} = \begin{bmatrix} 1.0 & 3.0 & 2.0 \end{bmatrix}$$

'rows': The mean is returned for each row of the `Matrix` separately. The resulting matrix `MatrixMean` has the identical number of rows as the input matrix and one column.

Example:

`MeanType` = *'rows'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMean} = \begin{bmatrix} 3.0 \\ 1.0 \end{bmatrix}$$

'full': The mean is returned using all elements of the `Matrix`. The resulting matrix `MatrixMean` has one row and one column.

Example:

`MeanType` = *'full'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMean} = \begin{bmatrix} 2.0 \end{bmatrix}$$

Parameters

▷ **`MatrixID`** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **`MeanType`** (input_control) ..... string ⇝ *string*
Type of mean determination.
Default: 'columns'
List of values: `MeanType` ∈ {'columns', 'rows', 'full'}

▷ **`MatrixMeanID`** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the mean values of the input matrix.

Result

If the parameters are valid, the operator `mean_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

See also

`norm_matrix`, `sum_matrix`

Module

Foundation

min_matrix (: : MatrixID, MinType : MatrixMinID)

Returns the elementwise minimum of a matrix.

The operator min_matrix returns the minimum values of the elements of the Matrix defined by the matrix handle MatrixID. A new matrix MatrixMin is generated with the result and the matrix handle MatrixMinID of this matrix is returned. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

The type of minimum determination of the matrix can be selected via the parameter MinType:

'columns': The minimum is returned for each column of the Matrix separately. The resulting matrix MatrixMin has one row and the identical number of columns as the input matrix.

Example:

MinType = *'columns'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMin} = \begin{bmatrix} -6.0 & 2.0 & -3.0 \end{bmatrix}$$

'rows': The minimum is returned for each row of the Matrix separately. The resulting matrix MatrixMin has the identical number of rows as the input matrix and one column.

Example:

MinType = *'rows'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

'full': The minimum is returned using all elements of the Matrix. The resulting matrix MatrixMin has one row and one column.

Example:

MinType = *'full'*

$$\texttt{MatrixA} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixMin} = \begin{bmatrix} -6.0 \end{bmatrix}$$

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **MinType** (input_control) ..... string ⇝ *string*
Type of minimum determination.
Default: 'columns'
List of values: MinType ∈ {'columns', 'rows', 'full'}

▷ **MatrixMinID** (output_control) ..... matrix ⇝ *handle*
Matrix handle with the minimum values of the input matrix.

Result

If the parameters are valid, the operator min_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

get_full_matrix, get_value_matrix

See also

max_matrix

Module

Foundation

norm_matrix (: : MatrixID, NormType : Value)

Norm of a matrix.

The operator norm_matrix computes the norm of the elements of the Matrix defined by the matrix handle MatrixID. The return value is a floating point number.

The type of norming of the matrix can be selected via the parameter NormType:

'frobenius-norm': The Frobenius norm is computed. The formula for the calculation of the result is:

$$\texttt{Value} = \sqrt{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\texttt{Matrix}_{ij}{}^2}$$

with m = number of rows and n = number of columns of the Matrix.

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 \\ 4.0 & -3.0 \\ -7.0 & 4.0 \end{bmatrix} \quad \rightarrow \quad \texttt{Value} = 10.0$$

'infinity-norm': The infinity norm is computed. The result is the largest value of the sum of the absolute values of the elements of the rows. The formula for the calculation is:

$$\texttt{Value} = \max_{i=0,m-1}\sum_{j=0}^{n-1}|\texttt{Matrix}_{ij}|$$

with m = number of rows and n = number of columns of the Matrix.

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 \\ 4.0 & -3.0 \\ -7.0 & 4.0 \end{bmatrix} \quad \rightarrow \quad \texttt{Value} = 11.0$$

'1-norm': The 1-norm is computed. The result is the largest value of the sum of the absolute values of the elements of the columns. The formula for the calculation is:

$$\texttt{Value} = \max_{j=0,n-1}\sum_{i=0}^{m-1}|\texttt{Matrix}_{ij}|$$

with m = number of rows and n = number of columns of the Matrix.

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 \\ 4.0 & -3.0 \\ -7.0 & 4.0 \end{bmatrix} \quad \rightarrow \quad \texttt{Value} = 14.0$$

'2-norm': The 2-norm is computed. The result is the largest singular value of the Matrix. The formula for the calculation of the result is:

$$\texttt{Value} \quad = \quad \max\left(\text{singular values}\left(\texttt{Matrix}\right)\right)$$

Example:

$$\texttt{Matrix} = \begin{bmatrix} 3.0 & 1.0 \\ 4.0 & -3.0 \\ -7.0 & 4.0 \end{bmatrix} \quad \rightarrow \quad \texttt{Value} = 9.7006$$

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **NormType** (input_control) ..... string ⇝ *string*
Type of norm.
Default: '2-norm'
List of values: NormType ∈ {'2-norm', '1-norm', 'infinity-norm', 'frobenius-norm'}

▷ **Value** (output_control) ..... real ⇝ *real*
Norm of the input matrix.

Result

If the parameters are valid, the operator norm_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

See also

sum_matrix, mean_matrix

References

David Poole: "Linear Algebra: A Modern Introduction"; Thomson; Belmont; 2006.

Gene H. Golub, Charles F. van Loan: "Matrix Computations"; The Johns Hopkins University Press; Baltimore and London; 1996.

Module

Foundation

sum_matrix (: : MatrixID, SumType : MatrixSumID)

Returns the elementwise sum of a matrix.

The operator sum_matrix returns the sum of the elements of the Matrix defined by the matrix handle MatrixID. A new matrix MatrixSum is generated with the result and the matrix handle MatrixSumID of this matrix is returned. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

The type of sum determination of the matrix can be selected via the parameter SumType:

'columns': The sum is returned for each column of the Matrix separately. The resulting matrix MatrixSum has one row and the identical number of columns as the input matrix.

Example:

SumType = *'columns'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixSum} = \begin{bmatrix} 2.0 & 6.0 & 4.0 \end{bmatrix}$$

'rows': The sum is returned for each row of the `Matrix` separately. The resulting matrix `MatrixSum` has the identical number of rows as the input matrix and one column.

Example:

`SumType` = *'rows'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixSum} = \begin{bmatrix} 9.0 \\ 3.0 \end{bmatrix}$$

'full': The sum is returned using all elements of the `Matrix`. The resulting matrix `MatrixSum` has one row and one column.

Example:

`SumType` = *'full'*

$$\texttt{Matrix} = \begin{bmatrix} 8.0 & 4.0 & -3.0 \\ -6.0 & 2.0 & 7.0 \end{bmatrix}$$

$$\rightarrow \quad \texttt{MatrixSum} = \begin{bmatrix} 12.0 \end{bmatrix}$$

Parameters

▷ **`MatrixID`** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **`SumType`** (input_control) . . . . . string ⇝ *string*
Type of summation.
Default: 'columns'
List of values: `SumType` ∈ {'columns', 'rows', 'full'}

▷ **`MatrixSumID`** (output_control) . . . . . matrix ⇝ *handle*
Matrix handle with the sum of the input matrix.

Result

If the parameters are valid, the operator `sum_matrix` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_matrix`

Possible Successors

`get_full_matrix`, `get_value_matrix`

See also

`norm_matrix`

Module

Foundation

19.7 File

deserialize_matrix (: : SerializedItemHandle : MatrixID)

Deserialize a serialized matrix.

deserialize_matrix deserializes a matrix, that was serialized by serialize_matrix (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized matrix is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created matrix with the handle MatrixID. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **MatrixID** (output_control) ..... matrix ⇝ *handle*
Matrix handle.

Result

If the parameters are valid, the operator deserialize_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_matrix

Possible Successors

get_full_matrix

Module

Foundation

read_matrix (: : FileName : MatrixID)

Read a matrix from a file.

The operator read_matrix reads a matrix, which has been written to the file with write_matrix, from the file FileName. The default HALCON file extension for the matrix is 'mtx'. The format of the file can be binary or ascii. Each row of the file contains one row of a matrix with a number of columns. The columns are separated with white spaces or tabs. Each row must have the same number of columns. Empty lines are ignored.

The reading results are stored in the matrix Matrix. The operator returns the matrix handle MatrixID of this matrix. Access to the elements of the matrix is possible e.g., with the operator get_full_matrix.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
File name.
File extension: .mtx

▷ **MatrixID** (output_control) ..... matrix ⇝ *handle*
Matrix handle.

Result

If the file name is valid, the operator read_matrix returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

get_full_matrix

Module

Foundation

serialize_matrix (: : MatrixID : SerializedItemHandle)

Serialize a matrix.

serialize_matrix serializes the data of a matrix (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_matrix is converted to a serialized item. The matrix is defined by the handle MatrixID. The serialized matrix is returned by the handle SerializedItemHandle and can be deserialized by deserialize_matrix.

Parameters

▷ **MatrixID** (input_control) ..... matrix ⇝ *handle*
Matrix handle.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_matrix returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_matrix

Module

Foundation

write_matrix (: : MatrixID, FileFormat, FileName :)

Write a matrix to a file.

The operator write_matrix writes a matrix to the file FileName. The matrix can be read with read_matrix. The default HALCON file extension for the matrix is 'mtx'. The format of the file can be selected via the parameter FileFormat. The following values are supported: *'binary'* for a binary file format, and *'ascii'* for a ascii text format. In the text format each row of the file contains one row of the matrix. The columns are separated with white spaces. The advantage of using the binary file format instead of the ascii text format is a smaller size of the output file.

Parameters

▷ **MatrixID** (input_control) . . . . . matrix ⇝ *handle*
Matrix handle of the input matrix.

▷ **FileFormat** (input_control) . . . . . string ⇝ *string*
Format of the file.
Default: 'binary'
List of values: FileFormat ∈ {'binary', 'ascii'}

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: .mtx

Result

If the file name is valid (write permission), the operator write_matrix returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_matrix

Module

Foundation

Chapter 20

Morphology

20.1 Gray Values

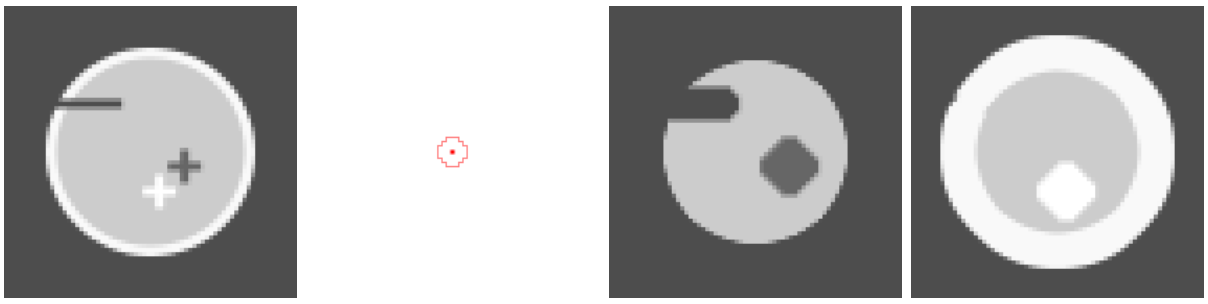
This chapter describes operators of gray value morphology.

Concept of Gray Value Morphology

Gray value morphology provides a set of operators that allow the non-linear manipulation of gray values in an image, depending on their pixel neighborhood. For instance, morphological gray value operators can be used to smooth or emphasize structural features in images. Unlike the binary operations in Morphology / Region, morphological gray value operators deal with input images that contain pixels with a range of more than one bit. Therefore gray value morphology can be seen as a generalization of region morphology. In the following paragraphs, we will take a closer look at the morphological gray value operators.

Dilation and Erosion of a Grayscale Image

To perform a dilation or erosion, each pixel of the image is assigned a gray value depending on its neighborhood. Area and shape of the neighborhood affecting each pixel are defined by the chosen structuring element with the current pixel being the reference point. Implementing a dilation, every pixel of the input image is assigned the maximum gray value of its neighborhood, respectively the minimum gray value for an erosion. Accordingly, bright areas of the input image are enlarged by gray value dilation, whereas gray value erosion emphasizes dark areas.



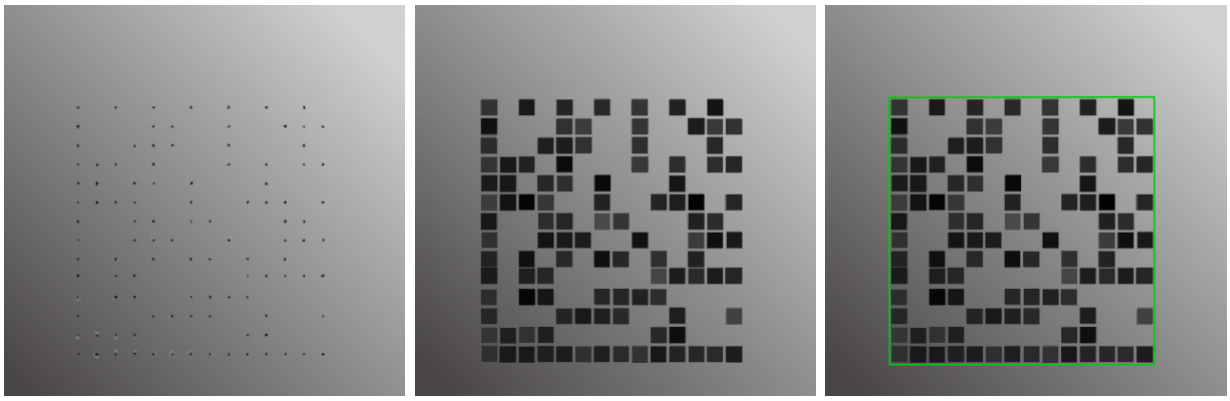
(1) (2) (3) (4)

(1) Original gray value image, (2) structuring element with reference point in the origin, (3) result of eroding the input image, (4) result of dilating the input image.

These operators can be used to dilate or erode an image:

Morphological Operator		Structuring Element
gray_dilation	gray_erosion	arbitrary
gray_dilation_rect	gray_erosion_rect	rectangular
gray_dilation_shape	gray_erosion_shape	rhombus/rectangle/octagon

Morphological grayscale operations are often part of the preprocessing of images before information can be extracted properly. The following example displays a case where a gray value erosion is necessary to read data code symbols. In order to fit a data model used for decoding, the gaps between the code elements in the image need to be reduced by enlarging local minima in a square shape. Therefore a gray value erosion is performed, using an adequately sized rectangle as the structuring element. The rectangle size depends on the data model created with `create_data_code_2d_model`, where the acceptable module gap size is determined.

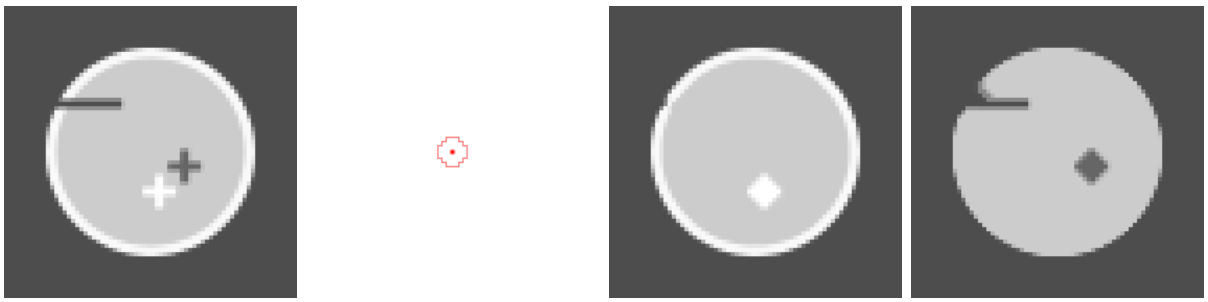


(1) (2) (3)

(1) Image of coded object, (2) erosion with square structuring element, (3) applying data model for decoding. These images are from the example program `2d_data_codes_minimize_module_gaps.hdev`.

Opening and Closing

Gray value opening and gray value closing operators each are a combination of the operators explained above. Closing is a dilation followed by an erosion, while for an opening an erosion precedes a dilation operation. As seen in the example images, `gray_closing` reduces or even removes parts of the image that are darker than their neighborhood whereas `gray_opening` reduces lighter areas. Furthermore, using a suited structuring element you can preserve shapes while removing unwanted image artifacts.



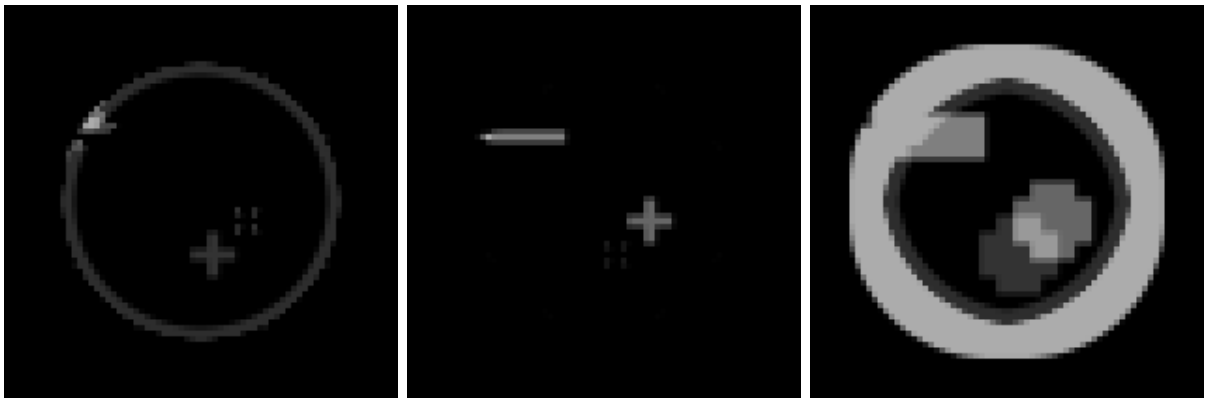
(1) (2) (3) (4)

(1) Original gray value image, (2) structuring element with reference point in the origin, (3) result of closing the input image, (4) result of opening the input image.

Further Operators

To take a closer look at areas that are affected by gray value opening or closing, you can perform a `gray_tophat` or `gray_bothat` transformation. The resulting image displays the difference between the original image and the opening respectively closing of an image. You can also use these operators to detect structures that match the shape of the structuring element.

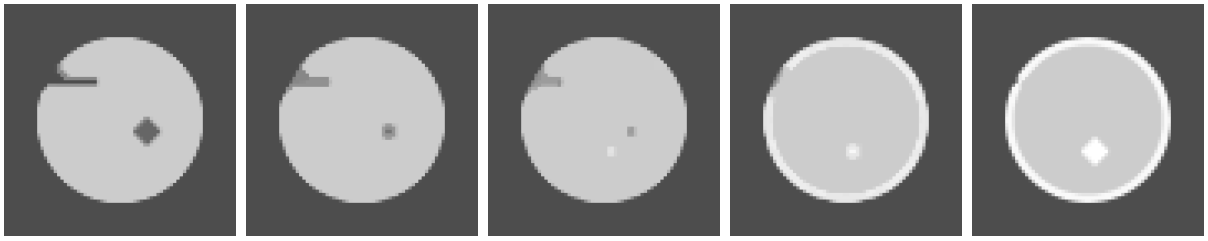
The `gray_range_rect` operator gives you the opportunity to detect fine structures on homogeneous surfaces by visualizing the extent of local variations in pixel values.



(1) (2) (3)

(1) Top hat, (2) bottom hat, (3) gray value range.

By applying the gray_range_rect operator you can perform a mitigated form of a gray value opening or closing operation. You can control the transformation by adjusting the parameter ModePercent.



(1) (2) (3) (4) (5)

Dual rank operations: (1) ModePercent = 0 (equals opening), (2) ModePercent = 25, (3) ModePercent = 50 (equals median), (4) ModePercent = 75, (5) ModePercent = 100 (equals closing).

Glossary

In the following list, the most important terms that are used in the context of Morphology are described.

non-linear operator Operator which does not necessarily preserve structures of the input image

structuring element Region which is used to scan the input image.

```
dual_rank ( Image : ImageRank : MaskType, Radius, ModePercent,
    Margin : )
```

Opening, Median and Closing with circle or rectangle mask.

The operator dual_rank carries out a non-linear transformation of the gray values of all input images (Image). Circles or squares can be used as structuring elements. The operator dual_rank effects two consecutive calls of rank_image. At the first call the range gray value is calculated with the indicated range (ModePercent). The result of this calculation is the input of a further call of rank_image, this time using the range value 100−ModePercent.

When filtering different parameters for border treatment (Margin) can be chosen:

gray value Pixels outside of the image border are assumed to be constant (with the indicated gray value).

'continued' Continuation of the gray values at the image border.

'cyclic' Cyclic continuation at the image borders.

'mirrored' Reflection of pixels at the image borders.

A range filtering is calculated according to the following scheme: The indicated mask is put over the image to be filtered in such a way that the center of the mask touches all pixels once. For each of these pixels all neighboring pixels covered by the mask are sorted in an ascending sequence corresponding to their gray values. Each sorted sequence of gray values contains the same number of gray values like the mask has image points. The n-th highest element, (= `ModePercent`, rank values between 0...100 in percent) is selected and set as result gray value in the corresponding result image.

If `ModePercent` is 0, then the operator equals to the gray value opening (`gray_opening`). If `ModePercent` is 50, the operator results in the median filter, which is applied twice (`median_image`). The `ModePercent` 100 in `dual_rank` means that it calculates the gray value closing (`gray_closing`). Choosing parameter values inside this range results in a smooth transformation of these operators.

For an explanation of the concept of smoothing filters see the introduction of chapter Filters / Smoothing.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **`Image`** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
  Image to be filtered.
- ▷ **`ImageRank`** (output_object) . . . . . multichannel-image(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
  Filtered Image.
- ▷ **`MaskType`** (input_control) . . . . . string ⇝ *string*
  Shape of the mask.
  Default: 'circle'
  List of values: `MaskType` ∈ {'circle', 'square'}
- ▷ **`Radius`** (input_control) . . . . . integer ⇝ *integer*
  Radius of the filter mask.
  Default: 1
  Suggested values: `Radius` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 19, 25, 31, 39, 47, 59}
  Value range: 1 ≤ `Radius` ≤ 101
  Minimum increment: 1
  Recommended increment: 2
- ▷ **`ModePercent`** (input_control) . . . . . integer ⇝ *integer*
  Filter Mode: 0 corresponds to a gray value opening , 50 corresponds to a median and 100 to a gray values closing.
  Default: 10
  Suggested values: `ModePercent` ∈ {0, 2, 5, 10, 15, 20, 40, 50, 60, 80, 85, 90, 95, 98, 100}
  Value range: 0 ≤ `ModePercent` ≤ 100
  Minimum increment: 1
  Recommended increment: 2
- ▷ **`Margin`** (input_control) . . . . . string ⇝ *string* / integer / real
  Border treatment.
  Default: 'mirrored'
  Suggested values: `Margin` ∈ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Example

```
read_image(Image,'fabrik')
dual_rank(Image,ImageOpening,'circle',10,10,'mirrored')
dev_display(ImageOpening)
```

Complexity

For each pixel: O($\sqrt{F} * 10$) with F = area of the structuring element.

Result

If the parameter values are correct the operator `dual_rank` returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system ('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Possible Predecessors

read_image

Possible Successors

threshold, dyn_threshold, sub_image, regiongrowing

Alternatives

rank_image, gray_closing, gray_opening, median_image

See also

gen_circle, gen_rectangle1, gray_erosion_rect, gray_dilation_rect, sigma_image

References

W. Eckstein, O. Munkelt "Extracting Objects from Digital Terrain Model" Remote Sensing and Reconstruction for Threedimensional Objects and Scenes, SPIE Symposium on Optical Science, Engeneering, and Instrumentation, July 1995, San Diego

Module

Foundation

gen_disc_se (: SE : Type, Width, Height, Smax :)

Generate ellipsoidal structuring elements for gray morphology.

gen_disc_se generates an ellipsoidal structuring element (SE) for gray morphology of images. The parameter Type determines the image type. It must match the image type of the image to be processed in subsequent operator calls using this structuring element. The parameters Width and Height determine the length of the two major axes of the ellipse. The value of Smax determines the maximum gray value of the structuring element. For the generation of arbitrary structuring elements, see read_gray_se. The structuring element can be saved as image with the help of the operator write_image. However, take care to use an image format that supports alpha channels to save the shape of the structuring element such as 'tiff', 'jp2' or 'png'. These files can then be loaded again with the operator read_image.

Parameters

▷ **SE** (output_object) ..... image ⇝ *object :* byte / uint2 / real
Generated structuring element.

▷ **Type** (input_control) ..... string ⇝ *string*
Pixel type.
Default: 'byte'
List of values: Type ∈ {'byte', 'uint2', 'real'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the structuring element.
Default: 5
Suggested values: Width ∈ {0, 1, 2, 3, 4, 5, 10, 15, 20}
Value range: 0 ≤ Width ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Height** (input_control) ..... integer ⇝ *integer*
Height of the structuring element.
Default: 5
Suggested values: Height ∈ {0, 1, 2, 3, 4, 5, 10, 15, 20}
Value range: 0 ≤ Height ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Smax** (input_control) . . . . . number ⇝ *real* / integer
Maximum gray value of the structuring element.
Default: 0
Suggested values: Smax ∈ {0, 1, 2, 5, 10, 20, 30, 40}

Result

gen_disc_se returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

gray_erosion, gray_dilation, gray_opening, gray_closing, gray_tophat, gray_bothat, write_image

Alternatives

read_gray_se, read_image

See also

paint_region, paint_gray, crop_part

Module

Foundation

gray_bothat (Image, SE : ImageBotHat : :)

Perform a gray value bottom hat transformation on an image.

gray_bothat applies a gray value bottom hat transformation to the input image Image with the structuring element SE. The image type of the structuring element SE must match the image type of the input image Image. The gray value bottom hat transformation of an image i with a structuring element s is defined as

$$bothat(i, s) = (i \bullet s) - i,$$

i.e., the difference of the closing of the image with s and the image (see gray_closing). For the generation of structuring elements, see read_gray_se.

The gray value erosion is particularly fast for flat structuring elements, i.e. structuring elements with a constant gray level within their domain.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **SE** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2 / real
Structuring element.

▷ **ImageBotHat** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Bottom hat image.

Result

gray_bothat returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

read_gray_se, gen_disc_se, read_image

Possible Successors

threshold

Alternatives

gray_closing

See also

gray_tophat, top_hat, gray_erosion_rect, sub_image

Module

Foundation

gray_closing (Image, SE : ImageClosing : :)

Perform a gray value closing on an image.

gray_closing applies a gray value closing to the input image Image with the structuring element SE. The image type of the structuring element SE must match the image type of the input image Image. The gray value closing of an image i with a structuring element s is defined as

$$i \bullet s = (i \oplus s) \ominus \check{s} \ ,$$

i.e., a dilation of the image with s followed by an erosion with the transposed structuring element (see gray_dilation and gray_erosion). For the generation of structuring elements, see read_gray_se.

The gray value closing is particularly fast for flat structuring elements, i.e., structuring elements with a constant gray level within their domain.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **SE** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
Structuring element.

▷ **ImageClosing** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Gray-closed image.

Result

gray_closing returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

read_gray_se, gen_disc_se, read_image

Alternatives

dual_rank, gray_closing_rect, gray_closing_shape

See also

closing, gray_dilation, gray_erosion

Module

Foundation

gray_closing_rect (Image : ImageClosing : MaskHeight, MaskWidth :)

Perform a gray value closing with a rectangular mask.

gray_closing_rect applies a gray value closing to the input image Image with a rectangular mask of size (MaskHeight, MaskWidth). The resulting image is returned in ImageClosing. If the parameters MaskHeight or MaskWidth are even, they are changed to the next larger odd value. At the border of the image the gray values are mirrored.

The gray value closing of an image i with a rectangular structuring element s is defined as

$$i \circ s = (i \oplus s) \ominus s \ ,$$

i.e., a dilation of the image with s followed by an erosion with s (see gray_dilation_rect and gray_erosion_rect).

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / uint2 / int2 / int4 / real

Input image.

▷ **ImageClosing** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / uint2 / int2 / int4 / real

Gray-closed image.

▷ **MaskHeight** (input_control) ..... extent.y ⇝ *integer*

Height of the filter mask.
Default: 11
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskHeight ≤ 511
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskHeight) && MaskHeight < height(Image) * 2

▷ **MaskWidth** (input_control) ..... extent.x ⇝ *integer*

Width of the filter mask.
Default: 11
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskWidth ≤ 511
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth) && MaskWidth < width(Image) * 2

Result

gray_closing_rect returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

gray_closing, gray_closing_shape

See also

closing_rectangle1, gray_dilation_rect, gray_erosion_rect

Module

Foundation

gray_closing_shape (Image : ImageClosing : MaskHeight, MaskWidth, MaskShape :)

Perform a gray value closing with a selected mask.

gray_closing_shape applies a gray value closing to the input image Image with the structuring element of shape MaskShape. The mask's offset values are 0 and its horizontal and vertical size is defined by MaskHeight and MaskWidth. The resulting image is returned in ImageClosing.

If the parameters MaskHeight or MaskWidth are of the type integer and are even, they are changed to the next larger odd value. In contrast, if at least one of the two parameters is of the type float, the input image Image is transformed with both the next larger and the next smaller odd mask size, and the output image ImageClosing is interpolated from the two intermediate images. Therefore, note that gray_closing_shape returns different results for mask sizes of, e.g., *4* and *4.0*!

In case of the values 'rhombus' and 'octagon' for the MaskShape control parameter, MaskHeight and MaskWidth must be equal. The parameter value 'octagon' for MaskShape denotes an equilateral octagonal mask which is a suitable approximation for a circular structure. At the border of the image the gray values are mirrored.

The gray value closing of an image i with a structuring element s is defined as

$$i \bullet s = (i \oplus s) \ominus s \ ,$$

i.e., a dilation of the image with s followed by an erosion with s (see gray_dilation_shape and gray_erosion_shape).

Attention

Note that gray_closing_shape requires considerably more time for mask sizes of type float than for mask sizes of type integer. This is especially true for rectangular masks with different width and height!

gray_closing_shape can be executed on OpenCL devices. In case of mask sizes of type float the result can vary slightly from the CPU as the interpolation is calculated in single precision on the OpenCL device.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image for which the minimum gray values are to be calculated.

▷ **ImageClosing** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image containing the minimum gray values.

▷ **MaskHeight** (input_control) ..... extent.y ⇝ *real* / integer
Height of the filter mask.
Default: 11
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 1.0 ≤ MaskHeight ≤ 511.0

▷ **MaskWidth** (input_control) ..... extent.x ⇝ *real* / integer
Width of the filter mask.
Default: 11
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 1.0 ≤ MaskWidth ≤ 511.0

▷ **MaskShape** (input_control) ..... string ⇝ *string*
Shape of the mask.
Default: 'octagon'
List of values: MaskShape ∈ {'rectangle', 'rhombus', 'octagon'}

Result

`gray_closing_shape` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

`gray_closing`

See also

`gray_dilation_shape`, `gray_erosion_shape`, `closing`

Module

Foundation

gray_dilation (Image, SE : ImageDilation : :)

Perform a gray value dilation on an image.

`gray_dilation` applies a gray value dilation to the input image `Image` with the structuring element `SE`. The image type of the structuring element `SE` must match the image type of the input image `Image`. The gray value dilation of an image i with a structuring element s at the pixel position x is defined as:

$$(i \oplus s)(x) = \max\{f(x - z) + s(z) | z \in S\}$$

Here, S is the domain of the structuring element s (see `read_gray_se`).

The gray value dilation is particularly fast for flat structuring elements, i.e., structuring elements with a constant gray level within their domain.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **SE** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
Structuring element.

▷ **ImageDilation** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Gray-dilated image.

Result

`gray_dilation` returns `2` (`H_MSG_TRUE`) if the structuring element is not the empty region. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

read_gray_se, gen_disc_se, read_image

Possible Successors

sub_image, gray_erosion

Alternatives

gray_dilation_rect, gray_dilation_shape

See also

gray_opening, gray_closing, dilation1, gray_skeleton

Module

Foundation

gray_dilation_rect (Image : ImageMax : MaskHeight, MaskWidth :)

Determine the maximum gray value within a rectangle.

gray_dilation_rect calculates the maximum gray value of the input image Image within a rectangular mask of size (MaskHeight, MaskWidth) for each image point. The resulting image is returned in ImageMax. If the parameters MaskHeight or MaskWidth are even, they are changed to the next larger odd value. At the border of the image the gray values are mirrored.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / uint2 / int2 / int4 / real
Image for which the maximum gray values are to be calculated.

▷ **ImageMax** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / uint2 / int2 / int4 / real
Image containing the maximum gray values.

▷ **MaskHeight** (input_control) ..... extent.y ⇝ *integer*
Height of the filter mask.
Default: 11
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskHeight ≤ 511
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskHeight) && MaskHeight < height(Image) * 2

▷ **MaskWidth** (input_control) ..... extent.x ⇝ *integer*
Width of the filter mask.
Default: 11
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskWidth ≤ 511
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth) && MaskWidth < width(Image) * 2

Result

gray_dilation_rect returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

gray_skeleton

Module

Foundation

gray_dilation_shape (Image : ImageMax : MaskHeight, MaskWidth, MaskShape :)

Determine the maximum gray value within a selected mask.

gray_dilation_shape calculates the maximum gray value of the input image Image within a mask of shape MaskShape, vertical size MaskHeight and horizontal size MaskWidth for each image point. The resulting image is returned in ImageMax.

If the parameters MaskHeight or MaskWidth are of the type integer and are even, they are changed to the next larger odd value. In contrast, if at least one of the two parameters is of the type float, the input image Image is transformed with both the next larger and the next smaller odd mask size, and the output image ImageMax is interpolated from the two intermediate images. Therefore, note that gray_dilation_shape returns different results for mask sizes of, e.g., *4* and *4.0*!

In case of the values 'rhombus' and 'octagon' for the MaskShape control parameter, MaskHeight and MaskWidth must be equal. The parameter value 'octagon' for MaskShape denotes an equilateral octagonal mask which is a suitable approximation for a circular structure. At the border of the image the gray values are mirrored.

Attention

Note that gray_dilation_shape requires considerably more time for mask sizes of type float than for mask sizes of type integer. This is especially true for rectangular masks with different width and height!

gray_dilation_shape can be executed on OpenCL devices. In case of mask sizes of type float the result can vary slightly from the CPU as the interpolation is calculated in single precision on the OpenCL device.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image for which the maximum gray values are to be calculated.

▷ **ImageMax** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image containing the maximum gray values.

▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *real* / integer
Height of the filter mask.
Default: 11
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 1.0 ≤ MaskHeight

▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *real* / integer
Width of the filter mask.
Default: 11
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 1.0 ≤ MaskWidth

▷ **MaskShape** (input_control) . . . . . string ⇝ *string*
Shape of the mask.
Default: 'octagon'
List of values: MaskShape ∈ {'rectangle', 'rhombus', 'octagon'}

Result

gray_dilation_shape returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

gray_dilation, gray_dilation_rect

See also

gray_opening_shape, gray_closing_shape, gray_skeleton

Module

Foundation

gray_erosion (Image, SE : ImageErosion : :)

Perform a gray value erosion on an image.

gray_erosion applies a gray value erosion to the input image Image with the structuring element SE. The image type of the structuring element SE must match the image type of the input image Image. The gray value erosion of an image i with a structuring element s at the pixel position x is defined as:

$$(i \ominus s)(x) = \min\{f(x+z) - s(z) | z \in S\}$$

Here, S is the domain of the structuring element s (see read_gray_se).

The gray value erosion is particularly fast for flat structuring elements, i.e., structuring elements with a constant gray level within their domain.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **SE** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
  Structuring element.
- ▷ **ImageErosion** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Gray-eroded image.

Result

gray_erosion returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

read_gray_se, gen_disc_se, read_image

Possible Successors

gray_dilation, sub_image

Alternatives

gray_erosion_rect, gray_erosion_shape

See also

gray_opening, gray_closing, erosion1, gray_skeleton

Module

Foundation

gray_erosion_rect (Image : ImageMin : MaskHeight, MaskWidth :)

Determine the minimum gray value within a rectangle.

gray_erosion_rect calculates the minimum gray value of the input image Image within a rectangular mask of size (MaskHeight, MaskWidth) for each image point. The resulting image is returned in ImageMin. If the parameters MaskHeight or MaskWidth are even, they are changed to the next larger odd value. At the border of the image the gray values are mirrored.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / uint2 / int2 / int4 / real
Image for which the minimum gray values are to be calculated.

▷ **ImageMin** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / uint2 / int2 / int4 / real
Image containing the minimum gray values.

▷ **MaskHeight** (input_control) ..... extent.y ⇝ *integer*
Height of the filter mask.
Default: 11
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskHeight ≤ 511 (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskHeight) && MaskHeight < height(Image) * 2

▷ **MaskWidth** (input_control) ..... extent.x ⇝ *integer*
Width of the filter mask.
Default: 11
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskWidth ≤ 511 (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth) && MaskWidth < width(Image) * 2

Result

gray_erosion_rect returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

See also

gray_dilation_rect

Module

Foundation

gray_erosion_shape (Image : ImageMin : MaskHeight, MaskWidth, MaskShape :)

Determine the minimum gray value within a selected mask.

gray_erosion_shape calculates the minimum gray value of the input image Image within a mask of shape MaskShape, vertical size MaskHeight and horizontal size MaskWidth for each image point. The resulting image is returned in ImageMin.

If the parameters MaskHeight or MaskWidth are of the type integer and are even, they are changed to the next larger odd value. In contrast, if at least one of the two parameters is of the type float, the input image Image is transformed with both the next larger and the next smaller odd mask size, and the output image ImageMin is interpolated from the two intermediate images. Therefore, note that gray_erosion_shape returns different results for mask sizes of, e.g., *4* and *4.0*!

In case of the values 'rhombus' and 'octagon' for the MaskShape control parameter, MaskHeight and MaskWidth must be equal. The parameter value 'octagon' for MaskShape denotes an equilateral octagonal mask which is a suitable approximation for a circular structure. At the border of the image the gray values are mirrored.

Attention

Note that gray_erosion_shape requires considerably more time for mask sizes of type float than for mask sizes of type integer. This is especially true for rectangular masks with different width and height!

gray_erosion_shape can be executed on OpenCL devices. In case of mask sizes of type float the result can vary slightly from the CPU as the interpolation is calculated in single precision on the OpenCL device.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image for which the minimum gray values are to be calculated.

▷ **ImageMin** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image containing the minimum gray values.

▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *real* / integer
Height of the filter mask.
Default: 11
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 1.0 ≤ MaskHeight

▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *real* / integer
Width of the filter mask.
Default: 11
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 1.0 ≤ MaskWidth

▷ **MaskShape** (input_control) . . . . . string ⇝ *string*
Shape of the mask.
Default: 'octagon'
List of values: MaskShape ∈ {'rectangle', 'rhombus', 'octagon'}

Result

gray_erosion_shape returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Supports OpenCL compute devices.

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

gray_erosion, gray_erosion_rect

See also

gray_opening_shape, gray_closing_shape, gray_skeleton

Module

Foundation

gray_opening (Image, SE : ImageOpening : :)

Perform a gray value opening on an image.

gray_opening applies a gray value opening to the input image Image with the structuring element SE. The image type of the structuring element SE must match the image type of the input image Image. The gray value opening of an image i with a structuring element s is defined as

$$i \circ s = (i \ominus s) \oplus \check{s} \ ,$$

i.e., an erosion of the image with s followed by a dilation with the transposed structuring element (see gray_erosion and gray_dilation). For the generation of structuring elements, see read_gray_se.

The gray value opening is particularly fast for flat structuring elements, i.e., structuring elements with a constant gray level within their domain.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **SE** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
  Structuring element.
- ▷ **ImageOpening** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
  Gray-opened image.

Result

gray_opening returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

read_gray_se, gen_disc_se, read_image

Alternatives

dual_rank, gray_opening_rect, gray_opening_shape

See also

opening, gray_dilation, gray_erosion

Module

Foundation

gray_opening_rect (Image : ImageOpening : MaskHeight, MaskWidth :)

Perform a gray value opening with a rectangular mask.

gray_opening_rect applies a gray value opening to the input image Image with a rectangular mask of size (MaskHeight, MaskWidth). The resulting image is returned in ImageOpening. If the parameters MaskHeight or MaskWidth are even, they are changed to the next larger odd value. At the border of the image the gray values are mirrored.

The gray value opening of an image i with a rectangular structuring element s is defined as

$$i \circ s = (i \ominus s) \oplus s \ ,$$

i.e., an erosion of the image with s followed by a dilation with s (see gray_erosion_rect and gray_dilation_rect).

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / uint2 / int2 / int4 / real
Input image.

▷ **ImageOpening** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / uint2 / int2 / int4 / real
Gray-opened image.

▷ **MaskHeight** (input_control) ..... extent.y ⇝ *integer*
Height of the filter mask.
Default: 11
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskHeight ≤ 511
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskHeight) && MaskHeight < height(Image) * 2

▷ **MaskWidth** (input_control) ..... extent.x ⇝ *integer*
Width of the filter mask.
Default: 11
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ MaskWidth ≤ 511
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth) && MaskWidth < width(Image) * 2

Result

gray_opening_rect returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

gray_opening, gray_opening_shape

See also

opening_rectangle1, gray_dilation_rect, gray_erosion_rect

Module

Foundation

gray_opening_shape (Image : ImageOpening : MaskHeight, MaskWidth, MaskShape :)

Perform a gray value opening with a selected mask.

gray_opening_shape applies a gray value opening to the input image Image with the structuring element of shape MaskShape. The mask's offset values are 0 and its horizontal and vertical size is defined by MaskHeight and MaskWidth. The resulting image is returned in ImageOpening.

If the parameters MaskHeight or MaskWidth are of the type integer and are even, they are changed to the next larger odd value. In contrast, if at least one of the two parameters is of the type float, the input image Image is transformed with both the next larger and the next smaller odd mask size, and the output image ImageOpening is interpolated from the two intermediate images. Therefore, note that gray_opening_shape returns different results for mask sizes of, e.g., *4* and *4.0*!

In case of the values 'rhombus' and 'octagon' for the MaskShape control parameter, MaskHeight and MaskWidth must be equal. The parameter value 'octagon' for MaskShape denotes an equilateral octagonal mask which is a suitable approximation for a circular structure. At the border of the image the gray values are mirrored.

The gray value opening of an image i with a structuring element s is defined as

$$i \circ s = (i \ominus s) \oplus s \ ,$$

i.e., an erosion of the image with s followed by a dilation with s (see gray_erosion_shape and gray_dilation_shape).

Attention

Note that gray_opening_shape requires considerably more time for mask sizes of type float than for mask sizes of type integer. This is especially true for rectangular masks with different width and height!

gray_opening_shape can be executed on OpenCL devices. In case of mask sizes of type float the result can vary slightly from the CPU as the interpolation is calculated in single precision on the OpenCL device.

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image for which the minimum gray values are to be calculated.

▷ **ImageOpening** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / uint2
Image containing the minimum gray values.

▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *real* / integer
Height of the filter mask.
Default: 11
Suggested values: MaskHeight ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 1.0 ≤ MaskHeight ≤ 511.0

▷ **MaskWidth** (input_control) . . . . . extent.x ⇝ *real* / integer
Width of the filter mask.
Default: 11
Suggested values: MaskWidth ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 1.0 ≤ MaskWidth ≤ 511.0

▷ **MaskShape** (input_control) . . . . . string ⇝ *string*
Shape of the mask.
Default: 'octagon'
List of values: MaskShape ∈ {'rectangle', 'rhombus', 'octagon'}

Result

`gray_opening_shape` returns 2 (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

`gray_opening`

See also

`gray_dilation_shape`, `gray_erosion_shape`, `opening`

Module

Foundation

`gray_range_rect` (Image : ImageResult : MaskHeight, MaskWidth :)

Determine the gray value range within a rectangle.

`gray_range_rect` calculates the gray value range, i.e., the difference $(\max - \min)$ of the maximum and minimum gray values, of the input image `Image` within a rectangular mask of size (`MaskHeight`, `MaskWidth`) for each image point. The resulting image is returned in `ImageResult`. If the parameters `MaskHeight` or `MaskWidth` are even, they are changed to the next smaller odd value. At the border of the image the gray values are mirrored.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **`Image`** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / cyclic / uint2 / int2 / int4 / real
Image for which the gray value range is to be calculated.

▷ **`ImageResult`** (output_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte / cyclic / uint2 / int2 / int4 / real
Image containing the gray value range.

▷ **`MaskHeight`** (input_control) . . . . . extent.y ⇝ *integer*
Height of the filter mask.
Default: 11
Suggested values: `MaskHeight` ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ `MaskHeight` ≤ 511 (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskHeight) && MaskHeight < height(Image) * 2

▷ **`MaskWidth`** (input_control) . . . . . extent.x ⇝ *integer*
Width of the filter mask.
Default: 11
Suggested values: `MaskWidth` ∈ {3, 5, 7, 9, 11, 13, 15}
Value range: 3 ≤ `MaskWidth` ≤ 511 (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: odd(MaskWidth) && MaskWidth < width(Image) * 2

Result

`gray_range_rect` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Supports OpenCL compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on domain level.

Alternatives

`gray_dilation_rect`, `gray_erosion_rect`, `sub_image`

Module

Foundation

gray_tophat (Image, SE : ImageTopHat : :)

Perform a gray value top hat transformation on an image.

`gray_tophat` applies a gray value top hat transformation to the input image `Image` with the structuring element `SE`. The image type of the structuring element `SE` must match the image type of the input image `Image`. The gray value top hat transformation of an image i with a structuring element s is defined as

$$tophat(i, s) = i - (i \circ s),$$

i.e., the difference of the image and its opening with s (see `gray_opening`). For the generation of structuring elements, see `read_gray_se`.

The top hat transformation is particularly fast for flat structuring elements, i.e. structuring elements with a constant gray level within their domain.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **SE** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
Structuring element.

▷ **ImageTopHat** (output_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / uint2 / real
Top hat image.

Result

`gray_tophat` returns `2` (`H_MSG_TRUE`) if the structuring element is not the empty region. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on channel level.
- Automatically parallelized on internal data level.

Possible Predecessors

read_gray_se, gen_disc_se, read_image

Possible Successors

threshold

Alternatives

gray_opening

See also

gray_bothat, top_hat, gray_erosion_rect, sub_image

Module

Foundation

read_gray_se (: SE : FileName :)

Load a structuring element for gray morphology.

read_gray_se loads a structuring element for gray morphology from a file. The file names of these structuring elements must end in '.gse' (for gray-scale structuring element). This suffix is automatically appended by read_gray_se to the passed file name, and thus must not be passed. The structuring element's data must be contained in the file in the following format: The first line specifies the data type of the structuring element as byte, uint2 or real (without enclosing quotes). It must match the image type of the image to be processed in subsequent operator calls using this structuring element. The following two numbers in the file determine the width and height of the structuring element and determine a rectangle enclosing the structuring element. Both values must be greater than 0. Then the number of chords of the region to be defined follows. Then the chords themselves follow: Each chord is defined by the row of the chord (L), the column begin (CB) and the column end (CE), separated by a space between them. Then values follow that are regarded as the corresponding values for gray morphology (their number depends on the defined width and height of the structuring element). Values that will not be part of the region of the structuring element (because no chord is defined for this points) must not be omitted. Nevertheless they will be ignored in the resulting structuring element. Structuring elements are stored internally as images with the data type specified above. Normal images can also be used as structuring elements, where the region of the image defines the shape of the structuring element. However, take care not to use too large images, since the runtime is proportional to the area of the image multiplied by the area of the structuring element. Flat structuring elements (i.e. elements with constant gray values) will be executed particularly fast though. The origin of the structuring element lies at half of the defined width and height. For even dimensions the origin will be rounded towards the upper left corner. The structuring element can be saved as image with the help of the operator write_image. However, take care to use an image format that supports alpha channels to save the shape of the structuring element such as 'tiff', 'jp2' or 'png'. These files can then be loaded again with the operator read_image.

Parameters

▷ **SE** (output_object) ..... image ⇝ *object :* byte / uint2 / real
Generated structuring element.

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the file containing the structuring element.
File extension: .gse

Example

```
* Load a byte structuring element of width 3 and height 3 that
* consists of 3 chords at lines 0 (from x=1 to x=1), 1 (from x=0
* to x=2) and 2 (from x=1 to x=1). The (used) grayvalues are
* 0, 0, 1, 0, 0 .
* contents of file 'isod4.gse':
* byte
*  3 3
*
* 3
* 0 1 1
```

```
* 1 0 2
* 2 1 1
*
*  -1  0 -1
*   0  1  0
*  -1  0 -1
read_gray_se (SE, environment('HALCONROOT')+'/filter/isod4')
```

Result

`read_gray_se` returns `2` (`H_MSG_TRUE`) if all parameters are correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`gray_erosion`, `gray_dilation`, `gray_opening`, `gray_closing`, `gray_tophat`, `gray_bothat`, `write_image`

Alternatives

`gen_disc_se`, `read_image`, `change_domain`

See also

`paint_region`, `paint_gray`, `crop_part`

Module

Foundation

20.2 Region

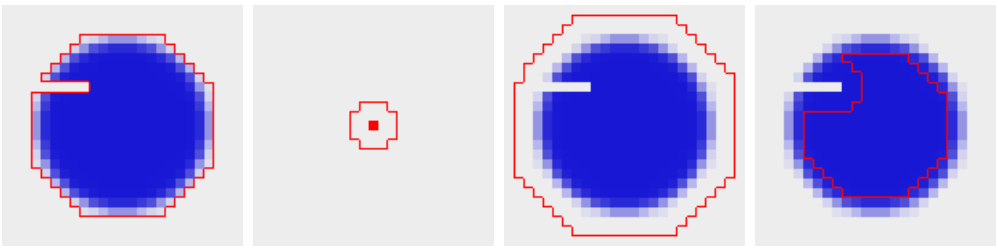
This chapter describes operators of region morphology.

Concept of Region Morphology

Region morphology provides a set of morphological operators that allow to modify or describe the shape of a region. The morphological operators can be used, for example, to connect or disconnect adjacent regions or to smooth the boundary of a region. In the following, we will take a closer look at the morphological operators.

Dilation and Erosion

To dilate or erode an input region, a structuring element is applied to the input region. This structuring element is scanned over the image line-by-line. During dilation the reference point of the structuring element is added to the resulting region whenever the structuring element and the input region have at least one pixel in common. This results in an enlarged region, as shown in the image below. Erosion reduces the area of the input region because the reference point is only added to the resulting region if the structuring element lies completely within the input region. As a result, erosion can alternatively be used to find objects.



(1) (2) (3) (4)

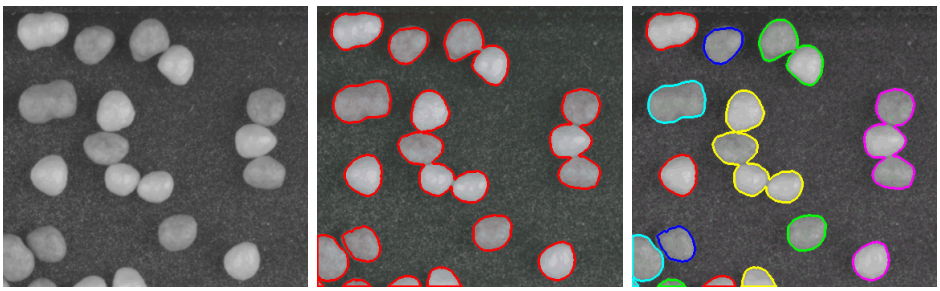
(1) Image with input region, (2) structuring element with reference point in the origin, (3) result of dilating the input region, (4) result of eroding the input region.

These operators can be used to dilate or erode a region:

Morphological Operators		Structuring Element	Reference Point
dilation1	erosion1	arbitrary	origin
minkowski_add1	minkowski_sub1	arbitrary, transposed	origin
dilation2	erosion2	arbitrary	arbitrary
minkowski_add2	minkowski_sub2	arbitrary, transposed	arbitrary
dilation_circle	erosion_circle	circular	origin
dilation_rectangle1	erosion_rectangle1	rectangular	origin

Note that Minkowski addition and dilation are identical if the structuring element is symmetric. The same applies to Minkowski subtraction and erosion. Erosion can be used to separate objects that are attached to each other. In the following, the steps that are required to separate objects are described briefly.

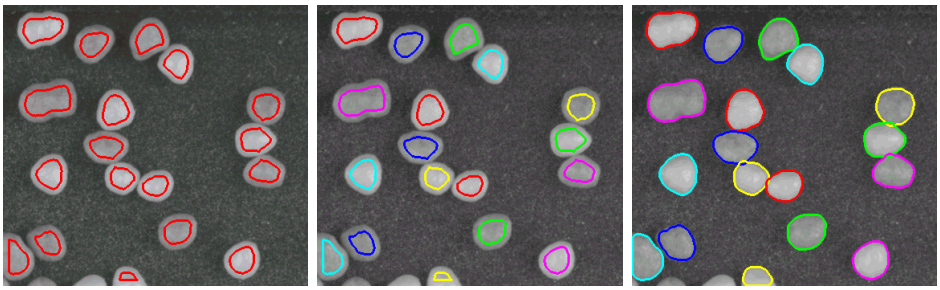
First, the objects of the image must be segmented, for example by using the operator threshold. Next, the operator connection is used to get multiple regions instead of a single region. As you can see in the image (3) below, the result of the connection is unsatisfactory because several objects are merged.



(1) (2) (3)

(1) Image with globular objects, (2) segmented regions, (3) connected components.

This problem can be solved using erosion. As mentioned above, erosion reduces the area of the input region. Thus, if erosion is applied prior to the operator connection, the regions are separated as desired. Lastly, dilation is applied on the separated regions to approximately get the original shape back.

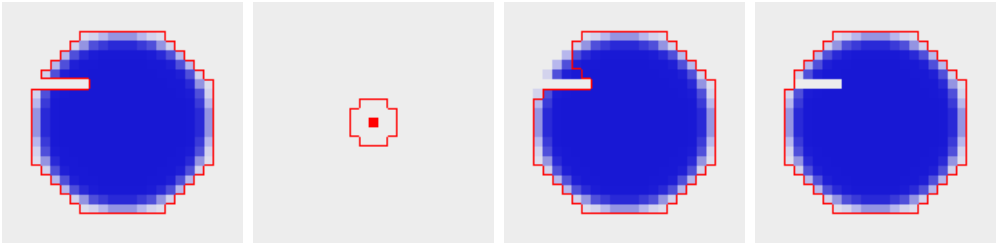


(1) (2) (3)

(1) Segmented regions after erosion, (2) connected components, (3) connected components after dilation.

Opening and Closing

Both operations generate the resulting region by combining dilation and erosion operations. Opening is an erosion followed by a dilation. It is useful to eliminate small unwanted structures. Closing is the opposite of opening, i.e., a dilation followed by an erosion. The closing operator is able to close small gaps, as shown below.



(1) (2) (3) (4)

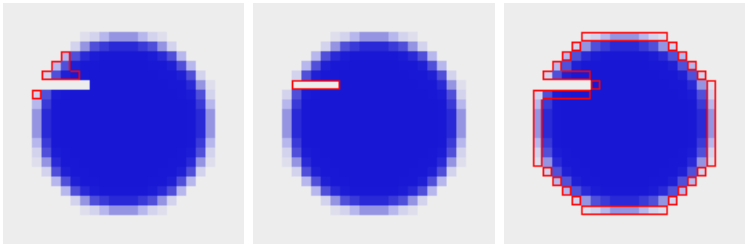
(1) Image with input region, (2) structuring element with reference point, (3) result of applying an opening to the input region, (4) result of applying a closing.

These operators can be used to open or close a region:

Morphological Operators		Structuring Element	Reference Point
opening	closing	arbitrary	origin
opening_circle	closing_circle	circular	origin
opening_rectangle1	closing_rectangle1	rectangular	origin

Further Operators

In addition to the operators mentioned above, you can use top_hat to get the difference between the input region and the result of the opening, or bottom_hat to get the difference between the result of the closing and the input region. Furthermore, you can calculate the boundary of a region with the operator boundary.



(1) (2) (3)

(1) Top hat, (2) bottom hat, (3) boundary.

The operator hit_or_miss can be used to find objects, taking the foreground and the background of the image into account. To remove unwanted branches from a skeleton, pruning is a suitable operator.

Glossary

In the following list, the most important terms that are used in the context of Morphology are described.

input region Region which is modified by morphological operators.

structuring element Region which is used to scan the input region.

bottom_hat (Region, StructElement : RegionBottomHat : :)

Compute the bottom hat of regions.

bottom_hat computes the closing of Region with StructElement. The difference between the result of the closing and the original region is called the bottom hat. In contrast to closing, which merges regions under certain circumstances, bottom_hat computes the regions generated by such a merge.

The position of StructElement is meaningless, since a closing operation is invariant with respect to the choice of the reference point.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element (position independent).

▷ **RegionBottomHat** (output_object) ..... region(-array) ⇝ *object*
Result of the bottom hat operator.

Example

```
read_image (Image, 'brake_disk/brake_disk_bike_01')
threshold (Image, Regions, 128, 255)
gen_circle (Circle, 40, 40, 20)
bottom_hat (Regions, Circle, RegionBottomHat)
```

Result

bottom_hat returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_region_polygon_filled

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

closing, difference

See also

top_hat, gray_bothat, opening

Module

Foundation

boundary (Region : RegionBorder : BoundaryType :)

Reduce a region to its boundary.

boundary computes the boundary of each input region in Region and returns them in RegionBorder. The parameter BoundaryType determines the type of boundary computation.

The contour computation is done using morphological operations. The resulting output regions consist only of the minimal border of the input regions. Their positions depend on the value of BoundaryType, accepting the following values:

- *'inner'*: The contour lies within the original region.
- *'inner_filled'*: The contour lies within the original region, holes in the interior of the input region are suppressed. Due to algorithm optimization this contours may slightly differ from corresponding ones obtained with *'inner'*.
- *'outer'*: The contour is one pixel outside of the original region.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions for which the boundary is to be computed.

▷ **RegionBorder** (output_object) ..... region(-array) ⇝ *object*
Resulting boundaries.

▷ **BoundaryType** (input_control) ..... string ⇝ *string*
Boundary type.
Default: 'inner'
List of values: `BoundaryType` ∈ {'inner', 'outer', 'inner_filled'}

Example

```
#include "HalconCpp.h"
using namespace Halcon;

main()
{
  HWindow  w;
  HRegion  circ1 = HRegion::GenCircle (20, 10, 10.5);

  circ1.Display (w);
  w.Click ();

  HRegion marg1 = circ1.Boundary ("inner");
  w.SetColor ("red");
  marg1.Display (w);
  w.Click ();

  return(0);
}
```

Complexity

Let A be the area of the input region. Then the runtime complexity for one region is

$$O(3\sqrt{A}) \ .$$

Result

`boundary` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

dilation_circle, erosion_circle, difference

See also

fill_up

Module

Foundation

closing (Region, StructElement : RegionClosing : :)

Close a region.

A closing operation is defined as a dilation followed by a Minkowski subtraction. By applying closing to a region, larger structures remain mostly intact, while small gaps between adjacent regions and holes smaller than StructElement are closed, and the regions' boundaries are smoothed. All closing variants share the property that separate regions are not merged, but remain separate objects. The position of StructElement is meaningless, since a closing operation is invariant with respect to the choice of the reference point.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Attention

closing is applied to each input region separately. If gaps between different regions are to be closed, union1 or union2 has to be called first.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Regions to be closed.
- ▷ **StructElement** (input_object) ..... region ⇝ *object*
  Structuring element (position-invariant).
- ▷ **RegionClosing** (output_object) ..... region(-array) ⇝ *object*
  Closed regions.

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"

main()
{
  cout << "Reproduction of 'closing ()' using " << endl;
  cout << "'dilation()' and 'minkowski_sub1()'" << endl;

  HByteImage img("monkey");
  HWindow    w;

  HRegion      circ = HRegion::GenCircle (10, 10, 1.5);
  HRegionArray regs = (img >= 128).Connection();

  HRegionArray dilreg = regs.Dilation1 (circ, 1);
  HRegionArray minsub = dilreg.MinkowskiSub1 (circ, 1);
```

```
                        img.Display (w);      w.Click ();
  w.SetColor ("red");   regs.Display (w);     w.Click ();
  w.SetColor ("green"); dilreg.Display (w);   w.Click ();
  w.SetColor ("blue");  minsub.Display (w);   w.Click ();

  return(0);
}
```

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(2 \cdot \sqrt{F1} \cdot \sqrt{F2}) \ .$$

Result

`closing` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`closing_circle`

See also

`dilation1`, `erosion1`, `opening`, `minkowski_sub1`

Module

Foundation

`closing_circle` (Region : RegionClosing : Radius :)

Close a region with a circular structuring element.

`closing_circle` behaves analogously to `closing`, i.e., the regions' boundaries are smoothed and holes within a region which are smaller than the circular structuring element of radius `Radius` are closed. The `closing_circle` operation is defined as a dilation followed by a Minkowski subtraction, both with the same circular structuring element.

Attention

`closing_circle` is applied to each input region separately. If gaps between different regions are to be closed, `union1` or `union2` has to be called first.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be closed.

▷ **RegionClosing** (output_object) . . . . . region(-array) ⇝ *object*
Closed regions.

▷ **Radius** (input_control) . . . . . real ⇝ *real* / integer
Radius of the circular structuring element.
Default: 3.5
Suggested values: Radius ∈ {1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 9.5, 12.5, 15.5, 19.5, 25.5, 33.5, 45.5, 60.5, 110.5}
Value range: 0.5 ≤ Radius ≤ 511.5 (lin)
Minimum increment: 1.0
Recommended increment: 1.0

Example

```
my_closing_circle(Hobject In, double Radius, Hobject *Out)
{
  Hobject  tmp, StructElement;
  gen_circle(StructElement,100.0,100.0,Radius);
  dilation1(In,StructElement,&tmp,1);
  minkowski_sub1(tmp,StructElement,Out,1);
}
```

Complexity

Let $F1$ be the area of the input region. Then the runtime complexity for one region is:

$$O(4 \cdot \sqrt{F1} \cdot \texttt{Radius}) \ .$$

Result

closing_circle returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

rank_region, fill_up, closing

See also

dilation1, minkowski_sub1, erosion1, opening

Module

Foundation

closing_rectangle1 (Region : RegionClosing : Width, Height :)

Close a region with a rectangular structuring element.

closing_rectangle1 behaves as closing with a rectangular structuring element on the input region Region. The size of the rectangular structuring element is determined by the parameters Width and Height. As is the case for all closing variants, regions' boundaries are smoothed and holes within a region which are smaller than the rectangular structuring element are closed.

Similar to erosion_rectangle1 and dilation_rectangle1 closing_rectangle1 is a very fast operation.

Attention

closing_rectangle1 is applied to each input region separately. If gaps between different regions are to be closed, union1 or union2 has to be called first.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be closed.

▷ **RegionClosing** (output_object) ..... region(-array) ⇝ *object*
Closed regions.

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the structuring rectangle.
Default: 10
Suggested values: Width ∈ {1, 2, 3, 4, 5, 7, 9, 12, 15, 19, 25, 33, 45, 60, 110, 150, 200}
Value range: 1 ≤ Width ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the structuring rectangle.
Default: 10
Suggested values: Height ∈ {1, 2, 3, 4, 5, 7, 9, 12, 15, 19, 25, 33, 45, 60, 110, 150, 200}
Value range: 1 ≤ Height ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of an input region and H be the height of the rectangle. Then the runtime complexity for one region is:

$$O(2 \cdot \sqrt{F1} \cdot log\_2(H)) \ .$$

Result

closing_rectangle1 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

closing

See also

dilation_rectangle1, erosion_rectangle1, opening_rectangle1, gen_rectangle1

Module

Foundation

dilation1 (Region, StructElement : RegionDilation : Iterations :)

Dilate a region.

dilation1 dilates the input regions with a structuring element. By applying dilation1 to a region, its boundary gets smoothed. In the process, the area of the region is enlarged. Furthermore, disconnected regions may be merged. Such regions, however, remain logically distinct region. The dilation is a set-theoretic region operation. It uses the union operation.

Let M (StructElement) and R (Region) be two regions, where M is the structuring element and R is the region to be processed. Furthermore, let m be a point in M. Then the displacement vector $\vec{v}_m = (dx, dy)$ is defined as the difference of the center of gravity of M and the vector $\vec{m}$. Let $t_{\vec{v}_m}(R)$ denote the translation of a region R by a vector $\vec{v}$. Then

$$\texttt{dilation1}(R, M) := \bigcup_{m \in M} t_{-\vec{v}_m}(R)$$

For each point m in M a translation of the region R is performed. The union of all these translations is the dilation of R with M. dilation1 is similar to the operator minkowski_add1, the difference is that in dilation1 the structuring element is mirrored at the origin. The position of StructElement is meaningless, since the displacement vectors are determined with respect to the center of gravity of M.

The parameter Iterations determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n. From the above definition it follows that an empty region is generated in case of an empty structuring element.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Attention

A dilation always results in enlarged regions. Closely spaced regions which may touch or overlap as a result of the dilation are still treated as two separate regions. If the desired behavior is to merge them into one region, the operator union1 has to be called first.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be dilated.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element.

▷ **RegionDilation** (output_object) ..... region(-array) ⇝ *object*
Dilated regions.

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \texttt{Iterations}) \ .$$

Result

`dilation1` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `add_channels`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_add1`, `minkowski_add2`, `dilation2`

See also

`erosion1`, `erosion2`, `opening`, `closing`

Module

Foundation

dilation2 (Region, StructElement : RegionDilation : Row, Column, Iterations :)

Dilate a region (using a reference point).

`dilation2` dilates the input regions with a structuring element (`StructElement`) having the reference point (`Row`,`Column`). `dilation2` has a similar effect as `dilation1`, the difference is that the reference point of the structuring element can be chosen arbitrarily. The parameter `Iterations` determines the number of iterations which are to be performed with the structuring element. The result of iteration $n-1$ is used as input for iteration n.

An empty region is generated in case of an empty structuring element.

Structuring elements (`StructElement`) can be generated with operators such as `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`, `draw_region`, `gen_region_polygon`, `gen_region_points`, etc.

Attention

A dilation always results in enlarged regions. Closely spaced regions which may touch or overlap as a result of the dilation are still treated as two separate regions. If the desired behavior is to merge them into one region, the operator `union1` has to be called first.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Regions to be dilated.
- ▷ **StructElement** (input_object) ..... region ⇝ *object*
  Structuring element.
- ▷ **RegionDilation** (output_object) ..... region(-array) ⇝ *object*
  Dilated regions.
- ▷ **Row** (input_control) ..... point.y ⇝ *integer*
  Row coordinate of the reference point.
  Default: 0
- ▷ **Column** (input_control) ..... point.x ⇝ *integer*
  Column coordinate of the reference point.
  Default: 0
- ▷ **Iterations** (input_control) ..... integer ⇝ *integer*
  Number of iterations.
  Default: 1
  Suggested values: `Iterations` ∈ {1, 2, 3, 4, 5, 7, 11, 17, 25, 32, 64, 128}
  Value range: 1 ≤ Iterations (lin)
  Minimum increment: 1
  Recommended increment: 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \texttt{Iterations}) \ .$$

Result

`dilation2` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `add_channels`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_add1`, `minkowski_add2`, `dilation1`

See also

`erosion1`, `erosion2`, `opening`, `closing`

Module

Foundation

dilation_circle (Region : RegionDilation : Radius :)

Dilate a region with a circular structuring element.

dilation_circle applies a Minkowski addition with a circular structuring element to the input regions Region. Because the circular mask is symmetrical, this is identical to a dilation. The size of the circle used as structuring element is determined by Radius.

The operator results in enlarged regions, smoothed region boundaries, and the holes smaller than the circular mask in the interior of the region are closed. It is useful to select only values like 3.5, 5.5, etc. for Radius in order to avoid a translation of a region, because integer radii result in the circle having a non-integer center of gravity which is rounded to the next integer.

Attention

dilation_circle is applied to each input region separately. If gaps between different regions are to be closed, union1 or union2 has to be called first.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Regions to be dilated.
- ▷ **RegionDilation** (output_object) ..... region(-array) ⇝ *object*
  Dilated regions.
- ▷ **Radius** (input_control) ..... real ⇝ *real* / integer
  Radius of the circular structuring element.
  Default: 3.5
  Suggested values: Radius ∈ {1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 9.5, 12.5, 15.5, 19.5, 25.5, 33.5, 45.5, 60.5, 110.5}
  Value range: 0.5 ≤ Radius ≤ 511.5 (lin)
  Minimum increment: 1.0
  Recommended increment: 1.0

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"

main()
{
  cout << "Reproduction of 'dilation_circle ()'" << endl;
  cout << "First = original image " << endl;
  cout << "Blue  = after dilation " << endl;
  cout << "Red   = before dilation " << endl;

  HByteImage img("monkey");
  HWindow    w;

  HRegion      circ   = HRegion::GenCircle (10, 10, 1.5);
  HRegionArray regs   = (img >= 128).Connection();
  HRegionArray minadd = regs.MinkowskiAdd1 (circ, 1);

                        img.Display (w);      w.Click ();
  w.SetColor ("blue");  minadd.Display (w);   w.Click ();
  w.SetColor ("red");   regs.Display (w);     w.Click ();

  return(0);
}
```

Complexity

Let $F1$ be the area of an input region. Then the runtime complexity for one region is:

$$O(2 \cdot \texttt{Radius} \cdot \sqrt{F1}) \ .$$

Result

`dilation_circle` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_add1`, `minkowski_add2`, `expand_region`, `dilation1`, `dilation2`, `dilation_rectangle1`

See also

`gen_circle`, `erosion_circle`, `closing_circle`, `opening_circle`

Module

Foundation

dilation_rectangle1 (Region : RegionDilation : Width, Height :)

Dilate a region with a rectangular structuring element.

`dilation_rectangle1` applies a dilation with a rectangular structuring element to the input regions `Region`. The size of the structuring rectangle is `Width` $\times$ `Height`. The operator results in enlarged regions, and the holes smaller than the rectangular mask in the interior of the regions are closed.

`dilation_rectangle1` is a very fast operation because the height of the rectangle enters only logarithmically into the runtime complexity, while the width does not enter at all. This leads to excellent runtime efficiency, even in the case of very large rectangles (edge length > 100).

Attention

`dilation_rectangle1` is applied to each input region separately. If gaps between different regions are to be closed, `union1` or `union2` has to be called first.

To enlarge a region by the same amount in all directions, `Width` and `Height` must be odd. If this is not the case, the region is dilated by a larger amount at the right or at the bottom, respectively, than at the left or at the top.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be dilated.

▷ **RegionDilation** (output_object) ..... region(-array) ⇝ *object*
Dilated regions.

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the structuring rectangle.
Default: 11
Suggested values: Width ∈ {1, 2, 3, 4, 5, 11, 15, 21, 31, 51, 71, 101, 151, 201}
Value range: 1 ≤ Width ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the structuring rectangle.
Default: 11
Suggested values: Height ∈ {1, 2, 3, 4, 5, 11, 15, 21, 31, 51, 71, 101, 151, 201}
Value range: 1 ≤ Height ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"

main()
{
  cout << "Reproduction of 'dilation_rectangle ()'" << endl;
  cout << "First = original image " << endl;
  cout << "Blue  = after dilation " << endl;
  cout << "Red   = after segmentation " << endl;

  HByteImage img("monkey");
  HWindow    w;

  HRegionArray regs   = (img >= 220).Connection();
  HRegionArray dilreg = regs.DilationRectangle1 (2, 4);

                        img.Display (w);      w.Click ();
  w.SetColor ("blue");  dilreg.Display (w);   w.Click ();
  w.SetColor ("red");   regs.Display (w);     w.Click ();

  return(0);
}
```

Complexity

Let $F1$ be the area of an input region and H be the height of the rectangle. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot ld(H)) \ .$$

Result

dilation_rectangle1 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

minkowski_add1, minkowski_add2, expand_region, dilation1, dilation2, dilation_circle

See also

gen_rectangle1, gen_region_polygon_filled

Module

Foundation

erosion1 (Region, StructElement : RegionErosion : Iterations :)

Erode a region.

erosion1 erodes the input regions with a structuring element. By applying erosion1 to a region, its boundary gets smoothed. In the process, the area of the region is reduced. Furthermore, connected regions may be split. Such regions, however, remain logically one region. The erosion is a set-theoretic region operation. It uses the intersection operation.

Let M (StructElement) and R (Region) be two regions, where M is the structuring element and R is the region to be processed. Furthermore, let m be a point in M. Then the displacement vector $\vec{v}_m = (dx, dy)$ is defined as the difference of the center of gravity of M and the vector $\vec{m}$. Let $t_{\vec{v}_m}(R)$ denote the translation of a region R by a vector $\vec{v}$. Then

$$\texttt{erosion1}(R, M) := \bigcap_{m \in M} t_{-\vec{v}_m}(R).$$

For each point m in M a translation of the region R is performed. The intersection of all these translations is the erosion of R with M. erosion1 is similar to the operator minkowski_sub1, the difference is that in erosion1 the structuring element is mirrored at the origin. The position of StructElement is meaningless, since the displacement vectors are determined with respect to the center of gravity of M.

The parameter Iterations determines the number of iterations which are to be performed with the structuring element. The result of iteration $n-1$ is used as input for iteration n. From the above definition it follows that the maximum region is generated in case of an empty structuring element.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be eroded.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element.

▷ **RegionErosion** (output_object) ..... region(-array) ⇝ *object*
Eroded regions.

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \texttt{Iterations}) \ .$$

Result

`erosion1` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_region_polygon_filled`

Possible Successors

`connection`, `reduce_domain`, `select_shape`, `area_center`

Alternatives

`minkowski_sub1`, `minkowski_sub2`, `erosion2`

See also

`transpose_region`

Module

Foundation

erosion2 (Region, StructElement : RegionErosion : Row, Column, Iterations :)

Erode a region (using a reference point).

`erosion2` erodes the input regions with a structuring element (`StructElement`) having the reference point (`Row`,`Column`). `erosion2` has a similar effect as `erosion1`, the difference is that the reference point of the structuring element can be chosen arbitrarily. The parameter `Iterations` determines the number of iterations which are to be performed with the structuring element. The result of iteration $n-1$ is used as input for iteration n.

A maximum region is generated in case of an empty structuring element.

Structuring elements (`StructElement`) can be generated with operators such as `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`, `draw_region`, `gen_region_polygon`, `gen_region_points`, etc.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be eroded.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element.

▷ **RegionErosion** (output_object) ..... region(-array) ⇝ *object*
Eroded regions.

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Row coordinate of the reference point.
Default: 0
Value range: 0 ≤ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column coordinate of the reference point.
Default: 0
Value range: 0 ≤ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \texttt{Iterations}) \ .$$

Result

erosion2 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_region_polygon_filled

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

minkowski_sub2, minkowski_sub1, erosion1

See also

transpose_region, gen_circle, gen_rectangle2, gen_region_polygon

Module

Foundation

erosion_circle (Region : RegionErosion : Radius :)

Erode a region with a circular structuring element.

erosion_circle applies a Minkowski subtraction with a circular structuring element to the input regions Region. Because the circular mask is symmetrical, this is identical to an erosion. The size of the circle used as structuring element is determined by Radius.

The operator results in reduced regions, smoothed region boundaries, and the regions smaller than the circular mask are eliminated. It is useful to select only values like 3.5, 5.5, etc. for Radius in order to avoid a translation of a region, because integer radii result in a circle having a non-integer center of gravity which is rounded to the next integer.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be eroded.

▷ **RegionErosion** (output_object) . . . . . region(-array) ⇝ *object*
Eroded regions.

▷ **Radius** (input_control) . . . . . real ⇝ *real* / integer
Radius of the circular structuring element.
Default: 3.5
Suggested values: Radius ∈ {1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 9.5, 12.5, 15.5, 19.5, 25.5, 33.5, 45.5, 60.5, 110.5}
Value range: 0.5 ≤ Radius ≤ 511.5 (lin)
Minimum increment: 1.0
Recommended increment: 1.0

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"

main()
{
  cout << "Simulation of 'erosion_circle ()'" << endl;
  cout << "First = original image " << endl;
  cout << "Red   = after segmentation " << endl;
  cout << "Blue  = after erosion " << endl;

  HByteImage img("monkey");
  HWindow    w;

  HRegion      circ   = HRegion::GenCircle (10, 10, 1.5);
  HRegionArray regs   = (img >= 128).Connection();
  HRegionArray minsub = regs.MinkowskiSub1 (circ, 1);

                         img.Display (w);      w.Click ();
  w.SetColor ("red");    regs.Display (w);     w.Click ();
  w.SetColor ("blue");   minsub.Display (w);   w.Click ();

  return(0);
}
```

Complexity

Let $F1$ be the area of an input region. Then the runtime complexity for one region is:

$$O(2 \cdot \texttt{Radius} \cdot \sqrt{F1}) \ .$$

Result

erosion_circle returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, watersheds, class_ndim_norm

Possible Successors

connection, reduce_domain, select_shape, area_center

Alternatives

minkowski_sub1

See also

gen_circle, dilation_circle, closing_circle, opening_circle

Module

Foundation

erosion_rectangle1 (Region : RegionErosion : Width, Height :)

Erode a region with a rectangular structuring element.

erosion_rectangle1 applies an erosion with a rectangular structuring element to the input regions Region. The size of the structuring rectangle is Width × Height. The operator results in reduced regions, and the areas smaller than the rectangular mask are eliminated.

erosion_rectangle1 is a very fast operation because the height of the rectangle enters only logarithmically into the runtime complexity, while the width does not enter at all. This leads to excellent runtime efficiency, even in the case of very large rectangles (edge length > 100).

Regions containing small connecting strips between large areas are separated only seemingly. They remain logically one region.

Attention

To reduce a region by the same amount in all directions, Width and Height must be odd. If this is not the case, the region is eroded by a smaller amount at the right or at the bottom, respectively, than at the left or at the top.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be eroded.

▷ **RegionErosion** (output_object) ..... region(-array) ⇝ *object*
Eroded regions.

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the structuring rectangle.
Default: 11
Suggested values: Width ∈ {1, 2, 3, 4, 5, 11, 15, 21, 31, 51, 71, 101, 151, 201}
Value range: $1 \leq$ Width ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the structuring rectangle.
Default: 11
Suggested values: Height ∈ {1, 2, 3, 4, 5, 11, 15, 21, 31, 51, 71, 101, 151, 201}
Value range: $1 \leq$ Height ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of an input region and H be the height of the rectangle. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot ld(H)) \ .$$

Result

`erosion_rectangle1` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`erosion1`, `minkowski_sub1`

See also

`gen_rectangle1`

Module

Foundation

hit_or_miss (Region, StructElement1, StructElement2 : RegionHitMiss : Row, Column :)

Hit-or-miss operation for regions.

`hit_or_miss` performs the hit-or-miss-transformation. First, an erosion with the structuring element `StructElement1` is done on the input region `Region`. Then an erosion with the structuring element `StructElement2` is performed on the complement of the input region. The intersection of the two resulting regions is the result `RegionHitMiss` of `hit_or_miss`.

The hit-or-miss-transformation selects precisely the points for which the conditions given by the structuring elements `StructElement1` and `StructElement2` are fulfilled. `StructElement1` determines the condition for the foreground pixels, while `StructElement2` determines the condition for the background pixels. In order to obtain sensible results, `StructElement1` and `StructElement2` must fit like key and lock. In any case, `StructElement1` and `StructElement2` must be disjunct. `Row` and `Column` determine the reference point of the structuring elements.

Structuring elements (`StructElement1`, `StructElement2`) can be generated by calling operators like `gen_region_points`, etc.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **StructElement1** (input_object) ..... region ⇝ *object*
Erosion mask for the input regions.

▷ **StructElement2** (input_object) . . . . . region ⇝ *object*
Erosion mask for the complements of the input regions.

▷ **RegionHitMiss** (output_object) . . . . . region(-array) ⇝ *object*
Result of the hit-or-miss operation.

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row coordinate of the reference point.
Default: 16
Suggested values: Row ∈ {0, 16, 32, 128, 256}
Value range: 0 ≤ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column coordinate of the reference point.
Default: 16
Suggested values: Column ∈ {0, 16, 32, 128, 256}
Value range: 0 ≤ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let F be the area of an input region, $F1$ the area of the structuring element 1, and $F2$ the area of the structuring element 2. Then the runtime complexity for one object is:

$$O\left(\sqrt{F} \cdot \left(\sqrt{F1} + \sqrt{F2}\right)\right) \quad .$$

Result

hit_or_miss returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm

Possible Successors

difference, reduce_domain, select_shape, area_center, connection

Alternatives

erosion2, dilation2

See also

gen_region_points, gen_region_polygon_filled

Module

Foundation

minkowski_add1 (Region,
StructElement : RegionMinkAdd : Iterations :)

Perform a Minkowski addition on a region.

minkowski_add1 dilates the input regions with a structuring element. By applying minkowski_add1 to a region, its boundary gets smoothed. In the process, the area of the region is enlarged. Furthermore, disconnected regions may be merged. Such regions, however, remain logically distinct region. The Minkowski addition is a set-theoretic region operation. It is based on translations and union operations.

Let M (StructElement) and R (Region) be two regions, where M is the structuring element and R is the region to be processed. Furthermore, let m be a point in M. Then the displacement vector $\vec{v}_m = (dx, dy)$ is defined as the difference of the center of gravity of M and the vector $\vec{m}$. Let $t_{\vec{v}_m}(R)$ denote the translation of a region R by a vector $\vec{v}$. Then

$$\texttt{minkowski\_add1}(R, M) := \bigcup_{m \in M} t_{\vec{v}_m}(R)$$

For each point m in M a translation of the region R is performed. The union of all these translations is the Minkowski addition of R with M. minkowski_add1 is similar to the operator dilation1, the difference is that in dilation1 the structuring element is mirrored at the origin. The position of StructElement is meaningless, since the displacement vectors are determined with respect to the center of gravity of M.

The parameter Iterations determines the number of iterations which are to be performed with the structuring element. The result of iteration $n-1$ is used as input for iteration n. From the above definition it follows that an empty region is generated in case of an empty structuring element.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Attention

A Minkowski addition always results in enlarged regions. Closely spaced regions which may touch or overlap as a result of the dilation are still treated as two separate regions. If the desired behavior is to merge them into one region, the operator union1 has to be called first.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be dilated.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element.

▷ **RegionMinkAdd** (output_object) ..... region(-array) ⇝ *object*
Dilated regions.

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \texttt{Iterations}) \ .$$

Result

minkowski_add1 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_region_polygon_filled

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

minkowski_add2, dilation1

See also

transpose_region, minkowski_sub1

Module

Foundation

minkowski_add2 (Region, StructElement : RegionMinkAdd : Row, Column, Iterations :)

Dilate a region (using a reference point).

minkowski_add2 computes the Minkowski addition of the input regions with a structuring element (StructElement) having the reference point (Row,Column). minkowski_add2 has a similar effect as minkowski_add1, the difference is that the reference point of the structuring element can be chosen arbitrarily. The parameter Iterations determines the number of iterations which are to be performed with the structuring element. The result of iteration $n-1$ is used as input for iteration n.

An empty region is generated in case of an empty structuring element.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Attention

A Minkowski addition always results in enlarged regions. Closely spaced regions which may touch or overlap as a result of the dilation are still treated as two separate regions. If the desired behavior is to merge them into one region, the operator union1 has to be called first.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be dilated.

▷ **StructElement** (input_object) . . . . . region ⇝ *object*
Structuring element.

▷ **RegionMinkAdd** (output_object) . . . . . region(-array) ⇝ *object*
Dilated regions.

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row coordinate of the reference point.
Value range: $1 \leq$ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column coordinate of the reference point.
Value range: $1 \leq$ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \texttt{Iterations}) \ .$$

Result

minkowski_add2 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_region_polygon_filled

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

minkowski_add1, dilation1

See also

transpose_region

Module

Foundation

minkowski_sub1 (Region, StructElement : RegionMinkSub : Iterations :)

Erode a region.

minkowski_sub1 computes the Minkowski subtraction of the input regions with a structuring element. By applying minkowski_sub1 to a region, its boundary gets smoothed. In the process, the area of the region is reduced. Furthermore, connected regions may be split. Such regions, however, remain logically one region. The Minkowski subtraction is a set-theoretic region operation. It uses the intersection operation.

Let M (StructElement) and R (Region) be two regions, where M is the structuring element and R is the region to be processed. Furthermore, let m be a point in M. Then the displacement vector $\vec{v}_m = (dx, dy)$ is defined as the difference of the center of gravity of M and the vector $\vec{m}$. Let $t_{\vec{v}_m}(R)$ denote the translation of a region R by a vector $\vec{v}$. Then

$$\texttt{minkowski\_sub1}(R, M) := \bigcap_{m \in M} t_{\vec{v}_m}(R)$$

For each point m in M a translation of the region R is performed. The intersection of all these translations is the Minkowski subtraction of R with M. minkowski_sub1 is similar to the operator erosion1, the difference is that in erosion1 the structuring element is mirrored at the origin. The position of StructElement is meaningless, since the displacement vectors are determined with respect to the center of gravity of M.

The parameter Iterations determines the number of iterations which are to be performed with the structuring element. The result of iteration $n-1$ is used as input for iteration n. From the above definition it follows that the maximum region is generated in case of an empty structuring element.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be eroded.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element.

▷ **RegionMinkSub** (output_object) ..... region(-array) ⇝ *object*
Eroded regions.

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \texttt{Iterations}) \ .$$

Result

minkowski_sub1 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_region_polygon_filled

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

minkowski_sub2, erosion1

See also

transpose_region

Module

Foundation

minkowski_sub2 (Region, StructElement : RegionMinkSub : Row, Column, Iterations :)

Erode a region (using a reference point).

minkowski_sub2 computes the Minkowski subtraction of the input regions with a structuring element (StructElement) having the reference point (Row,Column). minkowski_sub2 has a similar effect as minkowski_sub1, the difference is that the reference point of the structuring element can be chosen arbitrarily. The parameter Iterations determines the number of iterations which are to be performed with the structuring element. The result of iteration $n-1$ is used as input for iteration n.

A maximum region is generated in case of an empty structuring element.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be eroded.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element.

▷ **RegionMinkSub** (output_object) ..... region(-array) ⇝ *object*
Eroded regions.

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Row coordinate of the reference point.
Default: 0
Suggested values: Row ∈ {0, 10, 16, 32, 64, 100, 128}
Value range: 0 ≤ Row ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column coordinate of the reference point.
Default: 0
Suggested values: Column ∈ {0, 10, 16, 32, 64, 100, 128}
Value range: 0 ≤ Column ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Iterations** (input_control) ..... integer ⇝ *integer*
Number of iterations.
Default: 1
Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Iterations (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \text{Iterations}) \ .$$

Result

minkowski_sub2 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_region_polygon_filled

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

minkowski_sub1, erosion1, erosion2

See also

gen_circle, gen_rectangle2, gen_region_polygon

Module

Foundation

opening (Region, StructElement : RegionOpening : :)

Open a region.

An opening operation is defined as an erosion followed by a Minkowski addition. By applying opening to a region, larger structures remain mostly intact, while small structures like lines or points are eliminated. In contrast, a closing operation results in small gaps being retained or filled up (see closing).

opening serves to eliminate small regions (smaller than StructElement) and to smooth the boundaries of a region. The position of StructElement is meaningless, since an opening operation is invariant with respect to the choice of the reference point.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be opened.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element (position-invariant).

▷ **RegionOpening** (output_object) ..... region(-array) ⇝ *object*
Opened regions.

Example

```
* Large regions in an aerial picture (beech trees or meadows):
read_image(Image,'forest_road')
threshold(Image,Light,160,255)
gen_circle(StructElement,100,100,10)
* selecting the large regions
opening(Light,StructElement,Large)
* Selecting of edges with certain orientation:
read_image(Image,'fabrik')
sobel_amp(Image,Sobel,'sum_abs',3)
threshold(Sobel,Edges,10,255)
gen_rectangle2(StructElement,100,100,3.07819,20,1)
opening(Edges,StructElement,Direction)
```

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(2 \cdot \sqrt{F1} \cdot \sqrt{F2}) \ .$$

Result

`opening` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_add1`, `erosion1`, `opening_circle`

See also

`gen_circle`, `gen_rectangle2`, `gen_region_polygon`

Module

Foundation

opening_circle (Region : RegionOpening : Radius :)

Open a region with a circular structuring element.

`opening_circle` is defined as an erosion followed by a Minkowski addition with a circular structuring element (see example). `opening` serves to eliminate small regions (smaller than the circular structuring element) and to smooth the boundaries of a region.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be opened.

▷ **RegionOpening** (output_object) ..... region(-array) ⇝ *object*
Opened regions.

▷ **Radius** (input_control) ..... real ⇝ *real* / integer
Radius of the circular structuring element.
Default: 3.5
Suggested values: `Radius` ∈ {1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 9.5, 12.5, 15.5, 19.5, 25.5, 33.5, 45.5, 60.5, 110.5}
Value range: 0.5 ≤ `Radius` ≤ 511.5 (lin)
Minimum increment: 1.0
Recommended increment: 1.0

Example

```
* Large regions in an aerial picture (beech trees or meadows):
read_image (Image, 'forest_road')
threshold (Image, Region, 120, 255)
* Close the small gaps.
closing_circle (Region, RegionClosing, 3.5)
* Select the large regions.
opening_circle (RegionClosing, RegionOpening, 19.5)
```

Complexity

Let $F1$ be the area of the input region. Then the runtime complexity for one region is:

$$O(4 \cdot \sqrt{F1} \cdot \texttt{Radius}) \ .$$

Result

`opening_circle` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`opening`, `dilation1`, `minkowski_add1`, `gen_circle`

See also

`transpose_region`

Module

Foundation

opening_rectangle1 (Region : RegionOpening : Width, Height :)

Open a region with a rectangular structuring element.

`opening_rectangle1` behaves as `opening` with a rectangular structuring element on the input region `Region`. The size of the rectangular structuring element is determined by the parameters `Width` and `Height`. As is the case for all `opening` variants, larger structures are preserved, while small regions like lines or points are eliminated.

Similar to `erosion_rectangle1` and `dilation_rectangle1` `opening_rectangle1` is a very fast operation.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be opened.

▷ **RegionOpening** (output_object) . . . . . region(-array) ⇝ *object*
Opened regions.

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the structuring rectangle.
Default: 10
Suggested values: Width ∈ {1, 2, 3, 4, 5, 7, 9, 12, 15, 19, 25, 33, 45, 60, 110, 150, 200}
Value range: 1 ≤ Width ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the structuring rectangle.
Default: 10
Suggested values: Height ∈ {1, 2, 3, 4, 5, 7, 9, 12, 15, 19, 25, 33, 45, 60, 110, 150, 200}
Value range: 1 ≤ Height ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let $F1$ be the area of an input region and H be the height of the rectangle. Then the runtime complexity for one region is:

$$O(2 \cdot \sqrt{F1} \cdot ld(H)) \ .$$

Result

opening_rectangle1 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, watersheds, class_ndim_norm

Possible Successors

reduce_domain, select_shape, area_center, connection

Alternatives

opening, gen_rectangle1, dilation_rectangle1, erosion_rectangle1

Module

Foundation

pruning (Region : RegionPrune : Length :)

Prune the branches of a region.

pruning removes branches from a skeleton (Region) having a length less than Length. All other branches are preserved.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **RegionPrune** (output_object) ..... region(-array) ⇝ *object*
Result of the pruning operation.

▷ **Length** (input_control) ..... integer ⇝ *integer*
Length of the branches to be removed.
Default: 2
Suggested values: Length ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Value range: 1 ≤ Length ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let F be the area of the input region. Then the runtime complexity for one region is

$$O(\texttt{Length} \cdot 3 \cdot \sqrt{F}) \ .$$

Result

pruning returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: set_system('no_object_result',<RegionResult>)
- empty region: set_system('empty_region_result',<RegionResult>)

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

skeleton

Possible Successors

reduce_domain, select_shape, area_center, connection

See also

junctions_skeleton

Module

Foundation

top_hat (Region, StructElement : RegionTopHat : :)

Compute the top hat of regions.

top_hat computes the opening of Region with StructElement. The difference between the original region and the result of the opening is called the top hat. In contrast to opening, which splits regions under certain circumstances, top_hat computes the regions removed by such a splitting.

The position of StructElement is meaningless, since an opening operation is invariant with respect to the choice of the reference point.

Structuring elements (StructElement) can be generated with operators such as gen_circle, gen_rectangle1, gen_rectangle2, gen_ellipse, draw_region, gen_region_polygon, gen_region_points, etc.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be processed.

▷ **StructElement** (input_object) ..... region ⇝ *object*
Structuring element (position independent).

▷ **RegionTopHat** (output_object) ..... region(-array) ⇝ *object*
Result of the top hat operator.

Result

`top_hat` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`opening`, `difference`

See also

`bottom_hat`, `gray_tophat`, `opening`

Module

Foundation

Chapter 21

OCR

21.1 Convolutional Neural Networks

clear_ocr_class_cnn (: : OCRHandle :)

Clear an CNN-based OCR classifier.

clear_ocr_class_cnn clears the OCR classifier given by OCRHandle that was read with read_ocr_class_cnn and frees all memory required for the classifier. After calling clear_ocr_class_cnn, the classifier can no longer be used. The handle OCRHandle becomes invalid.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_cnn(-array) ⇝ *handle*
Handle of the OCR classifier.

Result

If OCRHandle is valid, the operator clear_ocr_class_cnn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

do_ocr_single_class_cnn, do_ocr_multi_class_cnn, do_ocr_word_cnn

See also

read_ocr_class_cnn

Module

OCR/OCV

deserialize_ocr_class_cnn (: : SerializedItemHandle : OCRHandle)

Deserialize a serialized CNN-based OCR classifier.

deserialize_ocr_class_cnn deserializes a CNN-based OCR classifier, that was serialized by serialize_ocr_class_cnn (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized OCR classifier is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created OCR classifier with the handle OCRHandle.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **OCRHandle** (output_control) . . . . . ocr_cnn ⇝ *handle*
Handle of the OCR classifier.

Result

If the parameters are valid, the operator deserialize_ocr_class_cnn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_ocr_class_cnn

Possible Successors

do_ocr_single_class_cnn, do_ocr_multi_class_cnn

Module

OCR/OCV

do_ocr_multi_class_cnn (Character, Image : : OCRHandle : Class, Confidence)

Classify multiple characters with an CNN-based OCR classifier.

do_ocr_multi_class_cnn computes the best class for each of the characters given by the regions Character and the gray values Image with the OCR classifier OCRHandle and returns the classes in Class and the corresponding confidences (probabilities) of the classes in Confidence.

In contrast to do_ocr_single_class_cnn, do_ocr_multi_class_cnn can classify multiple characters in one call, and therefore typically is faster than a loop that uses do_ocr_single_class_cnn to classify single characters. However, do_ocr_multi_class_cnn can only return the best class of each character. Because the confidences can be interpreted as probabilities, it is easier to check whether a character has been classified with too much uncertainty. This is usually not a disadvantage, except in cases where the classes overlap so much that in many cases the second best class must be examined to be able to decide the class of the character. In these cases, do_ocr_single_class_cnn should be used.

A string of the number '\032' (alternatively displayed as '\0x1A') in Class signifies that the region has been classified as rejection class.

Parameters

▷ **Character** (input_object) . . . . . region(-array) ⇝ *object*
Characters to be recognized.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Gray values of the characters.

▷ **OCRHandle** (input_control) . . . . . ocr_cnn ⇝ *handle*
Handle of the OCR classifier.

▷ **Class** (output_control) . . . . . string(-array) ⇝ *string*
Result of classifying the characters with the CNN.
Number of elements: Class == Character

▷ **Confidence** (output_control) . . . . . real(-array) ⇝ *real*
Confidence of the class of the characters.
Number of elements: Confidence == Character

Result

If the parameters are valid, the operator do_ocr_multi_class_cnn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

read_ocr_class_cnn

Alternatives

do_ocr_word_cnn, do_ocr_single_class_cnn

Module

OCR/OCV

do_ocr_single_class_cnn (Character, Image : : OCRHandle, Num : Class, Confidence)

Classify a single character with an CNN-based OCR classifier.

do_ocr_single_class_cnn computes the best classification for the character given by the region Character and the gray values Image with the OCR classifier OCRHandle and returns the best Num classes in Class and the corresponding confidences (probabilities) of the classes in Confidence. Because multiple classes may be returned by do_ocr_single_class_cnn, Character may only contain a single region (a single character). If multiple characters should be classified in a single call, do_ocr_multi_class_cnn must be used.

In most cases, do_ocr_multi_class_cnn should be preferred over do_ocr_single_class_cnn, since do_ocr_multi_class_cnn is typically faster than a loop with do_ocr_single_class_cnn. Furthermore, the resulting confidences can be interpreted as probabilities. They indicate the uncertainty of a character's classification result. The operator do_ocr_single_class_cnn has to be used if the nth-best class has to be examined explicitly for n greater than one.

A string of the number '\032' (alternatively displayed as '\0x1A') in Class signifies that the region has been classified as rejection class.

Parameters

▷ **Character** (input_object) . . . . . region ⇝ *object*
Character to be recognized.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object : byte / uint2*
Gray values of the character.

▷ **OCRHandle** (input_control) . . . . . ocr_cnn ⇝ *handle*
Handle of the OCR classifier.

▷ **Num** (input_control) . . . . . integer-array ⇝ *integer*
Number of best classes to determine.
Default: 1
Suggested values: Num ∈ {1, 2, 3, 4, 5}

▷ **Class** (output_control) . . . . . string(-array) ⇝ *string*
Result of classifying the character with the CNN.

▷ **Confidence** (output_control) . . . . . real(-array) ⇝ *real*
Confidence(s) of the class(es) of the character.

Example

```
read_image(Image, 'bottle2')
OffsetRow := 100
OffsetCol := 108
read_ocr_class_cnn('Universal_0-9_NoRej.occ', OCRHandle)
* Select each digit and use do_ocr_single_class_cnn to apply OCR
gen_rectangle1(ROI_Date, OffsetRow, OffsetCol, OffsetRow, OffsetCol)
for I := 1 to 6 by 1
  Offset := I % 2 * 10
  smallest_rectangle1(ROI_Date, Row1, Col1, Row2, Col2)
  gen_rectangle1(ROI_Date, OffsetRow, Offset + Col2, OffsetRow + 42, \
  Col2 + Offset + 31)
  do_ocr_single_class_cnn(ROI_Date, Image, OCRHandle, 1, Class, Confidence)
endfor
```

Result

If the parameters are valid, the operator `do_ocr_single_class_cnn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_ocr_class_cnn`

Alternatives

`do_ocr_multi_class_cnn`

Module

OCR/OCV

do_ocr_word_cnn (Character, Image : : OCRHandle, Expression, NumAlternatives, NumCorrections : Class, Confidence, Word, Score)

Classify a related group of characters with an CNN-based OCR classifier.

`do_ocr_word_cnn` works like `do_ocr_multi_class_cnn` insofar as it computes the best class for each of the characters given by the regions `Character` and the gray values `Image` with the OCR classifier `OCRHandle`, and returns the classes in `Class` and the corresponding confidences (probabilities) of the classes in `Confidence`.

In contrast to `do_ocr_multi_class_cnn`, `do_ocr_word_cnn` treats the group of characters as an entity which yields a `Word` by concatenating the class names for each character region. This allows to restrict the allowed classification results on a textual level by specifying an `Expression` describing the expected word.

The `Expression` may restrict the word to belong to a predefined lexicon created using `create_lexicon` or `import_lexicon`, or by specifying the name of the lexicon in angular brackets as in *'<mylexicon>'*. If the `Expression` is of any other form, it is interpreted as a regular expression with the same syntax as specified for `tuple_regexp_match`. Note that you will usually want to use an expression of the form '^...$' when using variable quantifiers like '*', to ensure that the entire word is used in the expression. Also note that in contrast to `tuple_regexp_match`, `do_ocr_word_cnn` does **not** support passing extra options in an expression tuple.

If the word derived from the best class for each character does not match the `Expression`, `do_ocr_word_cnn` attempts to correct it by considering the `NumAlternatives` best classes for each character. The alternatives used are identical to those returned by `do_ocr_single_class_cnn` for a single character. It does so by testing all possible corrections for which the classification result is changed for at most `NumCorrections` character

regions. Note that NumAlternatives and NumCorrections affect the complexity of the algorithm, so that in some cases internal restrictions are made. See the section 'Complexity' below for further information.

In case the Expression is a lexicon and the above procedure did not yield a result, the most similar word in the lexicon is returned as long as it requires less than NumCorrections edit operations for the correction (see suggest_lexicon).

The resulting word is graded by a Score between 0.0 (no correction found) and 1.0 (original word correct). The Score is lowered by adding a penalty according to the number of corrected characters and another (minor) penalty depending on how many classes with higher confidences have been ignored in order to match the Expression:

$$\texttt{Score} = 1.0 - \texttt{PenaltyCorrections} - \texttt{PenaltyAlternatives}$$

$$\begin{aligned} \texttt{PenaltyCorrections} &= \alpha * \texttt{num\_corr} \\ \texttt{PenaltyAlternatives} &= \alpha * \beta * \texttt{num\_alt} \end{aligned}$$

with num_corr being the actual number of applied corrections and num_alt the total number of discarded alternatives.

$$\begin{aligned} \alpha &= \frac{1}{\texttt{NumCorrections} + 1} \\ \beta &= \frac{1}{\texttt{NumCorrections} * (\texttt{NumAlternatives} - 1) + 2} \end{aligned}$$

Note that this is a combinatorial score which does **not** reflect the original Confidence of the best Class.

A string of the number '\032' (alternatively displayed as '\0x1A') in Class signifies that the region has been classified as rejection class.

Parameters

▷ **Character** (input_object) . . . . . region(-array) ⇝ *object*
Characters to be recognized.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Gray values of the characters.

▷ **OCRHandle** (input_control) . . . . . ocr_cnn ⇝ *handle*
Handle of the OCR classifier.

▷ **Expression** (input_control) . . . . . string ⇝ *string*
Expression describing the allowed word structure.

▷ **NumAlternatives** (input_control) . . . . . integer ⇝ *integer*
Number of classes per character considered for the internal word correction.
Default: 3
Suggested values: NumAlternatives ∈ {3, 4, 5}
Value range: 1 ≤ NumAlternatives

▷ **NumCorrections** (input_control) . . . . . integer ⇝ *integer*
Maximum number of corrected characters.
Default: 2
Suggested values: NumCorrections ∈ {1, 2, 3, 4, 5}
Value range: 0 ≤ NumCorrections

▷ **Class** (output_control) . . . . . string(-array) ⇝ *string*
Result of classifying the characters with the CNN.
Number of elements: Class == Character

▷ **Confidence** (output_control) . . . . . real(-array) ⇝ *real*
Confidence of the class of the characters.
Number of elements: Confidence == Character

▷ **Word** (output_control) . . . . . string ⇝ *string*
Word text after classification and correction.

▷ **Score** (output_control) . . . . . real ⇝ *real*
Measure of similarity between corrected word and uncorrected classification results.

Complexity

The complexity of checking all possible corrections is of magnitude $O((n * a)^{min(c,n)})$, where a is the number of alternatives, n is the number of character regions, and c is the number of allowed corrections. However, to guard against a near-infinite loop in case of large n, c is internally clipped to 5, 3, or 1 if $a * n >= 30$, 60, or 90, respectively.

Result

If the parameters are valid, the operator `do_ocr_word_cnn` returns the value 2 (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_ocr_class_cnn`

Alternatives

`do_ocr_multi_class_cnn`

Module

OCR/OCV

get_params_ocr_class_cnn (: : OCRHandle, GenParamName : GenParamValue)

Return the parameters of a CNN-based OCR classifier.

`get_params_ocr_class_cnn` returns the parameters `GenParamName` of an OCR classifier `OCRHandle` in `GenParamValue`. This is particularly useful if the classifier was read with `read_ocr_class_cnn`. The output of `get_params_ocr_class_cnn` can, for example, be used to check whether a character to be read is contained in the classifier.

Possible values for `GenParamName` are:

'characters': Returns a list of all trained symbols which may be recognized by the CNN-based OCR classifier. Note that if *'characters'* is specified then no other generic parameter is allowed. Otherwise an exception is thrown.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_cnn ⇝ *handle*
Handle of the OCR classifier.

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
A tuple of generic parameter names.
Default: 'characters'
List of values: `GenParamName` ∈ {'characters'}

▷ **GenParamValue** (output_control) . . . . . string(-array) ⇝ *integer* / string
A tuple of generic parameter values.

Result

If the parameters are valid, the operator `get_params_ocr_class_cnn` returns the value 2 (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_ocr_class_cnn

Possible Successors

do_ocr_single_class_cnn, do_ocr_multi_class_cnn

Module

OCR/OCV

query_params_ocr_class_cnn (: : OCRHandle : GenParamName)

Get the names of the parameters that can be used in get_params_ocr_class_cnn for a given CNN-based OCR classifier.

The operator query_params_ocr_class_cnn returns the parameter names that can be used to query all parameters for a given CNN-based OCR classifier. The resulting parameter names can be used as input parameter when calling get_params_ocr_class_cnn.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_cnn ⇝ *handle*
Handle of OCR classifier.

▷ **GenParamName** (output_control) ..... string(-array) ⇝ *string*
Names of the generic parameters.

Result

The operator query_params_ocr_class_cnn returns the value 2 (H_MSG_TRUE) if the given handle for the OCR classifier is correct. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_params_ocr_class_cnn

Module

OCR/OCV

read_ocr_class_cnn (: : FileName : OCRHandle)

Read an CNN-based OCR classifier from a file.

read_ocr_class_cnn reads a pretrained OCR classifier based on a convolutional neural network (CNN).

The CNN-based OCR classifier is read with read_ocr_class_cnn and subsequently used for classification with either do_ocr_single_class_cnn, do_ocr_multi_class_cnn, or do_ocr_word_cnn.

HALCON provides a number of pretrained OCR classifiers (see "Solution Guide I", chapter 'OCR', section 'Pretrained OCR Fonts'). These pretrained OCR classifiers make it possible to read a wide variety of different fonts without the need to train an OCR classifier. Note that the pretrained OCR classifiers were trained with symbols that are printed dark on light.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
Default: 'Universal_Rej.occ'
Suggested values: FileName ∈ {'Universal_NoRej.occ', 'Universal_Rej.occ', 'Universal_0-9_NoRej.occ', 'Universal_0-9_Rej.occ', 'Universal_0-9+_NoRej.occ', 'Universal_0-9+_Rej.occ', 'Universal_0-9A-Z_NoRej.occ', 'Universal_0-9A-Z_Rej.occ', 'Universal_0-9A-Z+_NoRej.occ', 'Universal_0-9A-Z+_Rej.occ', 'Universal_A-Z+_NoRej.occ', 'Universal_A-Z+_Rej.occ'}
File extension: .occ, .fnt

▷ **OCRHandle** (output_control) . . . . . ocr_cnn ⇝ *handle*
Handle of the OCR classifier.

Result

If the parameters are valid, the operator read_ocr_class_cnn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

do_ocr_single_class_cnn, do_ocr_multi_class_cnn, do_ocr_word_cnn

See also

get_params_ocr_class_cnn

Module

OCR/OCV

serialize_ocr_class_cnn (: : OCRHandle : SerializedItemHandle)

Serialize a CNN-based OCR classifier

serialize_ocr_class_cnn serializes a CNN-based OCR classifier (see fwrite_serialized_item for an introduction of the basic principle of serialization). The OCR classifier is defined by the handle OCRHandle. The serialized OCR classifier is returned by the handle SerializedItemHandle and can be deserialized by deserialize_ocr_class_cnn.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_cnn ⇝ *handle*
Handle of the OCR classifier.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_ocr_class_cnn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

clear_ocr_class_cnn, fwrite_serialized_item, send_serialized_item, deserialize_ocr_class_cnn

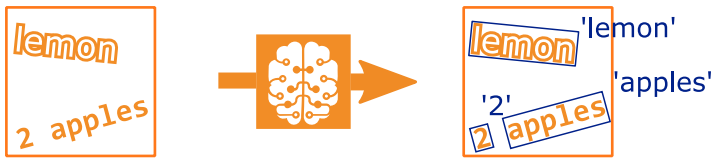
Module

OCR/OCV

21.2 Deep OCR

This chapter explains how to use deep-learning-based optical character recognition (Deep OCR).

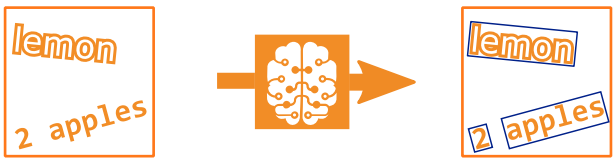
With Deep OCR we want to detect and/or recognize text in an image. Deep OCR detects and recognizes connected characters, which will be referred to as 'words' (in contrast to OCR methods which are used to read single characters).



A possible example for deep-learning-based optical character recognition: Words in an image are detected and recognized.

A Deep OCR model can contain two components, which are dedicated to two distinct tasks, the detection, thus the localization of words, and the recognition of words. By default, a model is created with both components, but the model can also be limited to either task.

HALCON already provides pretrained components, which are suited for a multitude of applications without additional training as the model is trained on a varied dataset and can therefore cope with many different fonts. Information on the available character set and model parameters can be retrieved using get_deep_ocr_param. To further adjust the reading to a specific task, it is possible to retrain the recognition or detection component separately on a given application domain using deep learning operators. Note that only one component can be retrained at a time.



The detection can be fine-tuned for an application by retraining the Deep OCR model with custom data.

The recognition can be fine-tuned for an application by retraining the Deep OCR model with custom data.

The general workflow as well as the retraining are described in the following paragraphs.

General Workflow for Deep OCR Inference

This paragraph describes the workflow how to localize and read words using a Deep OCR model. An application scenario can be seen in the HDevelop example deep_ocr_workflow.hdev.

Creation of the Deep OCR model Create a Deep OCR model containing either one or both of the two model components

- detection_model and

- recognition_model

using the operator create_deep_ocr.

To use a retrained model component instead of the provided one, adjust the created model by setting the retrained model component as *'recognition_model'* or *'detection_model'* using set_deep_ocr_param.

Inference Model parameters regarding, e.g., the used devices, image dimensions, or minimum scores can be set using set_deep_ocr_param.

The Deep OCR model is applied on your acquired images using apply_deep_ocr. The inference results depend on the used model components. See the operator reference of apply_deep_ocr for details regarding which dictionary entries are computed for each model composite.

The inference results can be retrieved from the dictionary DeepOCRResult. Some procedures are provided in order to visualize results and score maps:

- Show location and/or recognized word using dev_display_deep_ocr_results.
- Show location (and, if inferred, recognized word) on preprocessed image using dev_display_deep_ocr_results_preprocessed (if the model contains detection_model).
- Show score maps using dev_display_deep_ocr_score_maps (if the model contains detection_model).

Training and Evaluation of the Model Components

This paragraph describes the retraining and evaluation of the recognition or detection components of a Deep OCR model using custom data. See also the HDevelop examples deep_ocr_recognition_training_workflow.hdev or deep_ocr_detection_training_workflow.hdev for an application scenario.

Preprocess the data This part is about how to preprocess your data. See the section "Data" below for information on what data is to be provided at what stage of the Deep OCR workflow.

1. The information that is to be obtained from the images of your training dataset needs to be transferred. This is done by the procedure
   - read_dl_dataset_ocr_recognition for the recognition component of a Deep OCR model.
   - read_dl_dataset_ocr_detection for the detection component of a Deep OCR model.

   It creates a dictionary DLDataset which serves as a database and stores all necessary information about your data. For more information about datasets, see the chapter Deep Learning / Model.
2. Split the dataset represented by the dictionary DLDataset. This can be done using the procedure
   - split_dl_dataset.
3. The network imposes several requirements on the images. These requirements (for example the image size and gray value range) can be retrieved with
   - get_dl_model_param.

   For this you need to read the model first by using
   - read_dl_model.
4. Now you can preprocess your dataset. For this, you can use the procedure
   - preprocess_dl_dataset.

   To use this procedure, specify the preprocessing parameters as, e.g., the image size. Store all the parameter with their values in a dictionary DLPreprocessParam, for which you can use the procedure
   - create_dl_preprocess_param_from_model.

   We recommend to save this dictionary DLPreprocessParam in order to have access to the preprocessing parameter values later during the inference phase.

Training of the model This part explains how to train the recognition or detection component of a Deep OCR model.

1. Set the training parameters and store them in the dictionary `TrainParam`. This can be done using the procedure
   - `create_dl_train_param`.
2. Train the model. This can be done using the procedure
   - `train_dl_model`.

   The procedure expects:
   - the model handle `DLModelHandle`
   - the dictionary `DLDataset` containing the data information
   - the dictionary `TrainParam` containing the training parameters

Evaluation of the retrained model In this part, we evaluate the Deep OCR model.

1. Set the model parameters which may influence the evaluation.
2. The evaluation can be done conveniently using the procedure
   - `evaluate_dl_model`.

   This procedure expects a dictionary `GenParamEval` with the evaluation parameters.
3. The dictionary `EvaluationResult` holds the evaluation measures. To get a clue on how the retrained model performed against the pretrained model you can compare their evaluation values. To understand the different evaluation measures, see section "Evaluation Measures for Deep OCR Results".

After a successful evaluation the retrained model can be used for inference (see section "General Workflow for Deep OCR Inference" above).

Data

This section gives information on the data that needs to be provided in different stages of the Deep OCR workflow.

We distinguish between data used for training and evaluation, consisting of images with their information about the instances, and data for inference, which are bare images. How the data needs to be provided is explained in the according sections below.

As a basic concept, the model handles data over dictionaries, meaning it receives the input data over a dictionary `DLSample` and returns a dictionary `DLResult` and `DLTrainResult`, respectively. More information on the data handling can be found in the chapter Deep Learning / Model.

Data for training and evaluation The dataset consists of images and corresponding information. They have to be provided in a way the model can process them. Concerning the image requirements, find more information in the section "Images" below.

The training data is used to train and evaluate a network for your specific application. With the aid of this data the network can learn to detect or recognize text samples that resemble text that occurs during inference. The necessary information is given by providing the depicted word for each image.

How the data has to be formatted in HALCON for a DL model is explained in the chapter Deep Learning / Model. In short, a dictionary `DLDataset` serves as a database for the information needed by the training and evaluation procedures.

The data for `DLDataset` can be read using `read_dl_dataset_ocr_recognition` or `read_dl_dataset_ocr_detection` depending on which model type is used.

Dataset based on images with word labels In this case, images with words that are labeled with rotated bounding boxes need to be provided. You can label your data using the MVTec Deep Learning Tool, available from the MVTec website. The dataset must be built as follows:

- `'class_ids'`: class IDs
- `'class_names'`: class names (Needs to contain the class 'word'. All other classes are ignored.)

- 'image_dir': path to the image directory
- 'samples': tuple of dictionaries, one for each sample
  - 'image_file_name': name of the image file
  - 'image_id': image ID
  - 'bbox_col': bounding box column coordinate
  - 'bbox_row': bounding box row coordinate
  - 'bbox_phi': bounding box angle
  - 'bbox_length1': first half edge length of the bounding box
  - 'bbox_length2': second half edge length of the bounding box
  - 'label_custom_data': list of dictionaries containing custom label data for each bounding box
    * 'text' word to be read

Dataset based on word crop images (only recognition)

In this case, only images that are cropped to a single word each are included in the dataset. The dataset must be built as follows:

- 'image_dir': path to the image directory
- 'samples': tuple of dictionaries, one for each sample
  - 'image_file_name': name of the image file
  - 'image_id': image ID
  - 'word': word to be read in the image

The example program deep_ocr_prelabel_dataset.hdev can provide assistance by prelabeling your data.

Your training data should cover the full range of characters that might occur during inference. If a character is not or only very rarely contained in the training dataset the model might not properly learn to recognize that character. To keep track of the character distribution within the dataset the procedure gen_dl_dataset_ocr_recognition_statistics is provided, which generates statistics on how often every single character is contained in your dataset.

You also want enough training data to split it into three subsets, used for training, validation and testing the network. These subsets are preferably independent and identically distributed, see the section "Data" in the chapter Deep Learning.

Images The model poses requirements on the images, such as the dimensions, the gray value range, and the type. See the documentation of read_dl_model for the specific values of the trainable Deep OCR model. For a read model they can be queried with get_dl_model_param. In order to fulfill these requirements, you may have to preprocess your images. Standard preprocessing of an entire sample, including the image, is implemented in preprocess_dl_samples.

Requirements for images used for inference are described in apply_deep_ocr.

Model output The network output depends on the task:

training As output, the operator will return a dictionary DLTrainResult with the current value of the total loss as well as values for all other losses included in your model.

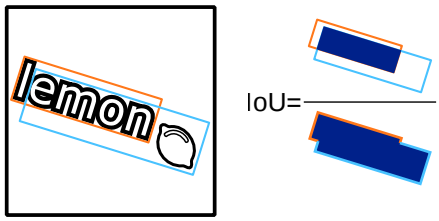
inference and evaluation As output, the network will return a dictionary DLResult for every sample. This dictionary will include the recognized word as well as the candidates and their confidences for every character of the word.

Evaluation Measures for Deep OCR Results

Deep OCR Detection The following evaluation measures are supported in HALCON. To compute these metrics for testing or validation, ground truth annotation is needed.

- Precision, Recall and F-score

  The performance of Deep OCR Detection is evaluated using precision and recall on word boxes. The evaluation uses the intersection over union (IoU) in order to compare ground truth and predicted word boxes. The default IoU threshold for a match is *0.5*, it can be increased or decreased if needed.



(1) (2)

Visual example of the IoU. (1) The input image with the ground truth bounding box (orange) and the predicted bounding box (light blue). (2) The IoU is the ratio between the area intersection and the union.

The precision is the proportion of true positives to all positives (true and false ones). Thus, it is a measure of how thrustworthy the detecor is.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

The recall is the proportion of the number of correctly detected words to all labeled words.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

To represent this with a single number, we compute the F-score, the harmonic mean of precision and recall.

$$\text{F-score} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

- Score of Angle Precision (SoAP)
  The SoAP value is a score for the precision of the inferred orientation angles. This score is determined by the angle differences between the inferred bounding boxes (I) and the corresponding ground truth annotations (GT):
  $\text{SoAP} = 1.0 - \frac{1}{\pi}\frac{1}{k} \cdot \sum_k \left|\varphi_k^{\text{I}} - \varphi_k^{\text{GT}}\right|$,
  where the index k runs over all inferred bounding boxes.

Deep OCR Recognition The accuracy for a Deep OCR Recognition task is given as the percentage of correctly read words (CR) to the ground truth words (GT) of a dataset. The accuracy is then defined as:

$$\text{Accuracy} = \frac{\text{CR}}{\text{GT}} * 100$$

apply_deep_ocr (Image : : DeepOcrHandle, Mode : DeepOcrResult)

Apply a Deep OCR model on a set of images for inference.

`apply_deep_ocr` applies the Deep OCR model given by `DeepOcrHandle` on the tuple of input images `Image`. The operator returns `DeepOcrResult`, a tuple with a result dictionary for every input image.

The operator `apply_deep_ocr` poses requirements on the input `Image`:

- Image type: byte.
- Number of channels: 1 or 3.

Further, the operator apply_deep_ocr will preprocess the given Image to match the model specifications. This means, the byte images will be normalized and converted to type real. Further, for Mode = *'auto'* or *'detection'* the input image Image is padded to the model input dimensions and, in case it has only one channel, converted into a three-channel image. For Mode = *'recognition'*, three-channel images are automatically converted to single-channel images.

The parameter Mode specifies a mode and with it, which component is executed. Supported values:

'auto' **(A):** Perform both parts, detection of the word and its recognition.

'detection' **(DET):** Perform only the detection part. Hence, the model will merely localize the word regions within the image.

'recognition' **(REC):** Perform only the recognition part. Hence, the model requires that the image contains solely a tight crop of the word.

Note, the model must have been created with the desired component, see create_deep_ocr.

The output dictionary DeepOcrResult can have entries according to the applied Mode (marked by its abbreviation):

image (A, DET, REC): Preprocessed image.

score_maps (A, DET): Scores given as image with four channels:

- Character score: Score for the character detection.
- Link score: Score for the connection of detected character centers to a connected word.
- Orientation 1: Sine component of the predicted word orientation.
- Orientation 2: Cosine component of the predicted word orientation.

words (A, DET): Dictionary containing the following entries. Thereby, the entries are tuples with a value for every found word.

- word (A): Recognized word.
- char_candidates (A): A dictionary with information for every character of every recognized word. The dictionary contains for every word a key/value pair: The index of the word as key and a tuple of dictionaries as value. Each of these character dictionaries contains the following key/value pairs:
  - 'candidate': Tuple with the best *'recognition_num_char_candidates'* candidates.
  - 'confidence': Softmax based confidence values of the best candidates. Note, these values are not calibrated and should be used with care. They can vary significantly for different models.
- word_image (A): Preprocessed image part containing the word.
- row (A, DET): Localized word: Center point, row coordinate.
- col (A, DET): Localized word: Center point, column coordinate.
- phi (A, DET): Localized word: Angle phi.
- length1 (A, DET): Localized word: Half length of edge 1.
- length2 (A, DET): Localized word: Half length of edge 2.
- line_index (A, DET): Line index of localized word if *'detection_sort_by_line'* set to *'true'*.

The word localization is given by the parameters of an oriented rectangle, see gen_rectangle2 for further information.

word_boxes_on_image (A, DET): Dictionary with the word localization on the coordinate system of the preprocessed images placed in image. The entries are tuples with a value for every found word.

- row (A, DET): Localized word: Center point, row coordinate.
- col (A, DET): Localized word: Center point, column coordinate.
- phi (A, DET): Localized word: Angle phi.
- length1 (A, DET): Localized word: Half length of edge 1.
- length2 (A, DET): Localized word: Half length of edge 2.

The word localization is given by the parameters of an oriented rectangle, see gen_rectangle2 for further information.

word_boxes_on_score_maps (A, DET): Dictionary with the word localization on the coordinate system of the score images placed in score_maps. The entries are the same as for word_boxes_on_image above.

word (REC): Recognized word.

char_candidates (REC): A tuple of dictionaries with information for every character in the recognized word.

Each of these character dictionaries contains the following key/value pairs:

- 'candidate': Tuple with the best *'recognition_num_char_candidates'* candidates.
- 'confidence': Softmax based confidence values of the best candidates. Note, these values are not calibrated and should be used with care. They can vary significantly for different models.

The recognition component can be retrained with custom data in order to further enhance the performance. See OCR / Deep OCR for more information.

Attention

System requirements: To run this operator on GPU (see get_deep_ocr_param), cuDNN and cuBLAS are required. For further details, please refer to the "Installation Guide", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods". Alternatively, this operator can also be run on CPU.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Input image.

▷ **DeepOcrHandle** (input_control) ..... deep_ocr ⇝ *handle*
Handle of the Deep OCR model.

▷ **Mode** (input_control) ..... string ⇝ *string*
Inference mode.
Default: []
List of values: Mode ∈ {'auto', 'recognition', 'detection'}

▷ **DeepOcrResult** (output_control) ..... dict(-array) ⇝ *handle*
Tuple of result dictionaries.

Result

If the parameters are valid, the operator apply_deep_ocr returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

get_deep_ocr_param, set_deep_ocr_param, create_deep_ocr

Module

OCR/OCV

create_deep_ocr (: : GenParamName, GenParamValue : DeepOcrHandle)

Create a Deep OCR model.

create_deep_ocr creates a Deep OCR model out of pretrained components and returns its handle in DeepOcrHandle. The handle states all parameters needed to run an inference.

A Deep OCR model usually consists of two components that are searched in the directory $HALCONROOT/dl/ by default:

- 'detection_model': detects words (connected characters) in the image.
- 'recognition_model': recognizes the word in the detected image part.

The following options may be set using GenParamName and according GenParamValue:

'mode': Specify a mode and with it, which component is executed. Supported values:

'detection': Perform only the detection part. Hence, the model will merely localize the word regions within the image.

'recognition': Perform only the recognition part. Hence, the model requires that the image contains solely a tight crop of the word.

'auto': Perform both parts, detection of the words and their recognition.

Default: *'auto'*.

The character set, which the model can recognize through its recognition component, can be retrieved using get_deep_ocr_param with *'recognition_alphabet'*.

To get an overview of the Deep OCR workflow see OCR / Deep OCR.

Parameters

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: GenParamName ∈ {'mode'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Value of the generic parameter.
Default: []
List of values: GenParamValue ∈ {'auto', 'recognition', 'detection'}

▷ **DeepOcrHandle** (output_control) . . . . . deep_ocr ⇝ *handle*
Handle of the Deep OCR model.

Result

If the parameters are valid, the operator create_deep_ocr returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

write_deep_ocr, apply_deep_ocr, get_deep_ocr_param, set_deep_ocr_param

Module

OCR/OCV

get_deep_ocr_param (: : DeepOcrHandle, GenParamName : GenParamValue)

Return the parameters of a Deep OCR model.

get_deep_ocr_param returns the parameter values GenParamValue of GenParamName for the Deep OCR model DeepOcrHandle.

Parameters can apply to the whole model or be specific for a given component. The following table gives an overview, which parameters can be set and which ones retrieved as well as for which model part they apply.

GenParamName	Model	Det. comp.	Recog. comp.
'device'	s	s	s
'detection_device'	–	s g	–
'detection_image_dimensions'	–	g	–
'detection_image_height'	–	s g	–
'detection_image_size'	–	s g	–
'detection_image_width'	–	s g	–
'detection_min_character_score'	–	s g	–
'detection_min_link_score'	–	s g	–
'detection_min_word_area'	–	s g	–
'detection_min_word_score'	–	s g	–
'detection_model'	–	s g	–
'detection_orientation'	–	s g	–
'detection_sort_by_line'	–	s g	–
'detection_tiling'	–	s g	–
'detection_tiling_overlap'	–	s g	–
'recognition_alphabet'	–	–	s g
'recognition_alphabet_internal'	–	–	s g
'recognition_alphabet_mapping'	–	–	s g
'recognition_batch_size'	–	–	s g
'recognition_device'	–	–	s g
'recognition_image_dimensions'	–	–	g
'recognition_image_height'	–	–	g
'recognition_image_width'	–	–	s g
'recognition_model'	–	–	s g
'recognition_num_char_candidates'	–	–	s g

Thereby, the symbols and abbreviations denote the following :

- 's': The parameter can be set using `set_deep_ocr_param`.
- 'g': The parameter can be retrieved using `get_deep_ocr_param`.
- '-': The parameter is not applicable.
- Model: Entire Deep OCR model with all its components.
- Det. comp.: Detection component.
- Recog. comp.: Recognition component.

Only parameters that do not change the model architecture can be set after the model has been optimized with `optimize_dl_model_for_inference`. A list of these parameters can be found at `optimize_dl_model_for_inference`. In the following the parameters are described, sorted according to the part of the model they apply to:

Entire model:

'device':
Handle of the device on which the model will be executed. To get a tuple of handles of all available potentially Deep-OCR-capable compute devices use `query_available_dl_devices`.
Note, the device can be reset for an individual component, in which case only the possibly remaining part of the model (e.g., the remaining component) will be executed on the device of this handle.
Default: Handle of the default device, thus the GPU with index *0* when querying a list using `get_system` with *'cuda_devices'*. If no device is available, this is an empty tuple.

Detection component:

'detection_device':
This parameter will set the device on which the detection component of the Deep OCR model is executed. For a further description, see *'device'*.
Default: The same value as for *'device'*.

'detection_image_dimensions':
Tuple containing the image dimensions (*'detection_image_width'*, *'detection_image_height'*, number of channels) the detection component will process.
The input image is scaled to *'detection_image_dimensions'* such that the original aspect ratio is preserved. The scaled image is padded with gray value *0* if necessary. Therefore, changing the *'detection_image_height'* or *'detection_image_width'* can influence the results.
Default: [*1024*, *1024*, *3*]

'detection_image_height':
Height of the images the detection component will process. This means, the network preserves the aspect ratio of the input image by scaling it to a maximum of this height before processing it. Thus this size can influence the results.
The model architecture requires that the height is a multiple of 32. If this is not the case, the height is rounded up to the nearest integer multiple of 32.
Suggested values: *768*, *1024*, *1280*
Default: *1024*

'detection_image_size':
Tuple containing the image size (*'detection_image_width'*, *'detection_image_height'*) the detection component will process.
Default: [*1024*, *1024*]

'detection_image_width':
Width of the images the detection component will process. This means, the network preserves the aspect ratio of the input image by scaling it to a maximum of this width before processing it. Thus this size can influence the results.
The model architecture requires that the width is a multiple of 32. If this is not the case, the width is rounded up to the nearest integer multiple of 32.
Suggested values: *768*, *1024*, *1280*
Default: *1024*

'detection_min_character_score':
The parameter *'detection_min_character_score'* specifies the lower threshold used for the character score map to estimate the dimensions of the characters. By adjusting the parameter, suggested instances can be split up or neighboring instances can be merged.
Value range: $\in [0, 1]$.
Default: *0.5*

'detection_min_link_score':
The parameter *'detection_min_link_score'* defines the minimum link score required between two localized characters to recognize these characters as coherent word.
Value range: $\in [0, 1]$.
Default: *0.3*

'detection_min_word_area':
The parameter *'detection_min_word_area'* defines the minimum size that a localized word must have in order to be suggested. This parameter can be used to filter suggestions that are too small.
Value range: ≥ 0.
Default: *10*.

'detection_min_word_score':
The parameter *'detection_min_word_score'* defines the minimum score a localized instance must contain to be suggested as valid word. With this parameter uncertain words can be filtered out.
Value range: $\in [0, 1]$.
Default: *0.7*

'detection_model':
The operator `get_deep_ocr_param` returns the handle of the Deep OCR model component for word detection. Using `set_deep_ocr_param` it is possible to either specify a handle, filename or special string. As a special string only *'default'* and *'compact'* are allowed. In case of *'default'* the default pretrained word detection component is loaded (i.e. *'pretrained_deep_ocr_detection.hdl'*). In case of *'compact'*, a more efficient word detection component

is loaded (i.e. *'pretrained_deep_ocr_detection_compact.hdl'*). If the given value is a string the model is loaded internally and the batch size is set to *1*.
Suggested values: *'default'*, *'compact'*, filename.
Default: *'default'*

'detection_orientation':
This parameter allows to set a predefined orientation angle for the word detection. To revert to default behavior using the internal orientation estimation, *'detection_orientation'* is set to *'auto'*.
Value range: $(-\pi, \pi]$.
Default: *'auto'*

'detection_sort_by_line':
The words are sorted line-wise based on the orientation of the localized word instances. If the parameter *'detection_sort_by_line'* is set to *'false'*, the results will not be sorted.
Default: *'true'*

'detection_tiling':
The input image is automatically split into overlapping tile images of size *'detection_image_size'*, which are processed separately by the detection component. This allows processing images that are much larger than the actual *'detection_image_size'* without having to zoom the input image. Thus, if *'detection_tiling'* = 'true', the input image will not be zoomed before processing it.
Default: *'false'*

'detection_tiling_overlap':
This parameter defines how much neighboring tiles overlap when input images are split (see *'detection_tiling'*). The overlap is given in pixels.
Value range: ≥ 0.
Default: *64*

Recognition component:

'recognition_alphabet':
The character set that can be recognized by the Deep OCR model.
It contains all characters that are not mapped to the Blank character of the internal alphabet (see parameters *'recognition_alphabet_mapping'* and *'recognition_alphabet_internal'*).
The alphabet can be changed or extended if needed. Changing the alphabet with this parameter will edit the internal alphabet and mapping in such a way that it tries to keep the length of the internal alphabet unchanged. After changing the alphabet, it is recommended to retrain the model on application specific data (see the HDevelop example `deep_ocr_recognition_training_workflow.hdev`). Previously unknown characters will need more training data.
Note, that if the length of the internal alphabet changes, the last model layers have to be randomly initialized and thus the output of the model will be random strings (see *'recognition_alphabet_internal'*). In that case it is required to retrain the model.

'recognition_alphabet_internal':
The full character set which the Deep OCR recognition component has been trained on.
The first character of the internal alphabet is a special character. In the pretrained model this character is specified as Blank (U+2800) and is not to be confused with a space. The Blank is never returned in a word output but can occur in the reported character candidates. It is required and cannot be omitted. If the internal alphabet is changed, the first character has to be the Blank. Furthermore, if *'recognition_alphabet'* is used to change the alphabet, the Blank symbol is added automatically to the character set.
The length of this tuple corresponds to the depth of the last convolution layer in the model. If the length changes, the last convolution layer and all layers after it have to be resized and potentially reinitialized randomly. After such a change, it is required to retrain the model (see HDevelop example `deep_ocr_recognition_training_workflow.hdev`).
It is recommend to use the parameter *'recognition_alphabet'* to change the alphabet, as it will automatically try to preserve the length of the internal alphabet.

'recognition_alphabet_mapping':
Tuple of integer indices.
It is a mapping that is applied by the model during the decoding step of each word. The mapping overwrites a character of *'recognition_alphabet_internal'* with the character at the specified index in *'recognition_alphabet_internal'*.

In the decoding step each prediction is mapped according to the index specified in this tuple. The tuple has to be of same length as the tuple *'recognition_alphabet_internal'*. Each integer index of the mapping has to be within *0* and |*'recognition_alphabet_internal'*|-1.
In some applications it can be helpful to map certain characters onto other characters. E.g. if only numeric words occur in an application it might be helpful to map the character "O" to the "0" character without the need to retrain the model.
If an entry contains a *0*, the corresponding character in *'recognition_alphabet_internal'* will not be decoded in the word.

'recognition_batch_size':
Number of images in a batch that is transferred to device memory and processed in parallel in the recognition component. For further details, please refer to the reference documentation of `apply_dl_model` with respect to the parameter *'batch_size'*. This parameter can be used to optimize the runtime of `apply_deep_ocr` on a given dl device. If the recognition component has to process multiple inputs (words), processing multiple inputs in parallel can result in a faster execution of `apply_deep_ocr`. Note, however, that a higher *'recognition_batch_size'* will require more device memory.
Default: *1*

'recognition_device':
This parameter will set the device on which the recognition component of the Deep OCR model is executed. For a further description, see *'device'*.
Default: The same value as for *'device'*.

'recognition_image_dimensions':
Tuple containing the image dimensions (*'recognition_image_width'*, *'recognition_image_height'*, number of channels) the recognition component will process.
This means, the network will first zoom the input image part to *'recognition_image_height'* while maintaining the aspect ratio of the input. If the width of the resulting image is smaller than *'recognition_image_width'*, the image part is padded with gray value *0* on the right. If it is larger, the image is zoomed to *'recognition_image_width'*.
Default: [*120*, *32*, *1*]

'recognition_image_height':
Height of the images the recognition component will process.
Default: *32*

'recognition_image_width':
Width of the images the recognition component will process.
Default: *120*

'recognition_model':
The operator `get_deep_ocr_param` returns the handle of the Deep OCR model component for word recognition.
Using `set_deep_ocr_param` it is possible to either specify a handle, filename or *'default'*. In case of *'default'* the pretrained word recognition component is loaded (i.e. *'pretrained_deep_ocr_recognition.hdl'*). If the given value is a string the model is loaded internally and the batch size is set to *1*.
Suggested values: *'default'*, filename.
Default: *'default'*

'recognition_num_char_candidates':
Controls the number of reported character candidates in the result of the operator `apply_deep_ocr`.
Default: *3*

Parameters

▷ **`DeepOcrHandle`** (input_control) ..... deep_ocr ⇝ *handle*
Handle of the Deep OCR model.

▷ **`GenParamName`** (input_control) ..... attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'recognition_model'
List of values: `GenParamName` ∈ {'detection_device', 'detection_image_dimensions', 'detection_image_height', 'detection_image_size', 'detection_image_width', 'detection_min_character_score', 'detection_min_link_score', 'detection_min_word_area', 'detection_min_word_score', 'detection_model', 'detection_orientation', 'detection_sort_by_line', 'detection_tiling', 'detection_tiling_overlap', 'recognition_alphabet', 'recognition_alphabet_internal',

'recognition_alphabet_mapping', 'recognition_batch_size', 'recognition_device', 'recognition_image_dimensions', 'recognition_image_height', 'recognition_image_width', 'recognition_model', 'recognition_num_char_candidates'}

▷ **GenParamValue** (output_control) . . . . . attribute.value(-array) ⇝ *integer* / handle / string / real
Value of the generic parameter.

Result

If the parameters are valid, the operator `get_deep_ocr_param` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_deep_ocr`, `set_deep_ocr_param`

Possible Successors

`set_deep_ocr_param`, `apply_deep_ocr`

See also

`set_deep_ocr_param`

Module

OCR/OCV

read_deep_ocr (: : FileName : DeepOcrHandle)

Read a Deep OCR model from a file.

The operator `read_deep_ocr` reads a Deep OCR model. Such models have to be in the HALCON format. As a result, the handle `DeepOcrHandle` is returned.

The model is loaded from the file `FileName`. This file is thereby searched in the directory `$HALCONROOT/dl/` as well as in the currently used directory. The default HALCON file extension for Deep OCR models is `'.hdo'`.

Please note that the values of runtime specific parameters are not written to file, see `write_deep_ocr`. As a consequence when reading a model these parameters are initialized with their default value, see `get_deep_ocr_param`.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Filename
File extension: `.hdo`

▷ **DeepOcrHandle** (output_control) . . . . . deep_ocr ⇝ *handle*
Handle of the Deep OCR model.

Result

If the parameters are valid, the operator `read_deep_ocr` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

create_deep_ocr

Possible Successors

set_deep_ocr_param, get_deep_ocr_param, apply_deep_ocr

Alternatives

create_deep_ocr

Module

OCR/OCV

set_deep_ocr_param (: : DeepOcrHandle, GenParamName, GenParamValue :)

Set the parameters of a Deep OCR model.

set_deep_ocr_param sets the parameters GenParamName of the Deep OCR model DeepOcrHandle to the values GenParamValue.

The possible parameters are listed and described in get_deep_ocr_param. An overview of the Deep OCR workflow is given in OCR / Deep OCR.

Attention

System requirements: To successfully set *'gpu'* parameters, cuDNN and cuBLAS are required, i.e., to set the parameter GenParamName *'device'* to a GPU. For further details, please refer to the "Installation Guide", paragraph "Requirements for Deep Learning and Deep-Learning-Based Methods".

Parameters

▷ **DeepOcrHandle** (input_control) . . . . . deep_ocr ⇝ *handle*
Handle of the Deep OCR model.

▷ **GenParamName** (input_control) . . . . . attribute.name ⇝ *string*
Name of the generic parameter.
Default: 'detection_image_width'
List of values: GenParamName ∈ {'device', 'detection_device', 'detection_image_height', 'detection_image_size', 'detection_image_width', 'detection_min_character_score', 'detection_min_link_score', 'detection_min_word_area', 'detection_min_word_score', 'detection_model', 'detection_orientation', 'detection_sort_by_line', 'detection_tiling', 'detection_tiling_overlap', 'recognition_alphabet', 'recognition_alphabet_internal', 'recognition_alphabet_mapping', 'recognition_batch_size', 'recognition_device', 'recognition_image_width', 'recognition_model', 'recognition_num_char_candidates'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *real* / string / integer / handle
Value of the generic parameter.
Default: 1024
Suggested values: GenParamValue ∈ {0, 10, 50, 100, 200, 512, 768, 1024, 1280, 0.3, 0.4, 0.5, 0.6, 0.7, 'true', 'false', 'auto', 'compact', 'default', 'pretrained_deep_ocr_detection.hdl', 'pretrained_deep_ocr_detection_compact.hdl', 'pretrained_deep_ocr_recognition.hdl'}

Result

If the parameters are valid, the operator set_deep_ocr_param returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_deep_ocr, get_deep_ocr_param

Possible Successors

get_deep_ocr_param, apply_deep_ocr

See also

get_deep_ocr_param

Module

OCR/OCV

write_deep_ocr (: : DeepOcrHandle, FileName :)

Write a Deep OCR model in a file.

write_deep_ocr writes the Deep OCR model DeepOcrHandle to the file given by FileName. Please note that the runtime specific parameters *'gpu'*, *'runtime'*, and *'runtime_init'* are not written.

The default HALCON file extension for Deep OCR models is '.hdo'.

The Deep OCR model can be read with read_deep_ocr.

Parameters

▷ **DeepOcrHandle** (input_control) ..... deep_ocr ⇝ *handle*
Handle of the Deep OCR model.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Filename
File extension: .hdo

Result

If the parameters are valid, the operator write_deep_ocr returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_deep_ocr, set_deep_ocr_param

Possible Successors

clear_handle

Module

OCR/OCV

21.3 K-Nearest Neighbors

clear_ocr_class_knn (: : OCRHandle :)

Clear an OCR classifier.

clear_ocr_class_knn clears the OCR classifier given by OCRHandle that was created with create_ocr_class_knn and frees all memory required for the classifier. After calling clear_ocr_class_knn, the classifier can no longer be used. The handle OCRHandle becomes invalid.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_knn ⇝ *handle*
Handle of the OCR classifier.

Result

If OCRHandle is valid, the operator clear_ocr_class_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

trainf_ocr_class_knn, read_ocr_class_knn

See also

create_class_knn

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

OCR/OCV

create_ocr_class_knn (: : WidthCharacter, HeightCharacter, Interpolation, Features, Characters, GenParamName, GenParamValue : OCRHandle)

Create an OCR classifier using a k-Nearest Neighbor (k-NN) classifier.

create_ocr_class_knn creates an OCR classifier that uses a k-Nearest Neighbor (k-NN). The handle of the k-NN classifier is returned in OCRHandle.

For a description on how a k-NN works, see create_class_knn.

The length of the feature vector of the k-NN is determined from the features that are used for the OCR, which are passed in Features. The features are described below. The number of classes is determined from the names of the characters which are passed in Characters.

Features can contain a tuple of several feature names. Each of these names results in one or more features to be calculated for the classifier. Some of the feature names compute gray value features (e.g., *'pixel_invar'*). Because a classifier requires a constant number of features (input variables), a character to be classified is transformed to a standard size, which is determined by WidthCharacter and HeightCharacter. The interpolation to be used for the transformation is determined by Interpolation. It has the same meaning as in affine_trans_image. The interpolation should be chosen such that no aliasing effects occur in the transformation. For most applications, Interpolation = *'constant'* should be used. It should be noted that the size of the transformed character is not chosen too large, because the generalization properties of the classifier may become bad for large sizes. In particular, large sizes will cause small segmentation errors to have a large influence on the computed features if gray value features are used. This happens because segmentation errors will change the smallest enclosing rectangle of the regions, which results in characters are zoomed differently than the characters in the training set. In most applications, sizes between 6×8 and 10×14 should be used.

The parameter Features can contain the following feature names for the classification of the characters.

'default': *'ratio'* and *'pixel_invar'* are selected.

'pixel': Gray values of the character (WidthCharacter $\times$ HeightCharacter features).

'pixel_invar': Gray values of the character with maximum scaling of the gray values (WidthCharacter $\times$ HeightCharacter features).

'pixel_binary': Region of the character as a binary image zoomed to a size of WidthCharacter $\times$ HeightCharacter (WidthCharacter $\times$ HeightCharacter features).

'gradient_8dir': Gradients are computed on the character image. The gradient directions are discretized into 8 directions. The amplitude image is decomposed into 8 channels according to these discretized directions. 25 samples on a 5×5 grid are extracted from each channel. These samples are used as features (200 features).

'projection_horizontal': Horizontal projection of the gray values (see gray_projections, HeightCharacter features).

'projection_horizontal_invar': Maximally scaled horizontal projection of the gray values (HeightCharacter features).

'projection_vertical': Vertical projection of the gray values (see gray_projections, WidthCharacter features).

'projection_vertical_invar': Maximally scaled vertical projection of the gray values (WidthCharacter features).

'ratio': Aspect ratio of the character (see height_width_ratio, 1 feature).

'anisometry': Anisometry of the character (see eccentricity, 1 feature).

'width': Width of the character before scaling the character to the standard size (not scale-invariant, see height_width_ratio, 1 feature).

'height': Height of the character before scaling the character to the standard size (not scale-invariant, see height_width_ratio, 1 feature).

'zoom_factor': Difference in size between the character and the values WidthCharacter and HeightCharacter (not scale-invariant, 1 feature).

'foreground': Fraction of pixels in the foreground (1 feature).

'foreground_grid_9': Fraction of pixels in the foreground in a 3×3 grid within the smallest enclosing rectangle of the character (9 features).

'foreground_grid_16': Fraction of pixels in the foreground in a 4×4 grid within the smallest enclosing rectangle of the character (16 features).

'compactness': Compactness of the character (see compactness, 1 feature).

'convexity': Convexity of the character (see convexity, 1 feature).

'moments_region_2nd_invar': Normalized 2nd moments of the character (see moments_region_2nd_invar, 3 features).

'moments_region_2nd_rel_invar': Normalized 2nd relative moments of the character (see moments_region_2nd_rel_invar, 2 features).

'moments_region_3rd_invar': Normalized 3rd moments of the character (see moments_region_3rd_invar, 4 features).

'moments_central': Normalized central moments of the character (see moments_region_central, 4 features).

'moments_gray_plane': Normalized gray value moments and the angle of the gray value plane (see moments_gray_plane, 4 features).

'phi': Sinus and cosinus of the orientation (angle) of the character (see elliptic_axis, 2 feature).

'num_connect': Number of connected components (see connect_and_holes, 1 feature).

'num_holes': Number of holes (see connect_and_holes, 1 feature).

'cooc': Values of the binary cooccurrence matrix (see gen_cooc_matrix, 8 features).

'num_runs': Number of runs in the region normalized by the height (1 feature).

'chord_histo': Frequency of the runs per row (not scale-invariant, HeightCharacter features).

After the classifier has been created, it is trained using trainf_ocr_class_knn. After this, the classifier can be saved using write_ocr_class_knn. Alternatively, the classifier can be used immediately after training to classify characters using do_ocr_single_class_knn or do_ocr_multi_class_knn.

A comparison of the k-NN and the support vector machine (SVM) (see create_ocr_class_svm) typically shows that SVMs are generally slower at training, especially for huge training sets, but achieve slightly better recognition rates than k-NNs. Please note that this guideline assumes optimal tuning of the parameters of the SVM.

Parameters

▷ **WidthCharacter** (input_control) . . . . . integer ⇝ *integer*
Width of the rectangle to which the gray values of the segmented character are zoomed.
Default: 8
Suggested values: WidthCharacter ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20}
Value range: 4 ≤ WidthCharacter ≤ 20

▷ **HeightCharacter** (input_control) . . . . . integer ⇝ *integer*
Height of the rectangle to which the gray values of the segmented character are zoomed.
Default: 10
Suggested values: HeightCharacter ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20}
Value range: 4 ≤ HeightCharacter ≤ 20

▷ **Interpolation** (input_control) . . . . . string ⇝ *string*
Interpolation mode for the zooming of the characters.
Default: 'constant'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'constant', 'weighted'}

▷ **Features** (input_control) . . . . . string(-array) ⇝ *string*
Features to be used for classification.
Default: 'default'
List of values: Features ∈ {'default', 'pixel', 'pixel_invar', 'pixel_binary', 'gradient_8dir', 'projection_horizontal', 'projection_horizontal_invar', 'projection_vertical', 'projection_vertical_invar', 'ratio', 'anisometry', 'width', 'height', 'zoom_factor', 'foreground', 'foreground_grid_9', 'foreground_grid_16', 'compactness', 'convexity', 'moments_region_2nd_invar', 'moments_region_2nd_rel_invar', 'moments_region_3rd_invar', 'moments_central', 'moments_gray_plane', 'phi', 'num_connect', 'num_holes', 'cooc', 'num_runs', 'chord_histo'}

▷ **Characters** (input_control) . . . . . string-array ⇝ *string*
All characters of the character set to be read.
Default: ['0','1','2','3','4','5','6','7','8','9']

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
This parameter is not yet supported.
Default: []
List of values: GenParamName ∈ {[]}

▷ **GenParamValue** (input_control) . . . . . number-array ⇝ *integer* / string
This parameter is not yet supported.
Default: []
List of values: GenParamValue ∈ {[]}

▷ **OCRHandle** (output_control) . . . . . ocr_knn ⇝ *handle*
Handle of the k-NN classifier.

Example

```
read_image (Image, 'letters')
* Segment the image.
binary_threshold(Image,&Region, 'otsu', 'dark', &UsedThreshold);
dilation_circle (Region, RegionDilation, 3.5)
connection (RegionDilation, ConnectedRegions)
intersection (ConnectedRegions, Region, RegionIntersection)
sort_region (RegionIntersection, Characters, 'character', 'true', 'row')
* Generate the training file.
count_obj (Characters, Number)
Classes := []
for J := 0 to 25 by 1
    Classes := [Classes,gen_tuple_const(20,chr(ord('a')+J))]
endfor
Classes := [Classes,gen_tuple_const(20,'.')]
write_ocr_trainf (Characters, Image, Classes, 'letters.trf')
* Generate and train the classifier.
read_ocr_trainf_names ('letters.trf', CharacterNames, CharacterCount)
create_ocr_class_knn (8, 10, 'constant', 'default', CharacterNames, \
```

```
                      [], [], OCRHandle)
trainf_ocr_class_knn (OCRHandle, 'letters.trf', [], [])
* Re-classify the characters in the image.
do_ocr_multi_class_knn (Characters, Image, OCRHandle, Class, Confidence)
```

Result

If the parameters are valid, the operator `create_ocr_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`trainf_ocr_class_knn`

Alternatives

`create_ocr_class_svm`

See also

`do_ocr_single_class_knn`, `do_ocr_multi_class_knn`, `clear_class_knn`, `create_class_knn`, `trainf_ocr_class_knn`, `classify_class_knn`

Module

OCR/OCV

deserialize_ocr_class_knn (: : SerializedItemHandle : OCRHandle)

Deserialize a serialized k-NN-based OCR classifier.

`deserialize_ocr_class_knn` deserializes a k-NN-based OCR classifier, that was serialized by `serialize_ocr_class_knn` (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The serialized OCR classifier is defined by the handle `SerializedItemHandle`. The deserialized values are stored in an automatically created OCR classifier with the handle `OCRHandle`.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **OCRHandle** (output_control) ..... ocr_knn ⇝ *handle*
Handle of the OCR classifier.

Result

If the parameters are valid, the operator `deserialize_ocr_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_ocr_class_knn`

Possible Successors

`classify_class_knn`

See also

create_ocr_class_knn, serialize_ocr_class_knn

References

Marius Muja, David G. Lowe: "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration"; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

OCR/OCV

do_ocr_multi_class_knn (Character, Image : : OCRHandle : Class, Confidence)

Classify multiple characters with an k-NN classifier.

do_ocr_multi_class_knn computes the best class for each of the characters given by the regions Character and the gray values Image with the k-NN classifier OCRHandle and returns the classes in Class and the corresponding confidence of the classes in Confidence. The confidences lie between 0.0 and 1.0. The larger the value, the more reliable is the classification of the single characters. In contrast to do_ocr_single_class_knn, do_ocr_multi_class_knn can classify multiple characters in one call, and therefore typically is faster than a loop that uses do_ocr_single_class_knn to classify single characters. However, do_ocr_multi_class_knn can only return the best class of each character.

Before calling do_ocr_multi_class_knn, the classifier must be trained with trainf_ocr_class_knn.

Parameters

- ▷ **Character** (input_object) ..... region(-array) ⇝ *object*
  Characters to be recognized.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
  Gray values of the characters.
- ▷ **OCRHandle** (input_control) ..... ocr_knn ⇝ *handle*
  Handle of the k-NN classifier.
- ▷ **Class** (output_control) ..... string(-array) ⇝ *string*
  Result of classifying the characters with the k-NN.
  Number of elements: Class == Character
- ▷ **Confidence** (output_control) ..... real(-array) ⇝ *real*
  Confidence of the class of the characters.
  Number of elements: Confidence == Character

Result

If the parameters are valid, the operator do_ocr_multi_class_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

trainf_ocr_class_knn, read_ocr_class_knn

Alternatives

do_ocr_single_class_knn

See also

create_ocr_class_knn, classify_class_knn

Module

OCR/OCV

do_ocr_single_class_knn (Character, Image : : OCRHandle, NumClasses, NumNeighbors : Class, Confidence)

Classify a single character with an OCR classifier.

do_ocr_single_class_knn uses the OCR-k-NN classifier OCRHandle to determine the best training samples for the features of the character given by the region Character and the gray values Image and returns their classes in Class and the corresponding confidences in Confidence. If the first NumNeighbors training vectors have different classes, maximally NumClasses classes are returned sorted by frequency and weighted distance. The confidences lie between 0.0 and 1.0. The larger the value, the more reliable is the classification. The confidence can be only computed robustly if NumNeighbors is large enough.

Because multiple classes may be returned by do_ocr_single_class_knn, Character may only contain a single region (a single character). If multiple characters should be classified in a single call, do_ocr_multi_class_knn must be used. Before calling do_ocr_single_class_knn, the classifier must be trained with trainf_ocr_class_knn or select_feature_set_trainf_knn. If all NumNeighbors neighbors are of the same class, only one class is returned.

Parameters

- ▷ **Character** (input_object) ..... region ⇝ *object*
  Character to be recognized.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
  Gray values of the character.
- ▷ **OCRHandle** (input_control) ..... ocr_knn ⇝ *handle*
  Handle of the k-NN classifier.
- ▷ **NumClasses** (input_control) ..... integer-array ⇝ *integer*
  Number of maximal classes to determine.
  Default: 1
  Suggested values: NumClasses ∈ {1, 2, 3, 4, 5}
- ▷ **NumNeighbors** (input_control) ..... integer-array ⇝ *integer*
  Number of neighbors to consider.
  Default: 1
  Suggested values: NumNeighbors ∈ {1, 2, 3, 4, 5}
- ▷ **Class** (output_control) ..... string(-array) ⇝ *string*
  Results of classifying the character with the k-NN.
- ▷ **Confidence** (output_control) ..... real(-array) ⇝ *real*
  Confidence(s) of the class(es) of the character.

Result

If the parameters are valid, the operator do_ocr_single_class_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_knn, read_ocr_class_knn

Alternatives

do_ocr_multi_class_knn

See also

create_ocr_class_knn, classify_class_knn

Module

OCR/OCV

do_ocr_word_knn (Character, Image : : OCRHandle, Expression, NumAlternatives, NumCorrections : Class, Confidence, Word, Score)

Classify a related group of characters with an OCR classifier.

do_ocr_word_knn works like do_ocr_multi_class_knn insofar as it computes the best class for each of the characters given by the regions Character and the gray values Image with the OCR classifier OCRHandle, and returns the classes in Class and the corresponding confidences of the classes in Confidence. The confidences lie between 0.0 and 1.0. The larger the value, the more reliable is the classification of the single characters.

In contrast to do_ocr_multi_class_knn, do_ocr_word_knn treats the group of characters as an entity which yields a Word by concatenating the class names for each character region. This allows to restrict the allowed classification results on a textual level by specifying an Expression describing the expected word.

The Expression may restrict the word to belong to a predefined lexicon created using create_lexicon or import_lexicon, by specifying the name of the lexicon in angular brackets as in *'<mylexicon>'*. If the Expression is of any other form, it is interpreted as a regular expression with the same syntax as specified for tuple_regexp_match. Note that you will usually want to use an expression of the form '^...$' when using variable quantifiers like '*', to ensure that the entire word is used in the expression. Also note that in contrast to tuple_regexp_match, do_ocr_word_knn does **not** support passing extra options in an expression tuple.

If the word derived from the best class for each character does not match the Expression, do_ocr_word_knn attempts to correct it by considering the NumAlternatives best classes for each character. The alternatives used are identical to those returned by do_ocr_single_class_knn for a single character. It does so by testing all possible corrections for which the classification result is changed for at most NumCorrections character regions. Note that NumAlternatives and NumCorrections affect the complexity of the algorithm, so that in some cases internal restrictions are made. See the section 'Complexity' below for further information.

In case the Expression is a lexicon and the above procedure did not yield a result, the most similar word in the lexicon is returned as long as it requires less than NumCorrections edit operations for the correction (see suggest_lexicon).

The resulting word is graded by a Score between 0.0 (no correction found) and 1.0 (original word correct). The Score is lowered by adding a penalty according to the number of corrected characters and another (minor) penalty depending on how many classes with higher confidences have been ignored in order to match the Expression:

$$\mathtt{Score} = 1.0 - \mathtt{PenaltyCorrections} - \mathtt{PenaltyAlternatives}$$

$$\begin{aligned} \mathtt{PenaltyCorrections} &= \alpha * \mathtt{num\_corr} \\ \mathtt{PenaltyAlternatives} &= \alpha * \beta * \mathtt{num\_alt} \end{aligned}$$

with num_corr being the actual number of applied corrections and num_alt the total number of discarded alternatives.

$$\begin{aligned} \alpha &= \frac{1}{\mathtt{NumCorrections} + 1} \\ \beta &= \frac{1}{\mathtt{NumCorrections} * (\mathtt{NumAlternatives} - 1) + 2} \end{aligned}$$

Note that this is a combinatorial score which does **not** reflect the original Confidence of the best Class.

Parameters

▷ **Character** (input_object) . . . . . region(-array) ⇝ *object*
Characters to be recognized.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Gray values of the characters.

▷ **OCRHandle** (input_control) . . . . . ocr_knn ⇝ *handle*
Handle of the OCR classifier.

▷ **Expression** (input_control) . . . . . string ⇝ *string*
Expression describing the allowed word structure.

▷ **NumAlternatives** (input_control) . . . . . integer ⇝ *integer*
Number of classes per character considered for the internal word correction.
Default: 3
Suggested values: NumAlternatives ∈ {3, 4, 5}
Value range: 1 ≤ NumAlternatives ≤ 5

▷ **NumCorrections** (input_control) . . . . . integer ⇝ *integer*
Maximum number of corrected characters.
Default: 2
Suggested values: NumCorrections ∈ {1, 2, 3, 4, 5}
Value range: 0 ≤ NumCorrections ≤ 5

▷ **Class** (output_control) . . . . . string(-array) ⇝ *string*
Result of classifying the characters with the k-NN.
Number of elements: Class == Character

▷ **Confidence** (output_control) . . . . . real(-array) ⇝ *real*
Confidence of the class of the characters.
Number of elements: Confidence == Character

▷ **Word** (output_control) . . . . . string ⇝ *string*
Word text after classification and correction.

▷ **Score** (output_control) . . . . . real ⇝ *real*
Measure of similarity between corrected word and uncorrected classification results.

Complexity

The complexity of checking all possible corrections is of magnitude $O((n * a)^{min(c,n)})$, where a is the number of alternatives, n is the number of character regions, and c is the number of allowed corrections. However, to guard against a near-infinite loop in case of large n, c is internally clipped to 5, 3, or 1 if $a * n >= 30$, 60, or 90, respectively.

Result

If the parameters are valid, the operator do_ocr_word_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_knn, read_ocr_class_knn

Alternatives

do_ocr_multi_class_knn

See also

create_ocr_class_knn, classify_class_knn

Module

OCR/OCV

get_features_ocr_class_knn (Character : : OCRHandle,
Transform : Features)

Compute the features of a character.

get_features_ocr_class_knn computes the features of the character given by Character with the k-NN classifier OCRHandle and returns them in Features. In contrast to do_ocr_single_class_knn and do_ocr_multi_class_knn, the character is passed as a single image object. Hence, before calling get_features_ocr_class_knn, reduce_domain must typically be called. The parameter Transform determines whether the feature transformation specified with Preprocessing in create_ocr_class_knn

for the classifier should be applied (Transform = *'true'*) or whether the untransformed features should be returned (Transform = *'false'*). get_features_ocr_class_knn can be used to inspect the features that are used for the classification.

Parameters

- ▷ **Character** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input character.
- ▷ **OCRHandle** (input_control) . . . . . ocr_knn ⇝ *handle*
  Handle of the k-NN classifier.
- ▷ **Transform** (input_control) . . . . . string ⇝ *string*
  Should the feature vector be transformed with the preprocessing?
  Default: 'true'
  List of values: Transform ∈ {'true', 'false'}
- ▷ **Features** (output_control) . . . . . real-array ⇝ *real*
  Feature vector of the character.

Result

If the parameters are valid, the operator get_features_ocr_class_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_knn

See also

create_ocr_class_knn

Module

OCR/OCV

get_params_ocr_class_knn (: : OCRHandle : WidthCharacter, HeightCharacter, Interpolation, Features, Characters, Preprocessing, NumTrees)

Return the parameters of an OCR classifier.

get_params_ocr_class_knn returns the parameters of an OCR classifier that were specified when the classifier was created with create_ocr_class_knn. This is particularly useful if the classifier was read with read_ocr_class_knn. The output of get_params_ocr_class_knn can, for example, be used to check whether a character to be read is contained in the classifier. For a description of the parameters, see create_ocr_class_knn and trainf_ocr_class_knn.

Parameters

- ▷ **OCRHandle** (input_control) . . . . . ocr_knn ⇝ *handle*
  Handle of the OCR classifier.
- ▷ **WidthCharacter** (output_control) . . . . . integer ⇝ *integer*
  Width of the rectangle to which the gray values of the segmented character are zoomed.
- ▷ **HeightCharacter** (output_control) . . . . . integer ⇝ *integer*
  Height of the rectangle to which the gray values of the segmented character are zoomed.
- ▷ **Interpolation** (output_control) . . . . . string ⇝ *string*
  Interpolation mode for the zooming of the characters.
- ▷ **Features** (output_control) . . . . . string(-array) ⇝ *string*
  Features to be used for classification.
- ▷ **Characters** (output_control) . . . . . string-array ⇝ *string*
  Characters of the character set to be read.

▷ **Preprocessing** (output_control) . . . . . string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
▷ **NumTrees** (output_control) . . . . . integer ⇝ *integer*
Number of different trees used during the classification.

Result

If the parameters are valid, the operator `get_params_ocr_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_ocr_class_mlp`, `read_ocr_class_mlp`

Possible Successors

`do_ocr_single_class_mlp`, `do_ocr_multi_class_mlp`

See also

`trainf_ocr_class_mlp`, `get_params_class_mlp`

Module

OCR/OCV

read_ocr_class_knn (: : FileName : OCRHandle)

Read an OCR classifier from a file.

`read_ocr_class_knn` reads an OCR classifier that has been stored with `write_ocr_class_knn`.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: `.onc`, `.okc`, `.fnt`
▷ **OCRHandle** (output_control) . . . . . ocr_knn ⇝ *handle*
Handle of the OCR classifier.

Result

If the parameters are valid, the operator `read_ocr_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`classify_class_knn`

See also

`create_ocr_class_knn`

References

Marius Muja, David G. Lowe: “Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration”; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

OCR/OCV

select_feature_set_trainf_knn (: : TrainingFile, FeatureList, SelectionMethod, Width, Height, GenParamName, GenParamValue : OCRHandle, FeatureSet, Score)

Select an optimal combination of features to classify OCR data.

select_feature_set_trainf_knn selects an optimal combination of features, to classify the data given in the training file TrainingFile with a k-Nearest Neighbor classifier, for details see create_ocr_class_knn.

Possible features are all OCR features listed and explained in create_ocr_class_knn. All candidates which should be tested can be specified in FeatureList. A subset of these features is returned as selected features in FeatureSet.

select_feature_set_trainf_knn is specialized on OCR problems and only supports the features in the list mentioned before. In order to use other features, please use the more general operator select_feature_set_knn.

The selection method SelectionMethod is either a greedy search *'greedy'* (iteratively add the feature with highest gain) or the dynamically oscillating search *'greedy_oscillating'* (add the feature with highest gain and test then if any of the already added features can be left out without great loss). The method *'greedy'* is generally preferable, since it is faster. Only in cases when a large training set is available the method *'greedy_oscillating'* might return better results.

The optimization criterion is the classification rate of a two-fold cross-validation of the training data. The best achieved value is returned in Score.

The k-NN classifier can be parametrized using the following values in GenParamName and GenParamValue:

'num_neighbors': The number of minimally evaluated nodes, increase this value for high dimensional data.
Suggested values: *1*, *2*, *5*, *10*
Default: *1*

'num_trees': Number of search trees in the k-NN classifier
Suggested values: *1*, *4*, *10*
Default: *4*

Attention

This operator may take considerable time, depending on the size of the data set in the training file, and the number of features.

Please note, that this operator should not be called, if only a small set of training data is available. Due to the risk of overfitting the operator select_feature_set_trainf_knn may deliver a classifier with a very high score. However, the classifier may perform poorly when tested.

Parameters

▷ **TrainingFile** (input_control) ..... filename.read(-array) ⇝ *string*
Names of the training files.
Default: ''
File extension: .trf, .otr

▷ **FeatureList** (input_control) ..... string(-array) ⇝ *string*
List of features that should be considered for selection.
Default: ['zoom_factor','ratio','width','height','foreground','foreground_grid_9','foreground_grid_16', 'anisometry','compactness','convexity','moments_region_2nd_invar','moments_region_2nd_rel_invar', 'moments_region_3rd_invar','moments_central','phi','num_connect','num_holes','projection_horizontal', 'projection_vertical','projection_horizontal_invar','projection_vertical_invar','chord_histo','num_runs', 'pixel','pixel_invar','pixel_binary','gradient_8dir','cooc','moments_gray_plane']
List of values: FeatureList ∈ {'default', 'zoom_factor', 'ratio', 'width', 'height', 'foreground', 'foreground_grid_9', 'foreground_grid_16', 'anisometry', 'compactness', 'convexity', 'moments_region_2nd_invar', 'moments_region_2nd_rel_invar', 'moments_region_3rd_invar', 'moments_central', 'phi', 'num_connect', 'num_holes', 'projection_horizontal', 'projection_vertical', 'projection_horizontal_invar', 'projection_vertical_invar', 'chord_histo', 'num_runs', 'pixel', 'pixel_invar', 'pixel_binary', 'gradient_8dir', 'cooc', 'moments_gray_plane'}

▷ **SelectionMethod** (input_control) ..... string ⇝ *string*
Method to perform the selection.
Default: 'greedy'
List of values: SelectionMethod ∈ {'greedy', 'greedy_oscillating'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the rectangle to which the gray values of the segmented character are zoomed.
Default: 15

▷ **Height** (input_control) ..... integer ⇝ *integer*
Height of the rectangle to which the gray values of the segmented character are zoomed.
Default: 16

▷ **GenParamName** (input_control) ..... string-array ⇝ *string*
Names of generic parameters to configure the selection process and the classifier.
Default: []
List of values: GenParamName ∈ {'num_neighbors'}

▷ **GenParamValue** (input_control) ..... number-array ⇝ *real* / integer / string
Values of generic parameters to configure the selection process and the classifier.
Default: []
Suggested values: GenParamValue ∈ {1, 2, 3}

▷ **OCRHandle** (output_control) ..... ocr_knn ⇝ *handle*
Trained OCR-k-NN classifier.

▷ **FeatureSet** (output_control) ..... string-array ⇝ *string*
Selected feature set, contains only entries from FeatureList.

▷ **Score** (output_control) ..... real-array ⇝ *real*
Achieved score using tow-fold cross-validation.

Result

If the parameters are valid, the operator select_feature_set_trainf_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Alternatives

select_feature_set_trainf_svm, select_feature_set_trainf_mlp

See also

select_feature_set_knn

Module

OCR/OCV

serialize_ocr_class_knn (: : OCRHandle : SerializedItemHandle)

Serialize a k-NN-based OCR classifier.

serialize_ocr_class_knn serializes a k-NN-based OCR classifier (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_ocr_class_knn is converted to a serialized item. The OCR classifier is defined by the handle OCRHandle. The serialized OCR classifier is returned by the handle SerializedItemHandle and can be deserialized by deserialize_ocr_class_knn.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_knn ⇝ *handle*
Handle of the OCR classifier.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_ocr_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`trainf_ocr_class_knn`, `read_ocr_class_knn`

Possible Successors

`fwrite_serialized_item`, `send_serialized_item`, `deserialize_ocr_class_knn`

See also

`create_class_knn`, `read_ocr_class_knn`, `deserialize_ocr_class_knn`

References

Marius Muja, David G. Lowe: “Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration”; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

OCR/OCV

trainf_ocr_class_knn (: : OCRHandle, TrainingFile, GenParamName, GenParamValue :)

Trains an k-NN classifier for an OCR task.

`trainf_ocr_class_knn` trains the k-NN classifier `OCRHandle` with the training characters stored in the OCR training files given by `TrainingFile`. The training files must have been created, e.g., using `write_ocr_trainf`, before calling `trainf_ocr_class_knn`. Please, note that training characters that have no corresponding class in the classifier `OCRHandle` are discarded.

The following options may be set using `GenParamName` and `GenParamValue`, respectively:

'num_trees' Sets the number of search trees. A higher number of trees improves the accuracy of the search, but also increases the runtime.
Default: *4*.

'normalization' Activates the data normalization, if set to *'true'*. This will change the stored training data permanently. Therefore, adding data after the training is not possible.
List of values: *'true'*, *'false'*.
Default: *'false'*.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **TrainingFile** (input_control) . . . . . filename.read(-array) ⇝ *string*
Names of the training files.
Default: 'ocr.trf'
File extension: `.trf`, `.otr`

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of the generic parameters that can be adjusted for the k-NN classifier creation.
Default: []
List of values: GenParamName ∈ {'num_trees', 'normalization'}

▷ **GenParamValue** (input_control) . . . . . number-array ⇝ *integer* / string / real
Values of the generic parameters that can be adjusted for the k-NN classifier creation.
Default: []
Suggested values: GenParamValue ∈ {4, 5, 'false', 'true'}

Example

```
* Train an OCR classifier
read_ocr_trainf_names ('ocr.trf', CharacterNames, CharacterCount)
create_ocr_class_knn (8, 10, 'constant', 'default', CharacterNames, [],\
                      [], OCRHandle)
trainf_ocr_class_knn (OCRHandle, 'ocr.trf', [], [])
write_ocr_class_knn (OCRHandle, 'ocr.omc')
```

Result

If the parameters are valid, the operator trainf_ocr_class_knn returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_ocr_class_knn, write_ocr_trainf, append_ocr_trainf, write_ocr_trainf_image

Possible Successors

do_ocr_single_class_knn, do_ocr_multi_class_knn

Alternatives

read_ocr_class_knn

See also

train_class_knn

Module

OCR/OCV

write_ocr_class_knn (: : OCRHandle, FileName :)

Write a k-NN classifier for an OCR task to a file.

write_ocr_class_knn writes the k-NN classifier for an OCR task with the OCRHandle to the file given by FileName. If a file extension is not specified in FileName the default extension '.onc' is appended to FileName. write_ocr_class_knn is typically called after the classifier has been trained with trainf_ocr_class_knn. The classifier can be read with read_ocr_class_knn.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_knn ⇝ *handle*
Handle of the k-NN classifier for an OCR task.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: .onc

Result

If the parameters are valid, the operator `write_ocr_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`trainf_ocr_class_knn`, `read_ocr_class_knn`

See also

`create_class_knn`, `read_ocr_class_knn`

References

Marius Muja, David G. Lowe: “Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration”; International Conference on Computer Vision Theory and Applications (VISAPP 09); 2009.

Module

OCR/OCV

21.4 Lexica

clear_lexicon (: : LexiconHandle :)

Clear a lexicon.

`clear_lexicon` clears a lexicon and releases its resources.

Parameters

▷ **LexiconHandle** (input_control) . . . . . lexicon ⇝ *handle*
Handle of the lexicon.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- LexiconHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

`create_lexicon`

Module

OCR/OCV

create_lexicon (: : Name, Words : LexiconHandle)

Create a lexicon from a tuple of words.

create_lexicon creates a new lexicon based on a tuple of Words. By specifying a unique textual Name, you can later refer to the lexicon from syntax expressions like those used, e.g., by do_ocr_word_mlp.

Note that lexicon support in HALCON is currently not aimed at natural languages. Rather, it is intended as a post-processing step in OCR applications that only need to distinguish between a limited set of not more than a few thousand valid words, e.g., country or product names. MVTec itself does not provide any lexica.

Parameters

▷ **Name** (input_control) . . . . . string ⇝ *string*
Unique name for the new lexicon.
Default: 'lex1'

▷ **Words** (input_control) . . . . . string-array ⇝ *string*
Word list for the new lexicon.
Default: ['word1','word2','word3']

▷ **LexiconHandle** (output_control) . . . . . lexicon ⇝ *handle*
Handle of the lexicon.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

do_ocr_word_mlp, do_ocr_word_svm

Alternatives

import_lexicon

See also

lookup_lexicon, suggest_lexicon

Module

OCR/OCV

import_lexicon (: : Name, FileName : LexiconHandle)

Create a lexicon from a text file.

import_lexicon creates a new lexicon based on a list of words in the file specified by FileName. The format of the file is a simple text file with one word per line. By specifying a unique textual Name, you can later refer to the lexicon from syntax expressions like those used, e.g., by do_ocr_word_mlp.

Note that lexicon support in HALCON is currently not aimed at natural languages. Rather, it is intended as a post-processing step in OCR applications that only need to distinguish between a limited set of not more than a few thousand valid words, e.g., country or product names. When the lexicon file contains entries with special, non-ASCII characters, it is expected to be encoded in UTF-8. However, old lexicon files which use the local 8-bit encoding, can still be loaded as long as they contain at least one byte sequence, which cannot be misinterpreted as UTF-8 character. MVTec itself does not provide any lexica.

Parameters

▷ **Name** (input_control) . . . . . string ⇝ *string*
Unique name for the new lexicon.
Default: 'lex1'

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Name of a text file containing words for the new lexicon.
Default: 'words.txt'
File extension: `.txt`

▷ **LexiconHandle** (output_control) . . . . . lexicon ⇝ *handle*
Handle of the lexicon.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`do_ocr_word_mlp`, `do_ocr_word_svm`

Alternatives

`create_lexicon`

See also

`lookup_lexicon`, `suggest_lexicon`

Module

OCR/OCV

inspect_lexicon (: : LexiconHandle : Words)

Query all words from a lexicon.

`inspect_lexicon` returns a tuple of all words in the lexicon in the parameter `Words`.

Parameters

▷ **LexiconHandle** (input_control) . . . . . lexicon ⇝ *handle*
Handle of the lexicon.

▷ **Words** (output_control) . . . . . string(-array) ⇝ *string*
List of all words.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`lookup_lexicon`

See also

`create_lexicon`

Module

OCR/OCV

lookup_lexicon (: : LexiconHandle, Word : Found)

Check if a word is contained in a lexicon.

lookup_lexicon checks whether Word is contained in the lexicon LexiconHandle, and returns 1 in Found if the word is found, otherwise 0.

Parameters

▷ **LexiconHandle** (input_control) . . . . . lexicon ⇝ *handle*
Handle of the lexicon.

▷ **Word** (input_control) . . . . . string ⇝ *string*
Word to be looked up.
Default: 'word'

▷ **Found** (output_control) . . . . . integer ⇝ *integer*
Result of the search.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

suggest_lexicon

See also

create_lexicon

Module

OCR/OCV

suggest_lexicon (: : LexiconHandle, Word : Suggestion, NumCorrections)

Find a similar word in a lexicon.

suggest_lexicon compares Word to all words in the lexicon and calculates the minimum number of edit operations NumCorrections required to transform Word into a word from the lexicon. Valid edit operations are the insertion, deletion and replacement of characters. The most similar word found in the lexicon is returned in Suggestion. If there are multiple words with the same minimum number of corrections, only the first of those words is returned.

Parameters

▷ **LexiconHandle** (input_control) . . . . . lexicon ⇝ *handle*
Handle of the lexicon.

▷ **Word** (input_control) . . . . . string ⇝ *string*
Word to be looked up.
Default: 'word'

▷ **Suggestion** (output_control) . . . . . string ⇝ *string*
Most similar word found in the lexicon.

▷ **NumCorrections** (output_control) . . . . . integer ⇝ *integer*
Difference between the words in edit operations.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

lookup_lexicon, tuple_str_distance

See also

create_lexicon

References

Vladimir I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Doklady Akademii Nauk SSSR, 163(4):845-848, 1965 (Russian). English translation in Soviet Physics Doklady, 10(8):707-710, 1966.

Module

OCR/OCV

21.5 Neural Nets

clear_ocr_class_mlp (: : OCRHandle :)

Clear an OCR classifier.

clear_ocr_class_mlp clears the OCR classifier given by OCRHandle that was created with create_ocr_class_mlp and frees all memory required for the classifier. After calling clear_ocr_class_mlp, the classifier can no longer be used. The handle OCRHandle becomes invalid.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_mlp(-array) ⇝ *handle*
Handle of the OCR classifier.

Result

If OCRHandle is valid, the operator clear_ocr_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

do_ocr_single_class_mlp, do_ocr_multi_class_mlp

See also

create_ocr_class_mlp, read_ocr_class_mlp, write_ocr_class_mlp, trainf_ocr_class_mlp

Module

OCR/OCV

create_ocr_class_mlp (: : WidthCharacter, HeightCharacter, Interpolation, Features, Characters, NumHidden, Preprocessing, NumComponents, RandSeed : OCRHandle)

Create an OCR classifier using a multilayer perceptron.

create_ocr_class_mlp creates an OCR classifier that uses a multilayer perceptron (MLP). The handle of the OCR classifier is returned in OCRHandle.

For a description on how an MLP works, see create_class_mlp. create_ocr_class_mlp creates an MLP with OutputFunction = *'softmax'*. The length of the feature vector of the MLP (NumInput in create_class_mlp) is determined from the features that are used for the OCR, which are passed in Features. The features are described below. The number of units in the hidden layer is determined by

NumHidden. The number of output variables of the MLP (NumOutput in create_class_mlp) is determined from the names of the characters to be used in the OCR, which are passed in Characters. As described with create_class_mlp, the parameters Preprocessing and NumComponents can be used to specify a preprocessing of the data (i.e., the feature vectors). The OCR already approximately normalizes the features. Hence, Preprocessing can typically be set to *'none'*. The parameter RandSeed has the same meaning as in create_class_mlp. Furthermore, like for general MLP classifiers (see create_class_mlp and set_regularization_params_class_mlp), it may be desirable to regularize OCR classifiers. This can be achieved by calling set_regularization_params_ocr_class_mlp before training the OCR classifier. In addition, like for general MLP classifiers (see create_class_mlp and set_rejection_params_class_mlp), it might be desirable to equip the OCR classifiers with the capability to reject unknown characters. The rejection class is by convention an additional symbol chr(26) that must be provided in Characters. The parameters of the rejection class can be set by calling set_rejection_params_ocr_class_mlp before training the OCR classifier.

The features to be used for the classification are determined by Features. Features can contain a tuple of several feature names. Each of these feature names results in one or more features to be calculated for the classifier. Some of the feature names compute gray value features (e.g., *'pixel_invar'*). Because a classifier requires a constant number of features (input variables), a character to be classified is transformed to a standard size, which is determined by WidthCharacter and HeightCharacter. The interpolation to be used for the transformation is determined by Interpolation. It has the same meaning as in affine_trans_image. The interpolation should be chosen such that no aliasing effects occur in the transformation. For most applications, Interpolation = *'constant'* should be used. It should be noted that the size of the transformed character is not chosen too large, because the generalization properties of the classifier may become bad for large sizes. In particular, large sizes will lead to the fact that small segmentation errors will have a large influence on the computed features if gray value features are used. This happens because segmentation errors will change the smallest enclosing rectangle of the regions, which leads to the fact that the character is zoomed differently than the characters in the training set. In most applications, sizes between 6×8 and 10×14 should be used.

The parameter Features can contain the following feature names for the classification of the characters.

'default' *'ratio'* and *'pixel_invar'* are selected.

'pixel' Gray values of the character (WidthCharacter $\times$ HeightCharacter features).

'pixel_invar' Gray values of the character with maximum scaling of the gray values (WidthCharacter $\times$ HeightCharacter features).

'pixel_binary' Region of the character as a binary image zoomed to a size of WidthCharacter $\times$ HeightCharacter (WidthCharacter $\times$ HeightCharacter features).

'gradient_8dir' Gradients are computed on the character image. The gradient directions are discretized into 8 directions. The amplitude image is decomposed into 8 channels according to these discretized directions. 25 samples on a 5×5 grid are extracted from each channel. These samples are used as features (200 features).

'projection_horizontal' Horizontal projection of the gray values (see gray_projections, HeightCharacter features).

'projection_horizontal_invar' Maximally scaled horizontal projection of the gray values (HeightCharacter features).

'projection_vertical' Vertical projection of the gray values (see gray_projections, WidthCharacter features).

'projection_vertical_invar' Maximally scaled vertical projection of the gray values (WidthCharacter features).

'ratio' Aspect ratio of the character (see height_width_ratio, 1 feature).

'anisometry' Anisometry of the character (see eccentricity, 1 feature).

'width' Width of the character before scaling the character to the standard size (not scale-invariant, see height_width_ratio, 1 feature).

'height' Height of the character before scaling the character to the standard size (not scale-invariant, see height_width_ratio, 1 feature).

'zoom_factor' Difference in size between the character and the values of WidthCharacter and HeightCharacter (not scale-invariant, 1 feature).

'foreground' Fraction of pixels in the foreground (1 feature).

'foreground_grid_9' Fraction of pixels in the foreground in a 3×3 grid within the smallest enclosing rectangle of the character (9 features).

'foreground_grid_16' Fraction of pixels in the foreground in a 4×4 grid within the smallest enclosing rectangle of the character (16 features).

'compactness' Compactness of the character (see `compactness`, 1 feature).

'convexity' Convexity of the character (see `convexity`, 1 feature).

'moments_region_2nd_invar' Normalized 2nd moments of the character (see `moments_region_2nd_invar`, 3 features).

'moments_region_2nd_rel_invar' Normalized 2nd relative moments of the character (see `moments_region_2nd_rel_invar`, 2 features).

'moments_region_3rd_invar' Normalized 3rd moments of the character (see `moments_region_3rd_invar`, 4 features).

'moments_central' Normalized central moments of the character (see `moments_region_central`, 4 features).

'moments_gray_plane' Normalized gray value moments and the angle of the gray value plane (see `moments_gray_plane`, 4 features).

'phi' Sinus and cosinus of the orientation (angle) of the character (see `elliptic_axis`, 2 feature).

'num_connect' Number of connected components (see `connect_and_holes`, 1 feature).

'num_holes' Number of holes (see `connect_and_holes`, 1 feature).

'cooc' Values of the binary cooccurrence matrix (see `gen_cooc_matrix`, 8 features).

'num_runs' Number of runs in the region normalized by the height (1 feature).

'chord_histo' Frequency of the runs per row (not scale-invariant, `HeightCharacter` features).

After the classifier has been created, it is trained using `trainf_ocr_class_mlp`. After this, the classifier can be saved using `write_ocr_class_mlp`. Alternatively, the classifier can be used immediately after training to classify characters using `do_ocr_single_class_mlp` or `do_ocr_multi_class_mlp`.

HALCON provides a number of pretrained OCR classifiers (see the "`Solution Guide I`", chapter 'OCR', section 'Pretrained OCR Fonts'). These pretrained OCR classifiers can be read directly with `read_ocr_class_mlp` and make it possible to read a wide variety of different fonts without the need to train an OCR classifier. Therefore, it is recommended to try if one of the pretrained OCR classifiers can be used successfully. If this is the case, it is not necessary to create and train an OCR classifier.

A comparison of the MLP and the support vector machine (SVM) (see `create_ocr_class_svm`) typically shows that SVMs are generally faster at training, especially for huge training sets, and achieve slightly better recognition rates than MLPs. The MLP is faster at classification and should therefore be preferred in time critical applications. Please note that this guideline assumes optimal tuning of the parameters.

Parameters

▷ **`WidthCharacter`** (input_control) . . . . . integer ⇝ *integer*
Width of the rectangle to which the gray values of the segmented character are zoomed.
Default: 8
Suggested values: `WidthCharacter` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20}
Value range: $4 \leq$ `WidthCharacter` ≤ 20

▷ **`HeightCharacter`** (input_control) . . . . . integer ⇝ *integer*
Height of the rectangle to which the gray values of the segmented character are zoomed.
Default: 10
Suggested values: `HeightCharacter` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20}
Value range: $4 \leq$ `HeightCharacter` ≤ 20

▷ **`Interpolation`** (input_control) . . . . . string ⇝ *string*
Interpolation mode for the zooming of the characters.
Default: 'constant'
List of values: `Interpolation` ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'constant', 'weighted'}

▷ **Features** (input_control) . . . . . string(-array) ⇝ *string*
Features to be used for classification.
Default: 'default'
List of values: Features ∈ {'default', 'pixel', 'pixel_invar', 'pixel_binary', 'gradient_8dir', 'projection_horizontal', 'projection_horizontal_invar', 'projection_vertical', 'projection_vertical_invar', 'ratio', 'anisometry', 'width', 'height', 'zoom_factor', 'foreground', 'foreground_grid_9', 'foreground_grid_16', 'compactness', 'convexity', 'moments_region_2nd_invar', 'moments_region_2nd_rel_invar', 'moments_region_3rd_invar', 'moments_central', 'moments_gray_plane', 'phi', 'num_connect', 'num_holes', 'cooc', 'num_runs', 'chord_histo'}

▷ **Characters** (input_control) . . . . . string-array ⇝ *string*
All characters of the character set to be read.
Default: ['0','1','2','3','4','5','6','7','8','9']

▷ **NumHidden** (input_control) . . . . . integer ⇝ *integer*
Number of hidden units of the MLP.
Default: 80
Suggested values: NumHidden ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 150}
Restriction: NumHidden >= 1

▷ **Preprocessing** (input_control) . . . . . string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
Default: 'none'
List of values: Preprocessing ∈ {'none', 'normalization', 'principal_components', 'canonical_variates'}

▷ **NumComponents** (input_control) . . . . . integer ⇝ *integer*
Preprocessing parameter: Number of transformed features (ignored for Preprocessing = *'none'* and Preprocessing = *'normalization'*).
Default: 10
Suggested values: NumComponents ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Restriction: NumComponents >= 1

▷ **RandSeed** (input_control) . . . . . integer ⇝ *integer*
Seed value of the random number generator that is used to initialize the MLP with random values.
Default: 42

▷ **OCRHandle** (output_control) . . . . . ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

Example

```
read_image (Image, 'letters')
* Segment the image.
binary_threshold(Image,&Region, 'otsu', 'dark', &UsedThreshold);
dilation_circle (Region, RegionDilation, 3.5)
connection (RegionDilation, ConnectedRegions)
intersection (ConnectedRegions, Region, RegionIntersection)
sort_region (RegionIntersection, Characters, 'character', 'true', 'row')
* Generate the training file.
count_obj (Characters, Number)
Classes := []
for J := 0 to 25 by 1
    Classes := [Classes,gen_tuple_const(20,chr(ord('a')+J))]
endfor
Classes := [Classes,gen_tuple_const(20,'.')]
write_ocr_trainf (Characters, Image, Classes, 'letters.trf')
* Generate and train the classifier.
read_ocr_trainf_names ('letters.trf', CharacterNames, CharacterCount)
create_ocr_class_mlp (8, 10, 'constant', 'default', CharacterNames, 20, \
                      'none', 81, 42, OCRHandle)
trainf_ocr_class_mlp (OCRHandle, 'letters.trf', 100, 0.01, 0.01, Error, \
                      ErrorLog)
* Re-classify the characters in the image.
do_ocr_multi_class_mlp (Characters, Image, OCRHandle, Class, Confidence)
```

Result

If the parameters are valid, the operator `create_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`trainf_ocr_class_mlp`, `set_regularization_params_ocr_class_mlp`, `set_rejection_params_ocr_class_mlp`

Alternatives

`create_ocr_class_svm`

See also

`do_ocr_single_class_mlp`, `do_ocr_multi_class_mlp`, `clear_ocr_class_mlp`, `create_class_mlp`, `train_class_mlp`, `classify_class_mlp`

Module

OCR/OCV

deserialize_ocr_class_mlp (: : SerializedItemHandle : OCRHandle)

Deserialize a serialized MLP-based OCR classifier.

`deserialize_ocr_class_mlp` deserializes a MLP-based OCR classifier, that was serialized by `serialize_ocr_class_mlp` (see `fwrite_serialized_item` for an introduction of the basic principle of serialization). The serialized OCR classifier is defined by the handle `SerializedItemHandle`. The deserialized values are stored in an automatically created OCR classifier with the handle `OCRHandle`.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **OCRHandle** (output_control) ..... ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

Result

If the parameters are valid, the operator `deserialize_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_ocr_class_mlp`

Possible Successors

`do_ocr_single_class_mlp`, `do_ocr_multi_class_mlp`

See also

`create_ocr_class_mlp`, `write_ocr_class_mlp`, `read_class_mlp`, `write_class_mlp`, `serialize_ocr_class_mlp`

Module

OCR/OCV

do_ocr_multi_class_mlp (Character, Image : : OCRHandle : Class, Confidence)

Classify multiple characters with an OCR classifier.

do_ocr_multi_class_mlp computes the best class for each of the characters given by the regions Character and the gray values Image with the OCR classifier OCRHandle and returns the classes in Class and the corresponding confidences (probabilities) of the classes in Confidence. In contrast to do_ocr_single_class_mlp, do_ocr_multi_class_mlp can classify multiple characters in one call, and therefore typically is faster than a loop that uses do_ocr_single_class_mlp to classify single characters. However, do_ocr_multi_class_mlp can only return the best class of each character. Because the confidences can be interpreted as probabilities (see classify_class_mlp and evaluate_class_mlp), and it is therefore easy to check whether a character has been classified with too much uncertainty, this is usually not a disadvantage, except in cases where the classes overlap so much that in many cases the second best class must be examined to be able to decide the class of the character. In these cases, do_ocr_single_class_mlp should be used.

A string of the number '\032' (alternatively displayed as '\0x1A') in Class signifies that the region has been classified as rejection class.

Before calling do_ocr_multi_class_mlp, the classifier must be trained with trainf_ocr_class_mlp.

Parameters

- ▷ **Character** (input_object) ..... region(-array) ⇝ *object*
  Characters to be recognized.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object* : byte / uint2
  Gray values of the characters.
- ▷ **OCRHandle** (input_control) ..... ocr_mlp ⇝ *handle*
  Handle of the OCR classifier.
- ▷ **Class** (output_control) ..... string(-array) ⇝ *string*
  Result of classifying the characters with the MLP.
  Number of elements: Class == Character
- ▷ **Confidence** (output_control) ..... real(-array) ⇝ *real*
  Confidence of the class of the characters.
  Number of elements: Confidence == Character

Result

If the parameters are valid, the operator do_ocr_multi_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

trainf_ocr_class_mlp, read_ocr_class_mlp

Alternatives

do_ocr_word_mlp, do_ocr_single_class_mlp

See also

create_ocr_class_mlp, classify_class_mlp

Module

OCR/OCV

do_ocr_single_class_mlp (Character, Image : : OCRHandle, Num : Class, Confidence)

Classify a single character with an OCR classifier.

do_ocr_single_class_mlp computes the best Num classes of the character given by the region Character and the gray values Image with the OCR classifier OCRHandle and returns the classes in Class and the corresponding confidences (probabilities) of the classes in Confidence. Because multiple classes may be returned by do_ocr_single_class_mlp, Character may only contain a single region (a single character). If multiple characters should be classified in a single call, do_ocr_multi_class_mlp must be used. Because do_ocr_multi_class_mlp typically is faster than a loop with do_ocr_single_class_mlp and because the confidences can be interpreted as probabilities (see classify_class_mlp and evaluate_class_mlp), and it is therefore easy to check whether a character has been classified with too much uncertainty, in most cases do_ocr_multi_class_mlp should be used, unless the second-best class should be examined explicitly.

A string of the number '\032' (alternatively displayed as '\0x1A') in Class signifies that the region has been classified as rejection class.

Before calling do_ocr_single_class_mlp, the classifier must be trained with trainf_ocr_class_mlp.

Parameters

- ▷ **Character** (input_object) ..... region ⇝ *object*
  Character to be recognized.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object* : byte / uint2
  Gray values of the character.
- ▷ **OCRHandle** (input_control) ..... ocr_mlp ⇝ *handle*
  Handle of the OCR classifier.
- ▷ **Num** (input_control) ..... integer-array ⇝ *integer*
  Number of best classes to determine.
  Default: 1
  Suggested values: Num ∈ {1, 2, 3, 4, 5}
- ▷ **Class** (output_control) ..... string(-array) ⇝ *string*
  Result of classifying the character with the MLP.
- ▷ **Confidence** (output_control) ..... real(-array) ⇝ *real*
  Confidence(s) of the class(es) of the character.

Result

If the parameters are valid, the operator do_ocr_single_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_mlp, read_ocr_class_mlp

Alternatives

do_ocr_multi_class_mlp

See also

create_ocr_class_mlp, classify_class_mlp

Module

OCR/OCV

do_ocr_word_mlp (Character, Image : : OCRHandle, Expression, NumAlternatives, NumCorrections : Class, Confidence, Word, Score)

Classify a related group of characters with an OCR classifier.

do_ocr_word_mlp works like do_ocr_multi_class_mlp insofar as it computes the best class for each of the characters given by the regions Character and the gray values Image with the OCR classifier OCRHandle, and returns the classes in Class and the corresponding confidences (probabilities) of the classes in Confidence.

In contrast to do_ocr_multi_class_mlp, do_ocr_word_mlp treats the group of characters as an entity which yields a Word by concatenating the class names for each character region. This allows to restrict the allowed classification results on a textual level by specifying an Expression describing the expected word.

The Expression may restrict the word to belong to a predefined lexicon created using create_lexicon or import_lexicon, by specifying the name of the lexicon in angular brackets as in *'<mylexicon>'*. If the Expression is of any other form, it is interpreted as a regular expression with the same syntax as specified for tuple_regexp_match. Note that you will usually want to use an expression of the form '^...$' when using variable quantifiers like '*', to ensure that the entire word is used in the expression. Also note that in contrast to tuple_regexp_match, do_ocr_word_mlp does **not** support passing extra options in an expression tuple.

If the word derived from the best class for each character does not match the Expression, do_ocr_word_mlp attempts to correct it by considering the NumAlternatives best classes for each character. The alternatives used are identical to those returned by do_ocr_single_class_mlp for a single character. It does so by testing all possible corrections for which the classification result is changed for at most NumCorrections character regions. Note that NumAlternatives and NumCorrections affect the complexity of the algorithm, so that in some cases internal restrictions are made. See the section 'Complexity' below for further information.

In case the Expression is a lexicon and the above procedure did not yield a result, the most similar word in the lexicon is returned as long as it requires less than NumCorrections edit operations for the correction (see suggest_lexicon).

The resulting word is graded by a Score between 0.0 (no correction found) and 1.0 (original word correct). The Score is lowered by adding a penalty according to the number of corrected characters and another (minor) penalty depending on how many classes with higher confidences have been ignored in order to match the Expression:

$$\mathtt{Score} = 1.0 - \mathtt{PenaltyCorrections} - \mathtt{PenaltyAlternatives}$$

$$\begin{aligned} \mathtt{PenaltyCorrections} &= \alpha * \mathtt{num\_corr} \\ \mathtt{PenaltyAlternatives} &= \alpha * \beta * \mathtt{num\_alt} \end{aligned}$$

with num_corr being the actual number of applied corrections and num_alt the total number of discarded alternatives.

$$\begin{aligned} \alpha &= \frac{1}{\mathtt{NumCorrections} + 1} \\ \beta &= \frac{1}{\mathtt{NumCorrections} * (\mathtt{NumAlternatives} - 1) + 2} \end{aligned}$$

Note that this is a combinatorial score which does **not** reflect the original Confidence of the best Class.

A string of the number '\032' (alternatively displayed as '\0x1A') in Class signifies that the region has been classified as rejection class.

Parameters

- ▷ **Character** (input_object) ..... region(-array) ⇝ *object*
  Characters to be recognized.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
  Gray values of the characters.
- ▷ **OCRHandle** (input_control) ..... ocr_mlp ⇝ *handle*
  Handle of the OCR classifier.
- ▷ **Expression** (input_control) ..... string ⇝ *string*
  Expression describing the allowed word structure.

▷ **NumAlternatives** (input_control) ..... integer ⇝ *integer*
Number of classes per character considered for the internal word correction.
Default: 3
Suggested values: NumAlternatives ∈ {3, 4, 5}
Value range: 1 ≤ NumAlternatives

▷ **NumCorrections** (input_control) ..... integer ⇝ *integer*
Maximum number of corrected characters.
Default: 2
Suggested values: NumCorrections ∈ {1, 2, 3, 4, 5}
Value range: 0 ≤ NumCorrections

▷ **Class** (output_control) ..... string(-array) ⇝ *string*
Result of classifying the characters with the MLP.
Number of elements: Class == Character

▷ **Confidence** (output_control) ..... real(-array) ⇝ *real*
Confidence of the class of the characters.
Number of elements: Confidence == Character

▷ **Word** (output_control) ..... string ⇝ *string*
Word text after classification and correction.

▷ **Score** (output_control) ..... real ⇝ *real*
Measure of similarity between corrected word and uncorrected classification results.

Complexity

The complexity of checking all possible corrections is of magnitude $O((n * a)^{min(c,n)})$, where a is the number of alternatives, n is the number of character regions, and c is the number of allowed corrections. However, to guard against a near-infinite loop in case of large n, c is internally clipped to 5, 3, or 1 if $a * n >= 30$, 60, or 90, respectively.

Result

If the parameters are valid, the operator do_ocr_word_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_mlp, read_ocr_class_mlp

Alternatives

do_ocr_multi_class_mlp

See also

create_ocr_class_mlp, classify_class_mlp

Module

OCR/OCV

get_features_ocr_class_mlp (Character : : OCRHandle, Transform : Features)

Compute the features of a character.

get_features_ocr_class_mlp computes the features of the character given by Character with the OCR classifier OCRHandle and returns them in Features. In contrast to do_ocr_single_class_mlp and do_ocr_multi_class_mlp, the character is passed as a single image object. Hence, before calling get_features_ocr_class_mlp, reduce_domain must typically be called. The parameter Transform determines whether the feature transformation specified with Preprocessing in create_ocr_class_mlp for the classifier should be applied (Transform = *'true'*) or whether the untransformed features should be returned (Transform = *'false'*). get_features_ocr_class_mlp can be used to inspect the features that are used for the classification.

Parameters

▷ **Character** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input character.

▷ **OCRHandle** (input_control) . . . . . ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **Transform** (input_control) . . . . . string ⇝ *string*
Should the feature vector be transformed with the preprocessing?
Default: 'true'
List of values: `Transform` ∈ {'true', 'false'}

▷ **Features** (output_control) . . . . . real-array ⇝ *real*
Feature vector of the character.

Result

If the parameters are valid, the operator `get_features_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`trainf_ocr_class_mlp`

See also

`create_ocr_class_mlp`

Module

OCR/OCV

get_params_ocr_class_mlp (: : OCRHandle : WidthCharacter, HeightCharacter, Interpolation, Features, Characters, NumHidden, Preprocessing, NumComponents)

Return the parameters of an OCR classifier.

`get_params_ocr_class_mlp` returns the parameters of an OCR classifier that were specified when the classifier was created with `create_ocr_class_mlp`. This is particularly useful if the classifier was read with `read_ocr_class_mlp`. The output of `get_params_ocr_class_mlp` can, for example, be used to check whether a character to be read is contained in the classifier. For a description of the parameters, see `create_ocr_class_mlp`.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **WidthCharacter** (output_control) . . . . . integer ⇝ *integer*
Width of the rectangle to which the gray values of the segmented character are zoomed.

▷ **HeightCharacter** (output_control) . . . . . integer ⇝ *integer*
Height of the rectangle to which the gray values of the segmented character are zoomed.

▷ **Interpolation** (output_control) . . . . . string ⇝ *string*
Interpolation mode for the zooming of the characters.

▷ **Features** (output_control) . . . . . string(-array) ⇝ *string*
Features to be used for classification.

▷ **Characters** (output_control) . . . . . string-array ⇝ *string*
Characters of the character set to be read.

▷ **NumHidden** (output_control) . . . . . integer ⇝ *integer*
Number of hidden units of the MLP.

▷ **Preprocessing** (output_control) ..... string ⇝ *string*
Type of preprocessing used to transform the feature vectors.

▷ **NumComponents** (output_control) ..... integer ⇝ *integer*
Preprocessing parameter: Number of transformed features.

Result

If the parameters are valid, the operator `get_params_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_ocr_class_mlp`, `read_ocr_class_mlp`

Possible Successors

`do_ocr_single_class_mlp`, `do_ocr_multi_class_mlp`

See also

`trainf_ocr_class_mlp`, `get_params_class_mlp`

Module

OCR/OCV

get_prep_info_ocr_class_mlp (: : OCRHandle, TrainingFile, Preprocessing : InformationCont, CumInformationCont)

Compute the information content of the preprocessed feature vectors of an OCR classifier.

`get_prep_info_ocr_class_mlp` computes the information content of the training vectors that have been transformed with the preprocessing given by `Preprocessing`. `Preprocessing` can be set to *'principal_components'* or *'canonical_variates'*. The OCR classifier `OCRHandle` must have been created with `create_ocr_class_mlp`. The preprocessing methods are described with `create_class_mlp`. The information content is derived from the variations of the transformed components of the feature vector, i.e., it is computed solely based on the training data, independent of any error rate on the training data. The information content is computed for all relevant components of the transformed feature vectors (`NumInput` for *'principal_components'* and $\min(\texttt{NumOutput}-1, \texttt{NumInput})$ for *'canonical_variates'*, see `create_class_mlp`), and is returned in `InformationCont` as a number between 0 and 1. To convert the information content into a percentage, it simply needs to be multiplied by 100. The cumulative information content of the first n components is returned in the n-th component of `CumInformationCont`, i.e., `CumInformationCont` contains the sums of the first n elements of `InformationCont`. To use `get_prep_info_ocr_class_mlp`, a sufficient number of samples must be stored in the training files given by `TrainingFile` (see `write_ocr_trainf`).

`InformationCont` and `CumInformationCont` can be used to decide how many components of the transformed feature vectors contain relevant information. An often used criterion is to require that the transformed data must represent x% (e.g., 90%) of the total data. This can be decided easily from the first value of `CumInformationCont` that lies above x%. The number thus obtained can be used as the value for `NumComponents` in a new call to `create_ocr_class_mlp`. The call to `get_prep_info_ocr_class_mlp` already requires the creation of a classifier, and hence the setting of `NumComponents` in `create_ocr_class_mlp` to an initial value. However, if `get_prep_info_ocr_class_mlp` is called it is typically not known how many components are relevant, and hence how to set `NumComponents` in this call. Therefore, the following two-step approach should typically be used to select `NumComponents`: In a first step, a classifier with the maximum number for `NumComponents` is created (`NumInput` for *'principal_components'* and $\min(\texttt{NumOutput}-1, \texttt{NumInput})$ for *'canonical_variates'*). Then, the training samples are saved in a training file using `write_ocr_trainf`. Subsequently, `get_prep_info_ocr_class_mlp` is used to determine the information content of the components, and with this `NumComponents`. After this, a new classifier with the desired number of components is created, and the classifier is trained with `trainf_ocr_class_mlp`.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **TrainingFile** (input_control) ..... filename.read(-array) ⇝ *string*
Names of the training files.
Default: 'ocr.trf'
File extension: `.trf`, `.otr`

▷ **Preprocessing** (input_control) ..... string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
Default: 'principal_components'
List of values: `Preprocessing` ∈ {'principal_components', 'canonical_variates'}

▷ **InformationCont** (output_control) ..... real-array ⇝ *real*
Relative information content of the transformed feature vectors.

▷ **CumInformationCont** (output_control) ..... real-array ⇝ *real*
Cumulative information content of the transformed feature vectors.

Example

```
* Create the initial OCR classifier.
read_ocr_trainf_names ('ocr.trf', CharacterNames, CharacterCount)
create_ocr_class_mlp (8, 10, 'constant', 'default', CharacterNames, 80, \
                      'canonical_variates', |CharacterNames|, 42, OCRHandle)
* Get the information content of the transformed feature vectors.
get_prep_info_ocr_class_mlp (OCRHandle, 'ocr.trf', 'canonical_variates', \
                             InformationCont, CumInformationCont)
* Determine the number of transformed components.
* NumComp = [...]
* Create the final OCR classifier.
create_ocr_class_mlp (8, 10, 'constant', 'default', CharacterNames, 80, \
                      'canonical_variates', NumComp, 42, OCRHandle)
* Train the final classifier.
trainf_ocr_class_mlp (OCRHandle, 'ocr.trf', 100, 1, 0.01, Error, ErrorLog)
write_ocr_class_mlp (OCRHandle, 'ocr.omc')
```

Result

If the parameters are valid, the operator `get_prep_info_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

`get_prep_info_ocr_class_mlp` may return the error 9211 (Matrix is not positive definite) if `Preprocessing` = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_ocr_class_mlp`, `write_ocr_trainf`, `append_ocr_trainf`, `write_ocr_trainf_image`

Possible Successors

`clear_ocr_class_mlp`, `create_ocr_class_mlp`

Module

OCR/OCV

get_regularization_params_ocr_class_mlp (: : OCRHandle, GenParamName : GenParamValue)

Return the regularization parameters of an OCR classifier.

get_regularization_params_ocr_class_mlp returns the regularization parameters of an OCR classifier that were specified with set_regularization_params_ocr_class_mlp. Furthermore, get_regularization_params_ocr_class_mlp returns the parameters that were determined by an automatic determination of the regularization parameters. For a description of the parameters, see set_regularization_params_ocr_class_mlp.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **GenParamName** (input_control) ..... string ⇝ *string*
Name of the regularization parameter to return.
Default: 'weight_prior'
List of values: GenParamName ∈ {'weight_prior', 'num_well_determined_params', 'fraction_well_determined_params', 'num_outer_iterations', 'num_inner_iterations'}

▷ **GenParamValue** (output_control) ..... number(-array) ⇝ *real* / integer
Value of the regularization parameter.

Result

If the parameters are valid, the operator get_regularization_params_ocr_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_regularization_params_ocr_class_mlp, read_ocr_class_mlp

Possible Successors

trainf_ocr_class_mlp

Module

OCR/OCV

get_rejection_params_ocr_class_mlp (: : OCRHandle, GenParamName : GenParamValue)

Return the rejection class parameters of an OCR classifier.

get_rejection_params_ocr_class_mlp returns the rejection class parameters of an OCR classifier that were specified with set_rejection_params_ocr_class_mlp. For a description of the parameters, see set_rejection_params_ocr_class_mlp and set_rejection_params_class_mlp.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **GenParamName** (input_control) ..... string(-array) ⇝ *string*
Name of the general parameter.
Default: 'sampling_strategy'
List of values: GenParamName ∈ {'sampling_strategy', 'hyperbox_tolerance', 'rejection_sample_factor', 'random_seed', 'rejection_class_index'}

▷ **GenParamValue** (output_control) ..... string(-array) ⇝ *string*
Value of the general parameter.

Result

If the parameters are valid, the operator `get_rejection_params_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_ocr_class_mlp`, `read_ocr_class_mlp`

Possible Successors

`trainf_ocr_class_mlp`

Module

OCR/OCV

read_ocr_class_mlp (: : FileName : OCRHandle)

Read an OCR classifier from a file.

`read_ocr_class_mlp` reads an OCR classifier that has been stored with `write_ocr_class_mlp`. Since the training of an OCR classifier can consume a relatively long time, the classifier is typically trained in an offline process and written to a file with `write_ocr_class_mlp`. In the online process the classifier is read with `read_ocr_class_mlp` and subsequently used for classification with `do_ocr_single_class_mlp` or `do_ocr_multi_class_mlp`.

HALCON provides a number of pretrained OCR classifiers (see the "`Solution Guide I`", chapter 'OCR', section 'Pretrained OCR Fonts'). These pretrained OCR classifiers make it possible to read a wide variety of different fonts without the need to train an OCR classifier. Note that the pretrained OCR classifiers were trained with symbols that are printed dark on light.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
Suggested values: `FileName` ∈ {'Document_0-9_NoRej.omc', 'Document_0-9_Rej.omc', 'Document_0-9A-Z_NoRej.omc', 'Document_0-9A-Z_Rej.omc', 'Document_A-Z+_NoRej.omc', 'Document_A-Z+_Rej.omc', 'Document_NoRej.omc', 'Document_Rej.omc', 'DotPrint_0-9_NoRej.omc', 'DotPrint_0-9_Rej.omc', 'DotPrint_0-9+_NoRej.omc', 'DotPrint_0-9+_Rej.omc', 'DotPrint_0-9A-Z_NoRej.omc', 'DotPrint_0-9A-Z_Rej.omc', 'DotPrint_A-Z+_NoRej.omc', 'DotPrint_A-Z+_Rej.omc', 'DotPrint_NoRej.omc', 'DotPrint_Rej.omc', 'HandWritten_0-9_NoRej.omc', 'HandWritten_0-9_Rej.omc', 'Industrial_0-9_NoRej.omc', 'Industrial_0-9_Rej.omc', 'Industrial_0-9+_NoRej.omc', 'Industrial_0-9+_Rej.omc', 'Industrial_0-9A-Z_NoRej.omc', 'Industrial_0-9A-Z_Rej.omc', 'Industrial_A-Z+_NoRej.omc', 'Industrial_A-Z+_Rej.omc', 'Industrial_NoRej.omc', 'Industrial_Rej.omc', 'OCRA_0-9_NoRej.omc', 'OCRA_0-9_Rej.omc', 'OCRA_0-9A-Z_NoRej.omc', 'OCRA_0-9A-Z_Rej.omc', 'OCRA_A-Z+_NoRej.omc', 'OCRA_A-Z+_Rej.omc', 'OCRA_NoRej.omc', 'OCRA_Rej.omc', 'OCRB_0-9_NoRej.omc', 'OCRB_0-9_Rej.omc', 'OCRB_0-9A-Z_NoRej.omc', 'OCRB_0-9A-Z_Rej.omc', 'OCRB_A-Z+_NoRej.omc', 'OCRB_A-Z+_Rej.omc', 'OCRB_NoRej.omc', 'OCRB_passport_NoRej.omc', 'OCRB_passport_Rej.omc', 'OCRB_Rej.omc', 'Pharma_0-9_NoRej.omc', 'Pharma_0-9_Rej.omc', 'Pharma_0-9+_NoRej.omc', 'Pharma_0-9+_Rej.omc', 'Pharma_0-9A-Z_NoRej.omc', 'Pharma_0-9A-Z_Rej.omc', 'Pharma_NoRej.omc', 'Pharma_Rej.omc', 'SEMI_NoRej.omc', 'SEMI_Rej.omc'}
File extension: `.omc`, `.fnt`

▷ **OCRHandle** (output_control) . . . . . ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

Result

If the parameters are valid, the operator `read_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`do_ocr_single_class_mlp`, `do_ocr_multi_class_mlp`

See also

`create_ocr_class_mlp`, `write_ocr_class_mlp`, `read_class_mlp`, `write_class_mlp`

Module

OCR/OCV

select_feature_set_trainf_mlp (: : TrainingFile, FeatureList, SelectionMethod, Width, Height, GenParamName, GenParamValue : OCRHandle, FeatureSet, Score)

Selects an optimal combination of features to classify OCR data.

`select_feature_set_trainf_mlp` selects an optimal combination of features, to classify the OCR data given in the training file `TrainingFile` with a multilayer perceptron, for details see `create_ocr_class_mlp`.

Possible features are all OCR features listed and explained in `create_ocr_class_mlp`. All candidates which should be tested can be specified in `FeatureList`. A subset of these features is returned as selected features in `FeatureSet`.

`select_feature_set_trainf_mlp` is specialized on OCR problems and only supports the features in the list mentioned before. In order to use other features, please use the more general operator `select_feature_set_mlp`.

The selection method `SelectionMethod` is either a greedy search *'greedy'* (iteratively add the feature with highest gain) or the dynamically oscillating search *'greedy_oscillating'* (add the feature with highest gain and test then if any of the already added features can be left out without great loss). The method *'greedy'* is generally preferable, since it is faster. Only in cases when a large training set is available the method *'greedy_oscillating'* might return better results.

The optimization criterion is the classification rate of a two-fold cross-validation of the training data. The best achieved value is returned in `Score`.

The parameters `GenParamName` and `GenParamValue` allow to adapt the setting of the number of hidden neurons in the MLP with *'num_hidden'*. The default value is *80*, a higher value leads to longer training times but might lead to a more expressive classifier.

Attention

This operator may take considerable time, depending on the size of the data set in the training file, and the number of features.

Please note, that this operator should not be called, if only a small set of training data is available. Due to the risk of overfitting the operator `select_feature_set_trainf_mlp` may deliver a classifier with a very high score. However, the classifier may perform poorly when tested.

Parameters

▷ **TrainingFile** (input_control) . . . . . filename.read(-array) ⇝ *string*
Names of the training files.
Default: ”
File extension: `.trf`, `.otr`

▷ **FeatureList** (input_control) . . . . . string(-array) ⇝ *string*
List of features that should be considered for selection.
Default: [’zoom_factor’,’ratio’,’width’,’height’,’foreground’,’foreground_grid_9’,’foreground_grid_16’, ’anisometry’,’compactness’,’convexity’,’moments_region_2nd_invar’,’moments_region_2nd_rel_invar’, ’moments_region_3rd_invar’,’moments_central’,’phi’,’num_connect’,’num_holes’,’projection_horizontal’, ’projection_vertical’,’projection_horizontal_invar’,’projection_vertical_invar’,’chord_histo’,’num_runs’, ’pixel’,’pixel_invar’,’pixel_binary’,’gradient_8dir’,’cooc’,’moments_gray_plane’]
List of values: `FeatureList` ∈ {’default’, ’zoom_factor’, ’ratio’, ’width’, ’height’, ’foreground’, ’foreground_grid_9’, ’foreground_grid_16’, ’anisometry’, ’compactness’, ’convexity’, ’moments_region_2nd_invar’, ’moments_region_2nd_rel_invar’, ’moments_region_3rd_invar’, ’moments_central’, ’phi’, ’num_connect’, ’num_holes’, ’projection_horizontal’, ’projection_vertical’, ’projection_horizontal_invar’, ’projection_vertical_invar’, ’chord_histo’, ’num_runs’, ’pixel’, ’pixel_invar’, ’pixel_binary’, ’gradient_8dir’, ’cooc’, ’moments_gray_plane’}

▷ **SelectionMethod** (input_control) . . . . . string ⇝ *string*
Method to perform the selection.
Default: ’greedy’
List of values: `SelectionMethod` ∈ {’greedy’, ’greedy_oscillating’}

▷ **Width** (input_control) . . . . . integer ⇝ *integer*
Width of the rectangle to which the gray values of the segmented character are zoomed.
Default: 15

▷ **Height** (input_control) . . . . . integer ⇝ *integer*
Height of the rectangle to which the gray values of the segmented character are zoomed.
Default: 16

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of generic parameters to configure the selection process and the classifier.
Default: []
List of values: `GenParamName` ∈ {’nu’}

▷ **GenParamValue** (input_control) . . . . . number-array ⇝ *real* / integer / string
Values of generic parameters to configure the selection process and the classifier.
Default: []
Suggested values: `GenParamValue` ∈ {’0.1’}

▷ **OCRHandle** (output_control) . . . . . ocr_mlp ⇝ *handle*
Trained OCR-MLP classifier.

▷ **FeatureSet** (output_control) . . . . . string-array ⇝ *string*
Selected feature set, contains only entries from `FeatureList`.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Achieved score using tow-fold cross-validation.

Result

If the parameters are valid, the operator `select_feature_set_trainf_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Alternatives

`select_feature_set_trainf_svm`, `select_feature_set_trainf_knn`

See also

select_feature_set_trainf_mlp_protected, select_feature_set_mlp

Module

OCR/OCV

select_feature_set_trainf_mlp_protected (: : TrainingFile, Password, FeatureList, SelectionMethod, Width, Height, GenParamName, GenParamValue : OCRHandle, FeatureSet, Score)

Select an optimal combination of features to classify OCR data from a (protected) training file.

select_feature_set_trainf_mlp_protected selects an optimal combination of features to classify the data given in the training files TrainingFile with a multilayer perceptron (MLP). Its functionality corresponds to the functionality of select_feature_set_trainf_mlp, with the addition that select_feature_set_trainf_mlp_protected can process unprotected and protected training files. Protected training files can be used only with the correct user password Password. If the number of passwords Password equals 1, then every input file TrainingFile is checked with that password, otherwise the number of passwords has to be equal to the number of input files and the input file at position n is checked with the password at position n. For unprotected training files the passwords are ignored.

For a more detailed description of the operator's functionality see select_feature_set_trainf_mlp. The concept of protecting OCR training data in HALCON is described in protect_ocr_trainf.

Attention

This operator may take considerable time, depending on the size of the data set in the training file, and the number of features.

Please note, that this operator should not be called, if only a small set of training data is available. Due to the risk of overfitting the operator select_feature_set_trainf_mlp_protected may deliver a classifier with a very high score. However, the classifier may perform poorly when tested.

Parameters

▷ **TrainingFile** (input_control) ..... filename.read(-array) ⇝ *string*
Names of the training files.
Default: ''
File extension: .trf, .otr

▷ **Password** (input_control) ..... string(-array) ⇝ *string*
Passwords for protected training files.

▷ **FeatureList** (input_control) ..... string(-array) ⇝ *string*
List of features that should be considered for selection.
Default: ['zoom_factor','ratio','width','height','foreground','foreground_grid_9','foreground_grid_16', 'anisometry','compactness','convexity','moments_region_2nd_invar','moments_region_2nd_rel_invar', 'moments_region_3rd_invar','moments_central','phi','num_connect','num_holes','projection_horizontal', 'projection_vertical','projection_horizontal_invar','projection_vertical_invar','chord_histo','num_runs', 'pixel','pixel_invar','pixel_binary','gradient_8dir','cooc','moments_gray_plane']
List of values: FeatureList ∈ {'default', 'zoom_factor', 'ratio', 'width', 'height', 'foreground', 'foreground_grid_9', 'foreground_grid_16', 'anisometry', 'compactness', 'convexity', 'moments_region_2nd_invar', 'moments_region_2nd_rel_invar', 'moments_region_3rd_invar', 'moments_central', 'phi', 'num_connect', 'num_holes', 'projection_horizontal', 'projection_vertical', 'projection_horizontal_invar', 'projection_vertical_invar', 'chord_histo', 'num_runs', 'pixel', 'pixel_invar', 'pixel_binary', 'gradient_8dir', 'cooc', 'moments_gray_plane'}

▷ **SelectionMethod** (input_control) ..... string ⇝ *string*
Method to perform the selection.
Default: 'greedy'
List of values: SelectionMethod ∈ {'greedy', 'greedy_oscillating'}

▷ **Width** (input_control) ..... integer ⇝ *integer*
Width of the rectangle to which the gray values of the segmented character are zoomed.
Default: 15

▷ **Height** (input_control) . . . . . integer ⇝ *integer*
Height of the rectangle to which the gray values of the segmented character are zoomed.
Default: 16

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of generic parameters to configure the selection process and the classifier.
Default: []
List of values: GenParamName ∈ {'nu'}

▷ **GenParamValue** (input_control) . . . . . number-array ⇝ *real* / integer / string
Values of generic parameters to configure the selection process and the classifier.
Default: []
Suggested values: GenParamValue ∈ {'0.1'}

▷ **OCRHandle** (output_control) . . . . . ocr_mlp ⇝ *handle*
Trained OCR-MLP classifier.

▷ **FeatureSet** (output_control) . . . . . string-array ⇝ *string*
Selected feature set, contains only entries from FeatureList.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Achieved score using tow-fold cross-validation.

Result

If the parameters are valid, the operator select_feature_set_trainf_mlp_protected returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

protect_ocr_trainf

Alternatives

select_feature_set_trainf_svm_protected

See also

select_feature_set_trainf_mlp, select_feature_set_mlp

Module

OCR/OCV

serialize_ocr_class_mlp (: : OCRHandle : SerializedItemHandle)

Serialize a MLP-based OCR classifier.

serialize_ocr_class_mlp serializes a MLP-based OCR classifier (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_ocr_class_mlp is converted to a serialized item. The OCR classifier is defined by the handle OCRHandle. The serialized OCR classifier is returned by the handle SerializedItemHandle and can be deserialized by deserialize_ocr_class_mlp.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`trainf_ocr_class_mlp`

Possible Successors

`clear_ocr_class_mlp`, `fwrite_serialized_item`, `send_serialized_item`, `deserialize_ocr_class_mlp`

See also

`create_ocr_class_mlp`, `read_ocr_class_mlp`, `write_class_mlp`, `read_class_mlp`, `deserialize_ocr_class_mlp`

Module

OCR/OCV

set_regularization_params_ocr_class_mlp (: : OCRHandle, GenParamName, GenParamValue :)

Set the regularization parameters of an OCR classifier.

`set_regularization_params_ocr_class_mlp` sets the regularization parameters of the OCR classifier passed in `OCRHandle`. The regularization parameter to be set is specified with `GenParamName`. Its value is specified with `GenParamValue`.

As described at `create_class_mlp`, it may be desirable to regularize the OCR classifier (i.e., the MLP of the OCR classifier) to enforce a smoother transition of the confidences between the different classes and to prevent overfitting of the OCR classifier to the training data. To achieve this, a penalty for large MLP weights (which are the main reason for very sharp transitions between classes) can be added to the training of the OCR classifier in `trainf_ocr_class_mlp` by setting `GenParamName` to *'weight_prior'* and setting `GenParamValue` to a value > 0. Furthermore, the regularization parameters can be determined automatically. For details, see `set_regularization_params_class_mlp`. If the regularization parameters should be determined automatically, please carefully note the advice in `set_regularization_params_class_mlp` on how to select the parameters `NumHidden` of the MLP and *'num_outer_iterations'* and on the memory and runtime implications of automatically determining the regularization parameters on the runtime of the training of the OCR classifier.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **GenParamName** (input_control) ..... string ⇝ *string*
Name of the regularization parameter to return.
Default: 'weight_prior'
List of values: `GenParamName` ∈ {'weight_prior', 'num_outer_iterations', 'num_inner_iterations'}

▷ **GenParamValue** (input_control) ..... number(-array) ⇝ *real* / integer
Value of the regularization parameter.
Default: 1.0
Suggested values: `GenParamValue` ∈ {0.01, 0.1, 1.0, 10.0, 100.0, 0, 1, 2, 3, 5, 10, 15, 20}

Example

```
* This example shows how to determine the regularization parameters
* automatically without examining the convergence of the
* regularization parameters.
```

```
* Create the OCR classifier.
read_ocr_trainf_names ('ocr.trf', CharacterNames, CharacterCount)
create_ocr_class_mlp (8, 10, 'constant', 'default', CharacterNames, \
                      40, 'none', |CharacterNames|, 42, OCRHandle)
* Set up the automatic determination of the regularization
* parameters.
set_regularization_params_ocr_class_mlp (OCRHandle, 'weight_prior', \
                                         [0.01,0.01,0.01,0.01])
set_regularization_params_ocr_class_mlp (OCRHandle, \
                                         'num_outer_iterations', 10)
* Train the classifier.
trainf_ocr_class_mlp (OCRHandle, 'ocr.trf', 100, 1, 0.01, Error, \
                      ErrorLog)
* Read out the estimate of the number of well-determined
* parameters.
get_regularization_params_ocr_class_mlp (OCRHandle, \
                                         'fraction_well_determined_params', \
                                          FractionParams)
* If FractionParams differs substantially from 1, consider reducing
* NumHidden appropriately and consider performing a preprocessing that
* reduces the number of input variables to the net, i.e., canonical
* variates or principal components.
write_ocr_class_mlp (OCRHandle, 'ocr.omc')



* This example shows how to determine the regularization parameters
* automatically while examining the convergence of the
* regularization parameters.
* Create the OCR classifier.
read_ocr_trainf_names ('ocr.trf', CharacterNames, CharacterCount)
create_ocr_class_mlp (8, 10, 'constant', 'default', CharacterNames, \
                      40, 'none', |CharacterNames|, 42, OCRHandle)
* Set up the automatic determination of the regularization
* parameters.
set_regularization_params_ocr_class_mlp (OCRHandle, 'weight_prior', \
                                         [0.01,0.01,0.01,0.01])
set_regularization_params_ocr_class_mlp (OCRHandle, \
                                         'num_outer_iterations', 1)
for OuterIt := 1 to 10 by 1
    * Train the classifier.
    trainf_ocr_class_mlp (OCRHandle, 'ocr.trf', 100, 1, 0.01, Error, \
                          ErrorLog)
    * Read out the regularization parameters
    get_regularization_params_ocr_class_mlp (OCRHandle, \
                                             'weight_prior', \
                                             WeightPrior)
    * Inspect the regularization parameters manually for
    * convergence and exit the loop manually if they have
    * converged.
    * [...]
endfor
* Read out the estimate of the number of well-determined
* parameters.
get_regularization_params_ocr_class_mlp (OCRHandle, \
                                         'fraction_well_determined_params', \
                                          FractionParams)
* If FractionParams differs substantially from 1, consider reducing
* NumHidden appropriately and consider performing a preprocessing that
```

```
* reduces the number of input variables to the net, i.e., canonical
* variates or principal components.
write_ocr_class_mlp (OCRHandle, 'ocr.omc')
```

Result

If the parameters are valid, the operator `set_regularization_params_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_ocr_class_mlp`

Possible Successors

`get_regularization_params_ocr_class_mlp`, `trainf_ocr_class_mlp`

Module

OCR/OCV

set_rejection_params_ocr_class_mlp (: : OCRHandle, GenParamName, GenParamValue :)

Set the rejection class parameters of an OCR classifier.

`set_rejection_params_ocr_class_mlp` sets the rejection class parameters of the OCR classifier passed in `OCRHandle`. The parameter to be set is specified with `GenParamName`. Its value is specified with `GenParamValue`. For some OCR applications it is important to know, if a letter to be classified is similar to the training set or not. If a letter is different to the training set, then it can be classified a rejection class. This additional rejection class must be specified as an additional character in `create_ocr_class_mlp`. For details on how to specify a rejection class, see `set_rejection_params_class_mlp`.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **GenParamName** (input_control) ..... string(-array) ⇝ *string*
Name of the general parameter.
Default: 'sampling_strategy'
List of values: `GenParamName` ∈ {'sampling_strategy', 'hyperbox_tolerance', 'rejection_sample_factor', 'random_seed', 'rejection_class_index'}

▷ **GenParamValue** (input_control) ..... string(-array) ⇝ *string*
Value of the general parameter.
Default: 'hyperbox_around_all_classes'
List of values: `GenParamValue` ∈ {'no_rejection_class', 'hyperbox_around_all_classes', 'hyperbox_around_each_class', 'hyperbox_ring_around_each_class'}

Result

If the parameters are valid, the operator `set_rejection_params_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_ocr_class_mlp`, `read_ocr_class_mlp`

Possible Successors

`trainf_ocr_class_mlp`

Module

OCR/OCV

`trainf_ocr_class_mlp` (: : `OCRHandle`, `TrainingFile`, `MaxIterations`, `WeightTolerance`, `ErrorTolerance` : `Error`, `ErrorLog`)

Train an OCR classifier.

`trainf_ocr_class_mlp` trains the OCR classifier `OCRHandle` with the training characters stored in the OCR training files given by `TrainingFile`. The training files must have been created, e.g., using `write_ocr_trainf`, before calling `trainf_ocr_class_mlp`.

The remaining parameters have the same meaning as in `train_class_mlp` and are described in detail with `train_class_mlp`. A regularization of the OCR classifier and an automatic determination of the regularization parameters (see `set_regularization_params_ocr_class_mlp`) is taken into account during the training. Furthermore, if a rejection class has been specified using `set_rejection_params_ocr_class_mlp`, before the actual training the samples for the rejection class are generated.

Please note that training characters that have no corresponding class in the classifier `OCRHandle` are discarded.

Parameters

▷ **`OCRHandle`** (input_control) . . . . . ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **`TrainingFile`** (input_control) . . . . . filename.read(-array) ⇝ *string*
Names of the training files.
Default: 'ocr.trf'
File extension: `.trf`, `.otr`

▷ **`MaxIterations`** (input_control) . . . . . integer ⇝ *integer*
Maximum number of iterations of the optimization algorithm.
Default: 200
Suggested values: `MaxIterations` ∈ {20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280, 300}

▷ **`WeightTolerance`** (input_control) . . . . . real ⇝ *real*
Threshold for the difference of the weights of the MLP between two iterations of the optimization algorithm.
Default: 1.0
Suggested values: `WeightTolerance` ∈ {1.0, 0.1, 0.01, 0.001, 0.0001, 0.00001}
Restriction: WeightTolerance >= 1.0e-8

▷ **`ErrorTolerance`** (input_control) . . . . . real ⇝ *real*
Threshold for the difference of the mean error of the MLP on the training data between two iterations of the optimization algorithm.
Default: 0.01
Suggested values: `ErrorTolerance` ∈ {1.0, 0.1, 0.01, 0.001, 0.0001, 0.00001}
Restriction: ErrorTolerance >= 1.0e-8

▷ **`Error`** (output_control) . . . . . real ⇝ *real*
Mean error of the MLP on the training data.

▷ **ErrorLog** (output_control) . . . . . real-array ⇝ *real*
Mean error of the MLP on the training data as a function of the number of iterations of the optimization algorithm.

Example

```
* Train an OCR classifier
read_ocr_trainf_names ('ocr.trf', CharacterNames, CharacterCount)
create_ocr_class_mlp (8, 10, 'constant', 'default', CharacterNames, 80, \
                      'none', 81, 42, OCRHandle)
trainf_ocr_class_mlp (OCRHandle, 'ocr.trf', 100, 1, 0.01, Error, ErrorLog)
write_ocr_class_mlp (OCRHandle, 'ocr.omc')
```

Result

If the parameters are valid, the operator `trainf_ocr_class_mlp` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

`trainf_ocr_class_mlp` may return the error 9211 (Matrix is not positive definite) if `Preprocessing` = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class. In this case we recommend to change `Preprocessing` to *'normalization'*. Another solution can be to add more training samples.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_ocr_class_mlp`, `write_ocr_trainf`, `append_ocr_trainf`, `write_ocr_trainf_image`, `set_regularization_params_ocr_class_mlp`

Possible Successors

`do_ocr_single_class_mlp`, `do_ocr_multi_class_mlp`, `write_ocr_class_mlp`

Alternatives

`read_ocr_class_mlp`

See also

`train_class_mlp`

Module

OCR/OCV

trainf_ocr_class_mlp_protected (: : OCRHandle, TrainingFile, Password, MaxIterations, WeightTolerance, ErrorTolerance : Error, ErrorLog)

Train an OCR classifier with data from a (protected) training file.

`trainf_ocr_class_mlp_protected` trains the OCR classifier `OCRHandle` with the training data stored in the OCR training files given by `TrainingFile`. Its functionality corresponds to the functionality of `trainf_ocr_class_mlp`, with the addition that `trainf_ocr_class_mlp_protected` can process unprotected and protected training files. Protected training files can be used only with the correct user password `Password`. If the number of passwords `Password` equals 1, then every input file `TrainingFile` is checked

with that password, otherwise the number of passwords has to be equal to the number of input files and the input file at position n is checked with the password at position n. For unprotected training files the passwords are ignored.

For a more detailed description of the operator's functionality see `trainf_ocr_class_mlp`. The concept of protecting OCR training data in HALCON is described in `protect_ocr_trainf`.

Parameters

▷ **`OCRHandle`** (input_control) . . . . . ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **`TrainingFile`** (input_control) . . . . . filename.read(-array) ⇝ *string*
Names of the training files.
Default: 'ocr.trf'
File extension: `.trf`, `.otr`

▷ **`Password`** (input_control) . . . . . string(-array) ⇝ *string*
Passwords for protected training files.

▷ **`MaxIterations`** (input_control) . . . . . integer ⇝ *integer*
Maximum number of iterations of the optimization algorithm.
Default: 200
Suggested values: `MaxIterations` ∈ {20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280, 300}

▷ **`WeightTolerance`** (input_control) . . . . . real ⇝ *real*
Threshold for the difference of the weights of the MLP between two iterations of the optimization algorithm.
Default: 1.0
Suggested values: `WeightTolerance` ∈ {1.0, 0.1, 0.01, 0.001, 0.0001, 0.00001}
Restriction: WeightTolerance >= 1.0e-8

▷ **`ErrorTolerance`** (input_control) . . . . . real ⇝ *real*
Threshold for the difference of the mean error of the MLP on the training data between two iterations of the optimization algorithm.
Default: 0.01
Suggested values: `ErrorTolerance` ∈ {1.0, 0.1, 0.01, 0.001, 0.0001, 0.00001}
Restriction: ErrorTolerance >= 1.0e-8

▷ **`Error`** (output_control) . . . . . real ⇝ *real*
Mean error of the MLP on the training data.

▷ **`ErrorLog`** (output_control) . . . . . real-array ⇝ *real*
Mean error of the MLP on the training data as a function of the number of iterations of the optimization algorithm.

Result

If the parameters are valid, the operator `trainf_ocr_class_mlp_protected` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

`trainf_ocr_class_mlp_protected` may return the error 9211 (Matrix is not positive definite) if `Preprocessing` = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class. In this case we recommend to change `Preprocessing` to *'normalization'*. Another solution can be to add more training samples.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_ocr_class_mlp, write_ocr_trainf, append_ocr_trainf, write_ocr_trainf_image, protect_ocr_trainf

Possible Successors

do_ocr_single_class_mlp, do_ocr_multi_class_mlp, write_ocr_class_mlp

Alternatives

read_ocr_class_mlp

See also

trainf_ocr_class_mlp, train_class_mlp

Module

OCR/OCV

write_ocr_class_mlp (: : OCRHandle, FileName :)

Write an OCR classifier to a file.

write_ocr_class_mlp writes the OCR classifier OCRHandle to the file given by FileName. If a file extension is not specified in FileName the default extension '.omc' is appended to FileName. write_ocr_class_mlp is typically called after the classifier has been trained with trainf_ocr_class_mlp. The classifier can be read with read_ocr_class_mlp.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_mlp ⇝ *handle*
Handle of the OCR classifier.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
File name.
File extension: .omc

Result

If the parameters are valid, the operator write_ocr_class_mlp returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_mlp

Possible Successors

clear_ocr_class_mlp

See also

create_ocr_class_mlp, read_ocr_class_mlp, write_class_mlp, read_class_mlp

Module

OCR/OCV

21.6 Segmentation

clear_text_model (: : TextModel :)

Clear a text model.

clear_text_model clears the text model in TextModel and frees all underlying memory.

Parameters

▷ **TextModel** (input_control) . . . . . text_model(-array) ⇝ *handle*
Text model to be cleared.

Result

`clear_text_model` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TextModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

`create_text_model_reader`

Module

OCR/OCV

clear_text_result (: : TextResultID :)

Clear a text result.

`clear_text_result` clears the text result in `TextResultID` and frees all underlying memory.

Parameters

▷ **TextResultID** (input_control) . . . . . text_result(-array) ⇝ *handle*
Text result to be cleared.

Result

`clear_text_result` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TextResultID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Module

Foundation

create_text_model_reader (: : Mode, OCRClassifier : TextModel)

Create a text model.

`create_text_model_reader` creates a `TextModel`, which describes the text to be segmented with `find_text`.

The parameter value of Mode determines which text segmentation approach is used. Possible values are *'auto'* and *'manual'*.

Typically, the parameter Mode should be set to *'auto'* because this mode is more stable and requires less configuration effort. Note that in this case, also an OCR classifier must be passed in OCRClassifier. Only if one of the following restrictions apply, Mode must be set to *'manual'*:

- The segmentation of text which has strong local variations of the polarity is required. For example, due to reflections, engraved text often has strong local variations.
- No suitable OCR classifier is available (see below).

If Mode = *'auto'*, find_text is able to extract text of arbitrary size. It is possible to restrict the search to characters with specific attributes, see set_text_model_param for details. In particular, if the text to be segmented contains dot printed characters, the text model parameter *'dot_print'* must be set to *'true'*. Furthermore, an OCR classifier must be passed in OCRClassifier. This OCR classifier must be based on a convolutional neural network (CNN) or a multilayer perceptron (MLP). Moreover, it is strongly recommended to use a CNN based OCR classifier with rejection class or a MLP based classifiers that has been trained with regularization parameters (see set_regularization_params_ocr_class_mlp and provides a rejection class (see set_rejection_params_ocr_class_mlp). A suitable OCR classifier can either be read with read_ocr_class_cnn or read_ocr_class_mlp, or be created with create_ocr_class_mlp. It is also possible to pass a string containing the path to a pretrained OCR classifier or an OCR classifier that has been stored with write_ocr_class_mlp.

To enable text segmentation when Mode = *'manual'*, reasonable parameters for the text model, including the expected character height and width, must be set using set_text_model_param. In this case, the value of OCRClassifier is ignored.

The parameters of the TextModel can be set and queried with set_text_model_param and get_text_model_param.

Since memory is allocated for the text model during the call of create_text_model_reader and during the following operations, the model should be freed explicitly by the operator clear_text_model as soon as it is no longer used.

Parameters

▷ **Mode** (input_control) . . . . . string ⇝ *string*
The Mode of the text model.
Default: 'auto'
List of values: Mode ∈ {'manual', 'auto'}

▷ **OCRClassifier** (input_control) . . . . . string ⇝ *string* / integer
OCR Classifier.
Default: 'Universal_Rej.occ'
Suggested values: OCRClassifier ∈ {'Document_Rej.omc', 'Document_0-9_Rej.omc', 'Document_0-9A-Z_Rej.omc', 'Document_A-Z+_Rej.omc', 'DotPrint_Rej.omc', 'DotPrint_0-9_Rej.omc', 'DotPrint_0-9+_Rej.omc', 'DotPrint_0-9A-Z_Rej.omc', 'DotPrint_A-Z+_Rej.omc', 'HandWritten_0-9_Rej.omc', 'Industrial_Rej.omc', 'Industrial_0-9_Rej.omc', 'Industrial_0-9+_Rej.omc', 'Industrial_0-9A-Z_Rej.omc', 'Industrial_A-Z+_Rej.omc', 'OCRA_Rej.omc', 'OCRA_0-9_Rej.omc', 'OCRA_0-9A-Z_Rej.omc', 'OCRA_A-Z+_Rej.omc', 'OCRB_Rej.omc', 'OCRB_0-9_Rej.omc', 'OCRB_0-9A-Z_Rej.omc', 'OCRB_A-Z+_Rej.omc', 'OCRB_passport_Rej.omc', 'Pharma_Rej.omc', 'Pharma_0-9_Rej.omc', 'Pharma_0-9+_Rej.omc', 'Pharma_0-9A-Z_Rej.omc', 'SEMI_Rej.omc', 'Universal_Rej.occ', 'Universal_0-9_Rej.occ', 'Universal_0-9+_Rej.occ', 'Universal_0-9A-Z_Rej.occ', 'Universal_0-9A-Z+_Rej.occ', 'Universal_A-Z+_Rej.occ'}

▷ **TextModel** (output_control) . . . . . text_model ⇝ *handle*
New text model.

Example

```
read_image (Image, 'numbers_scale')
create_text_model_reader ('auto', 'Document_Rej.omc', TextModel)
* Optionally specify text properties
set_text_model_param (TextModel, 'min_char_height', 20)
find_text (Image, TextModel, TextResultID)
```

```
* Return character regions and corresponding classification results
get_text_object (Characters, TextResultID, 'all_lines')
get_text_result (TextResultID, 'class', Class)
```

Result

`create_text_model_reader` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`set_text_model_param`, `get_text_model_param`, `find_text`

See also

`clear_text_model`

Module

OCR/OCV

find_text (Image : : TextModel : TextResultID)

Find text in an image.

`find_text` finds text specified by the `TextModel` in the `Image` and returns the result in `TextResultID`. Specific results of `TextResultID` can be obtained with `get_text_result` and `get_text_object`.

The text results in `TextResultID` differ, depending on how `Mode` was set when creating the `TextModel` with `create_text_model_reader`.

For a text model with `Mode` = *'auto'* `find_text` extracts and classifies text of arbitrary size within the `Image`. It is possible to restrict the search to characters with specific attributes, see `set_text_model_param` for details.

For a text model with `Mode` = *'manual'* reasonable parameters for the text model, including the expected character height and width, need to be set using `set_text_model_param`. Furthermore, the `Image` is preprocessed such that character like structures are enhanced. The resulting compensated image can be queried from `TextResultID` if *'persistence'* was set to *'true'* in `TextModel`.

Both the text models with `Mode` = *'auto'* and `Mode` = *'manual'* apply various thresholds to the input image and segment character candidates based on region and gray value features. These candidates are further clustered to lines. Each line is individually completed and tested if it fulfills the constraints of a text line. Text models with `Mode` set to *'auto'* require a text line to consist of at least two characters, whereas text models with `Mode` set to *'manual'* require a text line to consist of at least three characters. In a further step, punctuations and separators are added if the correspondent parameters were set via `set_text_model_param`. Finally, if `TextModel` contains *'text_line_structure'* entries, the completed line is fitted to these structures.

`find_text` finds text which is roughly horizontally aligned in `Image`. `text_line_orientation` and `rotate_image` can be used to achieve this alignment.

Since memory is allocated for the text result during the call of `find_text` and during the following operations, the text result should be freed explicitly by the operator `clear_text_result` as soon as it is no longer used.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image.

▷ **TextModel** (input_control) . . . . . text_model ⇝ *handle*
Text model specifying the text to be segmented.

▷ **TextResultID** (output_control) . . . . . text_result ⇝ *handle*
Result of the segmentation.

Example

```
read_image (Image, 'numbers_scale')
create_text_model_reader ('auto', 'Document_Rej.omc', TextModel)
* Optionally specify text properties
set_text_model_param (TextModel, 'min_char_height', 20)
find_text (Image, TextModel, TextResultID)
* Return character regions and corresponding classification results
get_text_object (Characters, TextResultID, 'all_lines')
get_text_result (TextResultID, 'class', Class)
```

Result

If the parameters are valid, the operator `find_text` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`create_text_model_reader`, `set_text_model_param`, `text_line_orientation`, `text_line_slant`

Possible Successors

`get_text_result`, `get_text_object`

Module

OCR/OCV

get_text_model_param (: : TextModel, GenParamName : GenParamValue)

Query parameters of a text model.

`get_text_model_param` queries the parameters of `TextModel`. With the exception of *'ocr_classifier'*, all valid parameters can be queried. See `set_text_model_param` for a detailed description of the possible values for `GenParamName` and `GenParamValue`.

Parameters

▷ **TextModel** (input_control) ..... text_model ⇝ *handle*
Text model.

▷ **GenParamName** (input_control) ..... string ⇝ *string*
Parameters to be queried.
Default: 'min_contrast'
Suggested values: `GenParamName` ∈ {'add_fragments', 'dot_print', 'dot_print_max_dot_gap', 'dot_print_min_char_gap', 'dot_print_tight_char_spacing', 'eliminate_border_blobs', 'max_char_height', 'max_char_width', 'max_stroke_width', 'min_char_height', 'min_char_width', 'min_contrast', 'min_stroke_width', 'num_classes', 'polarity', 'return_punctuation', 'return_separators', 'return_whole_line', 'separate_touching_chars', 'text_line_separators', 'text_line_structure', 'text_line_structure_0', 'text_line_structure_1', 'text_line_structure_2', 'manual_add_fragments', 'manual_base_line_tolerance', 'manual_char_height', 'manual_char_width', 'manual_eliminate_border_blobs', 'manual_eliminate_horizontal_lines', 'manual_fragment_size_min', 'manual_is_dotprint', 'manual_is_imprinted', 'manual_max_line_num', 'manual_persistence', 'manual_polarity', 'manual_return_punctuation', 'manual_return_separators', 'manual_stroke_width', 'manual_text_line_structure', 'manual_text_line_structure_0', 'manual_text_line_structure_1', 'manual_text_line_structure_2', 'manual_uppercase_only'}

▷ **GenParamValue** (output_control) . . . . . string(-array) ⇝ *integer* / real / string
Values of Parameters.

Result

If the input parameters are set correctly, the operator `get_text_model_param` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`set_text_model_param`

Module

OCR/OCV

`get_text_object ( : Characters : TextResultID, ResultName : )`

Query an iconic value of a text segmentation result.

`get_text_object` queries the iconic result `ResultName` of `TextResultID` returned by `find_text`. The possible parameter values for `ResultName` differ, depending on the text model used within the text segmentation process of `find_text`.

In the following, first the possible parameter values for text models with `Mode` = *'auto'* are listed, and then those for text models with `Mode` = *'manual'*.

The following results can be queried:

- Result objects for text segmentation with text models with `Mode` = *'auto'*
  For each polarity, the text lines are independently sorted from top to bottom and left to right. The character within the text lines are sorted from left to right.

  'all_lines': Returns all characters in all segmented text lines.

  ['line', LineIndex]: Returns all characters within the text line specified by *LineIndex*. For example, *['line', 0]* returns the first line.

  ['element', Index]: Returns the character at position *Index*. For example, *['element', 0]* returns the first character.
- Result objects for text segmentation with text models with `Mode` = *'manual'*

  'manual_all_lines': Returns all characters in all segmented text lines. The text lines are sorted from top to bottom and left to right. The characters within the text lines are sorted from left to right.

  ['manual_line', Index]: Returns all characters within the text line specified by *Index* (e.g., *['manual_line',0]* to return the first line). The characters within the text line are sorted from left to right.

  If *'manual_persistence'* was activated for the text model used to create `TextResultID`, the following additional value can be queried:

  'manual_compensated_image': the enhanced image used for segmentation.

Parameters

▷ **Characters** (output_object) . . . . . object(-array) ⇝ *object*
Returned result.

▷ **TextResultID** (input_control) . . . . . text_result ⇝ *handle*
Text result.

▷ **ResultName** (input_control) . . . . . string(-array) ⇝ *string* / integer
Name of the result to be returned.
Default: 'all_lines'
List of values: `ResultName` ∈ {'all_lines', 'element', 'line', 'manual_all_lines', 'manual_line', 'manual_compensated_image'}

Example

```
read_image (Image, 'numbers_scale')
create_text_model_reader ('auto', 'Document_Rej.omc', TextModel)
* Optionally specify text properties
set_text_model_param (TextModel, 'min_char_height', 20)
find_text (Image, TextModel, TextResultID)
* Return character regions and corresponding classification results
get_text_object (Characters, TextResultID, 'all_lines')
get_text_result (TextResultID, 'class', Class)
```

Result

If the parameters are valid, the operator `get_text_object` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`find_text`

See also

`get_text_result`

Module

Foundation

get_text_result (: : TextResultID, ResultName : ResultValue)

Query a control value of a text segmentation result.

`get_text_result` queries the control result `ResultName` of `TextResultID` returned by `find_text`. The possible parameter values for `ResultName` differ, depending on the text model used within the text segmentation process of `find_text`.

In the following, first the possible parameter values for text models with `Mode` = *'auto'* are listed, and then those for text models with `Mode` = *'manual'*.

The following results can be queried:

- **Results for text segmentation with text models with Mode = 'auto'**
  For each polarity, the text lines are independently sorted from top to bottom and left to right. The character within the text lines are sorted from left to right.

  'num_lines': Number of lines found.

  'num_classes': Number of best classes stored for each character. Depending on the number of classes of the used classifier this value might be smaller than the one set in the text model, see `set_text_model_param`.

  'class': Classification results of all segmented characters with the OCR classifier contained in the corresponding text model. Note that if the classifier was trained with rejection class, `get_text_result` returns the second best result for characters that are classified as rejection class.

  ['class', n]: Similar to *'class'*, except that the class with the (n+1)-highest confidence is returned. For example, *['class', 0]* returns for each character the class with the highest confidence. Moreover, in contrast to *'class'*, the rejection class can be returned if the classifier contained in the corresponding text model was trained with rejection class.

['class_line', LineIndex]: Classification results of the characters within the text line specified by *LineIndex* with the OCR classifier contained in the corresponding text model. For example, *['class_line', 0]* returns the character classes within the first text line. Note that if the classifier was trained with rejection class, `get_text_result` returns the second best result for characters that are classified as rejection class.

['class_line', LineIndex, n]: Similar to *['class_line', LineIndex]*, except that the classes with the (n+1)-highest confidences are returned. For example, *['class_line', LineIndex, 0]* returns for each character in the text line specified by *LineIndex* the class with the highest confidence. Moreover, in contrast to *['class_line', LineIndex]*, the rejection class can be returned if the classifier contained in the corresponding text model was trained with rejection class.

['class_element', Index]: Classification result of the character at position *Index* with the OCR classifier contained in the corresponding text model. For example *['class_element', 0]* returns the *'num_classes'* best classes (sorted by confidence) of the first character.

'confidence': Returns the confidences of the classes for all segmented characters, see *'class'*.

['confidence', n]: Returns the confidences of the classes with the (n+1)-highest confidence, see *['class', n]*.

['confidence_line', LineIndex]: Returns the confidences of the classes of all characters within the text line specified by *LineIndex*, see *['class_line', LineIndex]*.

['confidence_line', LineIndex, n]: Returns the confidences of the classes with the (n+1)-highest confidence for all characters within the text line specified by *LineIndex*, see *['class_line', LineIndex, n]*.

['confidence_element', Index]: Returns the confidences corresponding to the classes of the character at position *Index*, see *['class_element', Index]*.

'polarity': Returns the polarity of all segmented characters.

['polarity_line', LineIndex]: Returns the polarity of the characters in the segmented text line specified by *LineIndex*. For example, *['polarity_line', 0]* returns the polarity within the first text line.

['polarity_element', Index]: Returns the polarity of the character at the position *Index*. For example *['polarity_char', 0]* returns the polarity of the first character.

- **Results for text segmentation with text models with Mode = 'manual'**

  'manual_num_lines': Number of lines found.

  If *'manual_persistence'* was activated for the text model used to create `TextResultID`, the following additional value can be queried:

  'manual_thresholds': The thresholds used for segmentation.

Parameters

▷ **`TextResultID`** (input_control) . . . . . text_result ⇝ *handle*
Text result.

▷ **`ResultName`** (input_control) . . . . . string(-array) ⇝ *string* / integer
Name of the result to be returned.
Default: 'class'
List of values: `ResultName` ∈ {'class', 'class_element', 'class_line', 'confidence', 'confidence_element', 'confidence_line', 'polarity', 'polarity_element', 'polarity_line', 'num_classes', 'num_lines', 'manual_num_lines', 'manual_thresholds'}

▷ **`ResultValue`** (output_control) . . . . . string(-array) ⇝ *integer* / real / string
Value of ResultName.

Example

```
read_image (Image, 'numbers_scale')
create_text_model_reader ('auto', 'Document_Rej.omc', TextModel)
* Optionally specify text properties
set_text_model_param (TextModel, 'min_char_height', 20)
find_text (Image, TextModel, TextResultID)
* Return character regions and corresponding classification results
get_text_object (Characters, TextResultID, 'all_lines')
get_text_result (TextResultID, 'class', Class)
```

Result

If the parameters are valid, the operator `get_text_result` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

find_text

See also

get_text_object

Module

Foundation

segment_characters (Region, Image : ImageForeground, RegionForeground : Method, EliminateLines, DotPrint, StrokeWidth, CharWidth, CharHeight, ThresholdOffset, Contrast : UsedThreshold)

Segments characters in a given region of an image.

This operator is used to segment characters in a given Region of an Image. The Region is only used to reduce the working area. The segmented characters are returned in RegionForeground.

Two different methods to detect the characters are supplied. All segmentation methods assume that the text is darker than the background. If this is not the case, please invert the image with invert_image.

The parameter Method determines the algorithm for text segmentation. The possible values are

'local_contrast_best' This method extracts text that differ locally from the background. Therefore, it is suited for images with inhomogeneous illumination. The enhancement of the text borders, leads to a more accurate determination of the outline of the text. Which is especially useful if the background is highly textured. The parameter Contrast defines the minimum contrast,i.e., the minimum gray value difference between symbols and background.

'local_auto_shape' The minimum contrast is estimated automatically such that the number of very small regions is reduced. This method is especially suitable for noisy images. The parameter ThresholdOffset can be used to adjust the threshold. Let $g(x, y)$ be the gray value at position (x, y) in the input Image. The threshold condition is determined by:

$g(x, y) \leq$ UsedThreshold $+$ ThresholdOffset.

Select EliminateLines if the extraction of characters is disturbed by lines that are horizontal or vertical with respect to the lines of text and set its value to *'true'*. The elimination is influenced by the maximum of CharWidth and the maximum of CharHeight. For further information see the description of these parameters.

DotPrint: Should be set to *'true'* if dot prints should be read, else to *'false'*.

StrokeWidth: Specifies the stroke width of the text. It is used to calculate internally used mask sizes to determine the characters. This mask sizes are also influenced through the parameters DotPrint, the average CharWidth, and the average CharHeight.

CharWidth: This can be a tuple with up to three values. The first value is the average width of a character. The second is the minimum width of a character and the third is the maximum width of a character. If the minimum is not set or equal -1, the operator automatically sets these value depending on the average CharWidth. The same is the case if the maximum is not set. Some examples:

[10] sets the average character width to 10, the minimum and maximum are calculated by the operator.

[10,-1,20] sets the average character width to 10, the minimum value is calculated by the system, and the maximum is set to 20.

[10,5,20] sets the average character width to 10, the minimum to 5, and the maximum to 20.

CharHeight: This can be a tuple with up to three values. The first value is the average height of a character. The second is the minimum height of a character and the third is the maximum height of a character. If the minimum is

not set or equal -1, the operator automatically sets these value depending on the average `CharHeight`. The same is the case if the maximum is not set. Some examples:

[10] sets the average character height to 10, the minimum and maximum are calculated by the operator.

[10,-1,20] sets the average character height to 10, the minimum value is calculated by the system, and the maximum is set to 20.

[10,5,20] this sets the average character height to 10, the minimum to 5, and the maximum to 20.

`ThresholdOffset`: This parameter can be used to adjust the threshold, which is used when the segmentation method *'local_auto_shape'* is chosen.

`Contrast`: Defines the minimum contrast between the text and the background. This parameter is used if the segmentation method *'local_contrast_best'* is selected.

`UsedThreshold`: After the execution, this parameter returns the threshold used to segment the characters.

`ImageForeground` returns the image that was internally used for the segmentation.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Area in the image where the text lines are located.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image.

▷ **ImageForeground** (output_object) . . . . . image(-array) ⇝ *object :* byte / uint2
Image used for the segmentation.

▷ **RegionForeground** (output_object) . . . . . singlechannelregion(-array) ⇝ *object*
Region of characters.

▷ **Method** (input_control) . . . . . string ⇝ *string*
Method to segment the characters.
Default: 'local_auto_shape'
List of values: `Method` ∈ {'local_contrast_best', 'local_auto_shape'}

▷ **EliminateLines** (input_control) . . . . . string ⇝ *string*
Eliminate horizontal and vertical lines?
Default: 'false'
List of values: `EliminateLines` ∈ {'true', 'false'}

▷ **DotPrint** (input_control) . . . . . string ⇝ *string*
Should dot print characters be detected?
Default: 'false'
List of values: `DotPrint` ∈ {'true', 'false'}

▷ **StrokeWidth** (input_control) . . . . . string ⇝ *string*
Stroke width of a character.
Default: 'medium'
List of values: `StrokeWidth` ∈ {'ultra_light', 'light', 'medium', 'bold'}

▷ **CharWidth** (input_control) . . . . . integer-array ⇝ *integer*
Width of a character.
Default: 25
Value range: 1 ≤ CharWidth

▷ **CharHeight** (input_control) . . . . . integer-array ⇝ *integer*
Height of a character.
Default: 25
Value range: 1 ≤ CharHeight

▷ **ThresholdOffset** (input_control) . . . . . integer ⇝ *integer*
Value to adjust the segmentation.
Default: 0

▷ **Contrast** (input_control) . . . . . integer ⇝ *integer*
Minimum gray value difference between text and background.
Default: 10

▷ **UsedThreshold** (output_control) . . . . . integer(-array) ⇝ *integer*
Threshold used to segment the characters.

Example

```
for Index := 1 to 5 by 1
  read_image (Image, 'dot_print_rotated/dot_print_rotated_'+Index$'02d')
  text_line_orientation (Image, Image, 50, rad(-30), rad(30), \
                         OrientationAngle)
  rotate_image (Image, ImageRotate, deg(-OrientationAngle), 'constant')
  segment_characters (ImageRotate, ImageRotate, ImageForeground, \
                      RegionForeground, 'local_auto_shape', 'false', \
                      'false', 'medium', 25, 25, 0, 10, UsedThreshold)
endfor
```

Result

If the input parameters are set correctly, the operator `segment_characters` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`text_line_orientation`

Possible Successors

`select_characters`, `connection`

Alternatives

`threshold`

Module

Foundation

`select_characters` (Region : RegionCharacters : DotPrint, StrokeWidth, CharWidth, CharHeight, Punctuation, DiacriticMarks, PartitionMethod, PartitionLines, FragmentDistance, ConnectFragments, ClutterSizeMax, StopAfter :)

Selects characters from a given region.

`select_characters` selects from a given `Region` the areas which might be characters and returns them in `RegionCharacters`. This is done by using features like `StrokeWidth`, `DotPrint`, the size of the characters and some more. The given `Region` should be united, else every `Region` is processed separately. Thus do not call `connection` before calling `select_characters`, because then fragments or dots would not be connected to a character. If you have more than one region with text, you can of course handle them without merging them. The `Region` for `select_characters` typically comes from `segment_characters` but also any other segmentation operators can be used.

The process of the selection can be partitioned into four parts. All steps are influenced by the parameters `StrokeWidth`, `CharHeight`, and `CharWidth`. If you loose small objects like dots, adapt the minimum `CharWidth` and the minimum `CharHeight`. But some parameters affect the result of a certain step in particular. A closer description follows below. With the parameter `StopAfter` you can terminate after a specified step.

In the first step, *'step1_select_candidates'*, `CharWidth` and the `CharHeight` are used to select the candidates. The result of this step is also affected by `ClutterSizeMax`.

In the next step, *'step2_partition_characters'*, the parameter `PartitionMethod` and the parameter `PartitionLines` influence the result.

Step three, *'step3_connect_fragments'*, uses the parameters ConnectFragments and DotPrint. If dot-printed characters have to be detected and some dots are not connected to the character, there are two ways to overcome this problem: You can increase the FragmentDistance and/or decrease the StrokeWidth.

In the last step, *'step4_select_characters'*, the result is affected by the parameters DiacriticMarks and Punctuation.

DotPrint: Should be set to *'true'* if dot prints should be read, else to *'false'*.

StrokeWidth: Specifies the stroke width of the text. It is used to calculate internally used mask sizes to determine the characters. This mask sizes are also influenced through the parameters DotPrint, the average CharWidth, and the average CharHeight.

CharWidth: This can be a tuple with up to three values. The first value is the average width of a character. The second is the minimum width of a character and the third is the maximum width of a character. If the minimum is not set or equal -1, the operator automatically sets these value depending on average CharWidth. The same is the case if the maximum is not set. Some examples:

[10] sets the average character width to 10, the minimum and maximum are calculated by the operator.

[10,-1,20] sets the average character width to 10, the minimum value is calculated by the system, and the maximum is set to 20.

[10,5,20] sets the average character width to 10, the minimum to 5, and the maximum to 20.

CharHeight: This can be a tuple with up to three values. The first value is the average height of a character. The second is the minimum height of a character and the third is the maximum height of a character. If the minimum is not set or equal -1, the operator automatically sets these value depending on average CharHeight. The same is the case if the maximum is not set. Some examples:

[10] sets the average character height to 10, the minimum and maximum are calculated by the operator.

[10,-1,20] sets the average character height to 10 the minimum value is calculated by the system and the maximum is set to 20.

[10,5,20] this sets the average character height to 10, the minimum to 5 and the maximum to 20.

Punctuation: Set this parameter to *'true'* if the operator also has to detect punctuation marks (e.g., .,:'‘"), otherwise they will be suppressed.

DiacriticMarks: Set this parameter to *'true'* if the text in your application contains diacritic marks (e.g., â,é,ö), or to *'false'* to suppress them.

PartitionMethod: If neighboring characters are printed close to each other, they may be partly merged. With this parameter you can specify the method to partition such characters. The possible values are *'none'*, which means no partitioning is performed. *'fixed_width'* means that the partitioning assumes a constant character width. If the width of the extracted region is well above the average CharWidth, the region is split into parts that have the given average CharWidth. The partitioning starts at the left border of the region. *'variable_width'* means that the characters are partitioned at the position where they have the thinnest connection. This method can be selected for characters that are printed with a variable-width font or if many consecutive characters are extracted as one symbol. It could be helpful to call text_line_slant and/or use text_line_orientation before calling select_characters.

PartitionLines: If some text lines or some characters of different text lines are connected, set this parameter to *'true'*.

FragmentDistance: This parameter influences the connection of character fragments. If too much is connected, set the parameter to *'narrow'* or *'medium'*. In the case that more fragments should be connected, set the parameter to *'medium'* or *'wide'*. The connection is also influenced by the maximum of CharWidth and CharHeight. See also ConnectFragments.

ConnectFragments: Set this parameter to *'true'* if the extracted symbols are fragmented, i.e., if a symbol is not extracted as one region but broken up into several parts. See also FragmentDistance and StopAfter in the step *'step3_connect_fragments'*.

ClutterSizeMax: If the extracted characters contain clutter, i.e., small regions near the actual symbols, increase this value. If parts of the symbols are missing, decrease this value.

StopAfter: Use this parameter in the case the operator does not produce the desired results. By modifying this value the operator stops after the execution of the selected step and provides the corresponding results. To end on completion, set StopAfter to *'completion'*.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Region of text lines in which to select the characters.

▷ **RegionCharacters** (output_object) . . . . . region(-array) ⇝ *object*
Selected characters.

▷ **DotPrint** (input_control) . . . . . string ⇝ *string*
Should dot print characters be detected?
Default: 'false'
List of values: `DotPrint` ∈ {'true', 'false'}

▷ **StrokeWidth** (input_control) . . . . . string ⇝ *string*
Stroke width of a character.
Default: 'medium'
List of values: `StrokeWidth` ∈ {'ultra_light', 'light', 'medium', 'bold'}

▷ **CharWidth** (input_control) . . . . . integer-array ⇝ *integer*
Width of a character.
Default: 25
Value range: 1 ≤ CharWidth

▷ **CharHeight** (input_control) . . . . . integer-array ⇝ *integer*
Height of a character.
Default: 25
Value range: 1 ≤ CharHeight

▷ **Punctuation** (input_control) . . . . . string ⇝ *string*
Add punctuation?
Default: 'false'
List of values: `Punctuation` ∈ {'true', 'false'}

▷ **DiacriticMarks** (input_control) . . . . . string ⇝ *string*
Exist diacritic marks?
Default: 'false'
List of values: `DiacriticMarks` ∈ {'true', 'false'}

▷ **PartitionMethod** (input_control) . . . . . string ⇝ *string*
Method to partition neighbored characters.
Default: 'none'
List of values: `PartitionMethod` ∈ {'none', 'fixed_width', 'variable_width'}

▷ **PartitionLines** (input_control) . . . . . string ⇝ *string*
Should lines be partitioned?
Default: 'false'
List of values: `PartitionLines` ∈ {'true', 'false'}

▷ **FragmentDistance** (input_control) . . . . . string ⇝ *string*
Distance of fragments.
Default: 'medium'
List of values: `FragmentDistance` ∈ {'narrow', 'medium', 'wide'}

▷ **ConnectFragments** (input_control) . . . . . string ⇝ *string*
Connect fragments?
Default: 'false'
List of values: `ConnectFragments` ∈ {'true', 'false'}

▷ **ClutterSizeMax** (input_control) . . . . . integer ⇝ *integer*
Maximum size of clutter.
Default: 0
Value range: 0 ≤ ClutterSizeMax

▷ **StopAfter** (input_control) . . . . . string ⇝ *string*
Stop execution after this step.
Default: 'completion'
List of values: `StopAfter` ∈ {'step1_select_candidates', 'step2_partition_characters', 'step3_connect_fragments', 'step4_select_characters', 'completion'}

Example

```
for Index := 1 to 5 by 1
  read_image (Image, 'dot_print_rotated/dot_print_rotated_'+Index$'02d')
  text_line_orientation (Image, Image, 50, rad(-30), rad(30), \
                         OrientationAngle)
  rotate_image (Image, ImageRotate, deg(-OrientationAngle), 'constant')
  segment_characters (ImageRotate, ImageRotate, ImageForeground, \
                      RegionForeground, 'local_auto_shape', 'false', \
                      'false', 'medium', 25, 25, 0, 10, UsedThreshold)
  select_characters (RegionForeground, RegionCharacters, 'true', \
                     'ultra_light', [60,1,100], [60,1,100], 'false', \
                     'false', 'none', 'true', 'wide', 'true', 0, 'completion')
endfor
```

Result

If the input parameters are set correctly, the operator `select_characters` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`segment_characters`, `text_line_slant`

Possible Successors

`do_ocr_single_class_mlp`, `do_ocr_multi_class_mlp`

Alternatives

`connection`

Module

Foundation

set_text_model_param (: : TextModel, GenParamName, GenParamValue :)

Set parameters of a text model.

`set_text_model_param` sets parameters of a text model. The list of allowed parameter values for `GenParamName` differs, depending on which `Mode` was set when creating the text model with `create_text_model_reader`. In the following, first the parameter values for text models with `Mode` = *'auto'* are listed, and then those for text models with `Mode` = *'manual'*.

The name and value of a parameter must be given in `GenParamName` and `GenParamValue`. The following values are possible:

- **Parameters of text models with Mode = 'auto'**
  - **Segmentation behavior**

    'min_contrast': The minimal contrast the characters have to their surrounding background.
    Value range: integer or float value between *1* and *255* for byte images and between *1* and *65.535* for uint2 images.
    Default: *15*

    'polarity': *'dark_on_light'* if the text to be segmented is darker than its background, *'light_on_dark'* if the text to be segmented is lighter than its background, and *'both'* if both kinds of text are to be segmented.
    List of values: *'dark_on_light'*, *'light_on_dark'*, *'both'*
    Default: *'both'*

'eliminate_border_blobs': *'true'* if regions that are touching the border of the image domain should be discarded, otherwise *'false'*.
List of values: *'true'*, *'false'*
Default: *'false'*

'add_fragments': *'true'* if fragments, such as the dot on the 'i', should be added to the segmented characters, otherwise *'false'*. Be aware, that this can cause noise to be added to the segmented characters.
List of values: *'true'*, *'false'*
Default: *'true'*

'separate_touching_chars': Controls the handling of pairs or small groups of neighboring characters that are segmented as one single region. When selecting *'standard'* or *'enhanced'*, such regions are detected and separated into two or more single characters. While the *'enhanced'* method yields more accurate results, the *'standard'* method is less complex and thus faster. If *'separate_touching_chars'* is set to *'false'*, no separation of touching characters is performed.
Remark: If *'enhanced'* is selected, the file *find_text_support.hotc* from the ocr subdirectory of the root directory of the HALCON installation is needed. It is also possible to place this file in the current working directory.
List of values: *'false'*, *'standard'*, *'enhanced'*
Default: *'standard'*

– **Character size**

'min_char_height': The minimal height of the characters in pixel. If text of arbitrary height is to be segmented, *'auto'* may be passed. Note that *'min_char_height'* refers to characters only. The height of punctuation marks or separators is not restricted by *'min_char_height'*.
Default: *'auto'*
Restriction: integer or float value greater or equal to *1*.

'max_char_height': The maximal height of the characters in pixel. If text of arbitrary height is to be segmented, *'auto'* may be passed. Note that *'max_char_height'* refers to characters only. The height of punctuation marks or separators is not restricted by *'max_char_height'*.
Default: *'auto'*
Restriction: integer or float value greater or equal to *1*.

'min_char_width': The minimal width of the characters in pixel. If text of arbitrary width is to be segmented, *'auto'* may be passed. Note that *'min_char_width'* refers to characters only. The width of punctuation marks or separators is not restricted by *'min_char_width'*.
Default: *'auto'*
Restriction: integer or float value greater or equal to *1*.

'max_char_width': The maximal width of the characters in pixel. If text of arbitrary width is to be segmented, *'auto'* may be passed. Note that *'max_char_width'* refers to characters only. The width of punctuation marks or separators is not restricted by *'max_char_width'*.
Default: *'auto'*
Restriction: integer or float value greater or equal to *1*.

'min_stroke_width': The minimal stroke width of the characters in pixel. If the minimal stroke width is to be estimated within the text segmentation process automatically, *'auto'* may be passed. Note that *'min_stroke_width'* refers to characters only. The stroke width of punctuation marks or separators is not restricted by *'min_stroke_width'*.
Default: *'auto'*
Restriction: integer or float value greater or equal to *1*.

'max_stroke_width': The maximal stroke width of the characters in pixel. If the maximal stroke width is to be estimated within the text segmentation process automatically, *'auto'* may be passed. Note that *'max_stroke_width'* refers to characters only. The stroke width of punctuation marks or separators is not restricted by *'max_stroke_width'*.
Default: *'auto'*
Restriction: integer or float value greater or equal to *1*.

– **Special characters**

'return_punctuation': *'true'* if small punctuation marks that lie close to the base line of the corresponding text line (e.g., dots or commas) are to be returned. *'false'* if no such punctuations should be returned.
List of values: *'true'*, *'false'*
Default: *'true'*

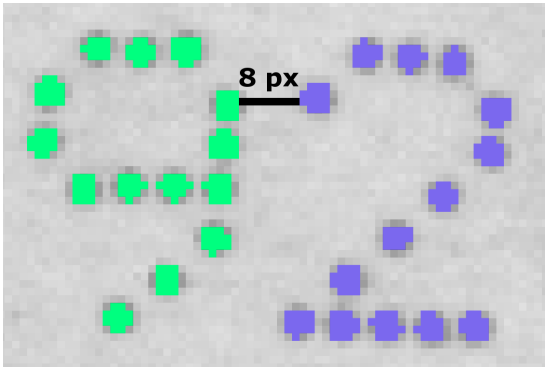
'return_separators': *'true'* if separators such as a minus or the equality sign should be returned as well. *'false'* if no separators should be returned.
List of values: *'true'*, *'false'*
Default: *'true'*

– **Handling of dot prints**

'dot_print': *'true'* if the text to be segmented contains dot printed characters, otherwise *'false'*.
List of values: *'true'*, *'false'*
Default: *'false'*

'dot_print_tight_char_spacing': *'true'* if the gap between adjacent characters is smaller than the largest gap between two dots within a single character, otherwise *'false'*. If *'dot_print'* is set to *'false'* this parameter does not have any effect. In cases where the minimal gap size between characters is exactly known, *'dot_print_min_char_gap'* can be set instead. In this case the value of *'dot_print_tight_char_spacing'* is ignored.
List of values: *'true'*, *'false'*
Default: *'false'*

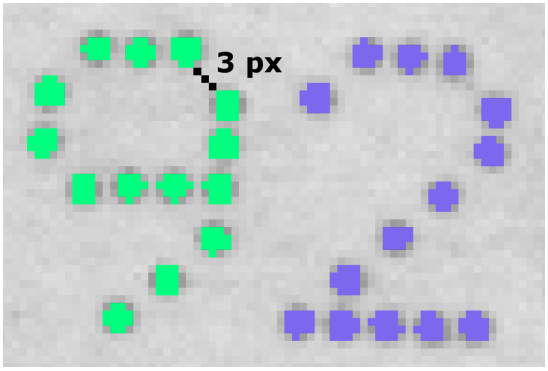
'dot_print_min_char_gap': The minimal gap size between two characters in pixel. This parameter can be used to improve the text result in cases where the minimal gap size between characters is smaller than the maximal gap size between dots within characters. If the minimal character gap size is not known or is bigger than the maximal dot gap size, *'auto'* may be passed. If *'dot_print'* is set to *'false'* this parameter does not have any effect. In cases where the minimal gap size between characters is not known but the characters are printed close to each other, *'dot_print_tight_char_spacing'* might be used instead.



Here, the minimal gap size between characters is 8 pixel.

Default: *'auto'*
Restriction: integer or float value greater or equal to *0*.

'dot_print_max_dot_gap': The maximal gap size between two dots within a character in pixel. If arbitrary dot printed characters are to be segmented, *'auto'* may be passed. If *'dot_print'* is set to *'false'* this parameter does not have any effect. In cases where the maximal dot gap size is larger than or equal to the minimal gap size between characters, *'dot_print_tight_char_spacing'* or *'dot_print_min_char_gap'* should be set accordingly. Setting *'dot_print_max_dot_gap'* can reduce the runtime of `find_text` significantly.



Here, the maximal gap size between dots is 3 pixel.

Default: *'auto'*
Restriction: integer or float value greater or equal to *1*.

– **Line structures**

'text_line_structure': To simplify the search for specific structures (e.g., dates or serial numbers) within the segmented text, it is possible to define text line structures. For each text line the distances between the characters are calculated, and based on these distances, the text line is divided into text blocks. Short characters such as '.', '_' and '-' are ignored in this process and treated as spaces. Furthermore, it is possible to define user specific separators which are also ignored. See the description of *'text_line_separators'* for details. It is then tested if any of the user defined text line structures fit the resulting text blocks.
For example, if the text to be found is a date with two characters for month, day, and year the structure would be '2 2 2'. If the year may consist of two or four characters, the structure would be '2 2 2-4', indicating that the last character block consists of two to four characters. It is possible to provide more than one structure to match by appending an index to the parameter name, e.g., *'text_line_structure_0'*, *'text_line_structure_1'*. If *'text_line_structure'* is set to an empty string *' '*, the text to be found may have any structure.
Please observe, that every text line structure which is found, is saved as a unique text line within the text result. Hence, when calling `get_text_object`, a 'line' then refers to a valid text line structure. If the whole text line containing the text line structure is to be returned instead, it is possible to set *'return_whole_line'* accordingly.
Default: *' '*

'text_line_separators': A string containing the list of characters which are to be ignored in the process of finding text line structures, see *'text_line_structure'* for further details. Please note, user specific separators need to be valid characters within the used OCR classifier. For example, if *':'* and '\' are to be ignored, *':\\'* should be passed. Please observe, that '\' escapes any special symbol to treat it as a literal, and hence '\\' needs to be passed to use '\' as a separator.
Suggested values: *'/'*, *':'*, *':\\'*, *':\\/:'*
Default: *' '*

'return_whole_line': *'false'* if only the segmented text line structures are to be returned as text lines. *'true'* if each whole text line containing a text line structure is to be returned in text lines.
List of values: *'true'*, *'false'*
Default: *'false'*

– **OCR classifier**

'ocr_classifier': The OCR classifier used within `find_text` for text segmentation and classification. An initial classifier is set when the text model is created. See `create_text_model_reader` for more information about the required OCR Classifier.

'num_classes': The number of best classes to be stored for each character (e.g., if *'num_classes'* is set to 2, `find_text` returns the classification results with the highest and second highest confidence). If *'num_classes'* exceeds the number of classes of the classifier stored in the text model, *'num_classes'* is decreased accordingly. The actual number of classes can be queried by `get_text_result`. For classifiers with rejection class, *'num_classes'* should be at least 2 in order to be able to use the second best result if a character is classified as rejection class.
Default: *2*
Restriction: integer or float value greater or equal to *1*.

- **Parameters of text models with Mode = 'manual'**

'manual_char_height': Height of the characters in pixel. Refers to an uppercase character.
Default: *30*

'manual_char_width': Width of the characters in pixel. Refers to an uppercase character.
Default: *20*

'manual_stroke_width': Stroke width of the characters in pixel.
Default: *4.0*

'manual_base_line_tolerance': Maximum base line deviation of the characters (in percent of *'manual_char_height'*).
Default: *0.15*

'manual_polarity': *'dark_on_light'* if the text to be segmented is darker than its background, otherwise *'light_on_dark'*.
Default: *'dark_on_light'*

'manual_uppercase_only': *'true'* if the text to be segmented contains uppercase characters or numbers only, otherwise *'false'*.
Default: *'false'*

'manual_is_dotprint': *'true'* if the text to be segmented is a dotprint, otherwise *'false'*.
Default: *'false'*

'manual_is_imprinted': *'true'* if the text to be segmented suffers of local changes of polarity due to reflections, otherwise *'false'*. **Default:** *'false'*

'manual_eliminate_horizontal_lines': *'true'* if there are longer horizontal structures close to the text to be segmented, otherwise *'false'*. **Default:** *'false'*

'manual_eliminate_border_blobs': *'true'* if regions that are touching the border of the image domain should be discarded, otherwise *'false'*.
Default: *'false'*

'manual_max_line_num': Maximum number of lines to be found. Zero or negative values indicate no limitation. Setting *'manual_max_line_num'* to a low value may strongly improve the runtime of `find_text`.
Default: no limitation

'manual_return_punctuation': *'true'* if punctuation marks (e.g., dots or comma) should be added to the segmented characters.
Default: *'true'*

'manual_return_separators': *'true'* if separators such as a minus or the equality sign should be added to the segmented characters.
Default: *'true'*

'manual_add_fragments': *'true'* if fragments, such as the dot on the 'i', should be added to the segmented characters. Be aware, that this can cause noise to be added to the segmented characters.
Default: *'true'*

'manual_fragment_size_min': minimum area of fragment regions that are added if *'manual_add_fragments'* is set to *'true'*.
Default: *1*

'manual_text_line_structure': specifies the structure of the text to be found to reduce the search space and to avoid false positives. The structure is a string that contains the number of characters for every character block and spaces between these character blocks. For example, if the text to be found is a date with two characters for month, day, and year the structure would be '2 2 2'. If the year may also consist of four characters the structure would be '2 2 2-4', indicating that the last character block consists of two to four characters. It is possible to provide more than one structure to match by appending an index to the parameter name, e.g., *'manual_text_line_structure_0'*, *'manual_text_line_structure_1'*. If *'manual_text_line_structure'* is set to an empty string ' ', the text to be found may have any structure.
Default: ' '

'manual_persistence': *'true'* if selected intermediate results should be kept with the output result of `find_text`.

Parameters

▷ **`TextModel`** (input_control) . . . . . text_model ⇝ *handle*
Text model.

▷ **`GenParamName`** (input_control) . . . . . string(-array) ⇝ *string*
Names of the parameters to be set.
Default: 'min_contrast'
Suggested values: `GenParamName` ∈ {'add_fragments', 'dot_print', 'dot_print_max_dot_gap', 'dot_print_min_char_gap', 'dot_print_tight_char_spacing', 'eliminate_border_blobs', 'max_char_height', 'max_char_width', 'max_stroke_width', 'min_char_height', 'min_char_width', 'min_contrast', 'min_stroke_width', 'num_classes', 'ocr_classifier', 'polarity', 'return_punctuation', 'return_separators', 'return_whole_line', 'separate_touching_chars', 'text_line_separators', 'text_line_structure', 'text_line_structure_0', 'text_line_structure_1', 'text_line_structure_2', 'manual_add_fragments', 'manual_base_line_tolerance', 'manual_char_height', 'manual_char_width', 'manual_eliminate_border_blobs', 'manual_eliminate_horizontal_lines', 'manual_fragment_size_min', 'manual_is_dotprint', 'manual_is_imprinted', 'manual_max_line_num', 'manual_persistence', 'manual_polarity', 'manual_return_punctuation', 'manual_return_separators', 'manual_stroke_width', 'manual_text_line_structure', 'manual_text_line_structure_0', 'manual_text_line_structure_1', 'manual_text_line_structure_2', 'manual_uppercase_only'}

▷ **`GenParamValue`** (input_control) . . . . . string(-array) ⇝ *integer* / real / string
Values of the parameters to be set.
Default: 10
Suggested values: `GenParamValue` ∈ {'true', 'false', 'dark_on_light', 'light_on_dark', 'both', 'auto',

'standard', 'enhanced'}

Example

```
read_image (Image, 'numbers_scale')
create_text_model_reader ('auto', 'Document_Rej.omc', TextModel)
* Optionally specify text properties
set_text_model_param (TextModel, 'min_char_height', 20)
find_text (Image, TextModel, TextResultID)
* Return character regions and corresponding classification results
get_text_object (Characters, TextResultID, 'all_lines')
get_text_result (TextResultID, 'class', Class)
```

Result

If the input parameters are set correctly, the operator `set_text_model_param` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- TextModel

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_text_model_reader`

Possible Successors

`find_text`

See also

`get_text_model_param`

Module

OCR/OCV

text_line_orientation (Region, Image : : CharHeight, OrientationFrom, OrientationTo : OrientationAngle)

Determines the orientation of a text line or paragraph.

`text_line_orientation` determines the orientation of a single text line or a paragraph in relation to the horizontal image axis. If the orientation of a single text line should be determined, the range for the `OrientationFrom` and `OrientationTo` should be in the interval from -pi/4 to pi/4.

The parameter `Region` specifies the area of the image in which the text lines are located. The `Region` is only used to reduce the working area. To determine the slant, the gray values inside that area are used. The text lines are segmented by the operator `text_line_orientation` itself. If more than one region is passed, the numerical values of the orientation angle are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

`CharHeight` specifies the approximately height of the existing text lines in the region `Region`. It's assumed, that the text lines are darker than the background.

The search area can be restricted by the parameters `OrientationFrom` and `OrientationTo`, whereby also the runtime of the operator is influenced.

With the calculated angle OrientationAngle and operators like affine_trans_image, the region Region of the image Image can be rotated such, that the text lines lie horizontally in the image. This may simplify the character segmentation for OCR applications.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Area of text lines.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object* : byte / uint2
  Input image.
- ▷ **CharHeight** (input_control) ..... integer ⇝ *integer*
  Height of the text lines.
  Default: 25
  Value range: 1 ≤ CharHeight
  Restriction: CharHeight >= 1
- ▷ **OrientationFrom** (input_control) ..... angle.rad ⇝ *real*
  Minimum rotation of the text lines.
  Default: -0.523599
  Value range: -3.141593 ≤ OrientationFrom ≤ 3.141593
  Restriction: OrientationFrom <= OrientationTo
- ▷ **OrientationTo** (input_control) ..... angle.rad ⇝ *real*
  Maximum rotation of the text lines.
  Default: 0.523599
  Value range: -3.141593 ≤ OrientationTo ≤ 3.141593
  Restriction: OrientationFrom <= OrientationTo && OrientationTo < OrientationFrom + pi
- ▷ **OrientationAngle** (output_control) ..... angle.rad(-array) ⇝ *real*
  Calculated rotation angle of the text lines.

Example

```
read_image(Image,'letters')
text_line_orientation(Image,Image,50,rad(-80),rad(80),OrientationAngle)
rotate_image(Image,ImageRotate,-OrientationAngle/rad(180)*180,'constant')
```

Result

If the input parameters are set correctly, the operator text_line_orientation returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

rotate_image, affine_trans_image, affine_trans_image_size

Module

Foundation

text_line_slant (Region, Image : : CharHeight, SlantFrom, SlantTo : SlantAngle)

Determines the slant of characters of a text line or paragraph.

text_line_slant determines the slant of a single text line or a paragraph.

The parameter Region specifies the area of the image in which the text lines are located. The Region is only used to reduce the working area. To determine the slant, the gray values inside that area are used. The text lines are segmented by the operator text_line_slant itself. If more than one region is passed, the numerical values of

the orientation angle are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

CharHeight specifies the approximately high of the existing text lines in the region Region. It's assumed, that the text lines are darker than the background.

The search area can be restricted by the parameters SlantFrom and SlantTo, whereby also the runtime of the operator is influenced.

With the calculated slant angle SlantAngle and operators for affine transformations, the slant can be removed from the characters. This may simplify the character separation for OCR applications. To work correctly all characters of a region should have nearly the same slant.

Parameters

- ▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
  Area of text lines.
- ▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image.
- ▷ **CharHeight** (input_control) . . . . . integer ⇝ *integer*
  Height of the text lines.
  Default: 25
  Value range: 1 ≤ CharHeight
- ▷ **SlantFrom** (input_control) . . . . . angle.rad ⇝ *real*
  Minimum slant of the characters
  Default: -0.523599
  Value range: -0.785398 ≤ SlantFrom ≤ 0.785398
  Restriction: SlantFrom <= SlantTo
- ▷ **SlantTo** (input_control) . . . . . angle.rad ⇝ *real*
  Maximum slant of the characters
  Default: 0.523599
  Value range: -0.785398 ≤ SlantTo ≤ 0.785398
  Restriction: SlantFrom <= SlantTo
- ▷ **SlantAngle** (output_control) . . . . . angle.rad(-array) ⇝ *real*
  Calculated slant of the characters in the region

Example

```
hom_mat2d_identity(HomMat2DIdentity)
read_image(Image,'dot_print_slanted')
* correct slant
text_line_slant(Image,Image,50,rad(-45),rad(45),SlantAngle)
hom_mat2d_slant(HomMat2DIdentity,-SlantAngle,'x',0,0,HomMat2DSlant)
affine_trans_image(Image,Image,HomMat2DSlant,'constant','true')
```

Result

If the input parameters are set correctly, the operator text_line_slant returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

hom_mat2d_slant, affine_trans_image, affine_trans_image_size

Module

Foundation

21.7 Support Vector Machines

clear_ocr_class_svm (: : OCRHandle :)

Clear an SVM-based OCR classifier.

clear_ocr_class_svm clears the OCR classifier given by OCRHandle and frees all memory required for the classifier. After calling clear_ocr_class_svm, the classifier can no longer be used. The handle OCRHandle becomes invalid.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_svm(-array) ⇝ *handle*
Handle of the OCR classifier.

Result

If OCRHandle is valid the operator clear_ocr_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

do_ocr_single_class_svm, do_ocr_multi_class_svm

See also

create_ocr_class_svm, read_ocr_class_svm, write_ocr_class_svm, trainf_ocr_class_svm

Module

OCR/OCV

create_ocr_class_svm (: : WidthCharacter, HeightCharacter, Interpolation, Features, Characters, KernelType, KernelParam, Nu, Mode, Preprocessing, NumComponents : OCRHandle)

Create an OCR classifier using a support vector machine.

create_ocr_class_svm creates an OCR classifier that uses a support vector machine (SVM). The handle of the OCR classifier is returned in OCRHandle.

For a description on how an SVM works, see create_class_svm. create_ocr_class_svm creates an SVM for classification with the classification mode given by Mode. The length of the feature vector of the SVM (NumFeatures in create_class_svm) is determined from the features that are used for the OCR, which are passed in Features. The features are described below. The kernel is parametrized with KernelType, KernelParam and Nu like in create_class_svm. The number of classes of the SVM (NumClasses in create_class_svm) is determined from the names of the characters to be used in the OCR, which are passed in Characters. As described with create_class_svm, the parameters Preprocessing and NumComponents can be used to specify a preprocessing of the data (i.e., the feature vectors). For the sake of numerical stability, Preprocessing can typically be set to *'normalization'*. In order to speed up classification time, *'principal_components'* or *'canonical_variates'* can be used, as the number of input features can be significantly reduced without deterioration of the recognition rate.

The features to be used for the classification are determined by Features. Features can contain a tuple of feature names. Each of these feature names results in one or more features to be calculated for the classifier. Some of the feature names compute gray value features (e.g., *'pixel_invar'*). Because a classifier requires a constant number of features (input variables), a character to be classified is transformed to a standard size, which is determined by WidthCharacter and HeightCharacter. The interpolation to be used for the transformation is determined by Interpolation. It has the same meaning as in affine_trans_image. The interpolation should be chosen such that no aliasing effects occur in the transformation. For most applications, Interpolation = *'constant'* should be used. It should be noted that the size of the transformed character is not chosen too large, because the generalization properties of the classifier may become bad for large sizes. In particular, for large sizes small segmentation errors will have a large influence on the computed features if gray value features are used. This happens because segmentation errors will change the smallest enclosing rectangle of the regions, thus the character is zoomed differently than the characters in the training set. In most applications, sizes between 6×8 and 10×14 should be used.

The parameter Features can contain the following feature names for the classification of the characters.

'default' *'ratio'* and *'pixel_invar'* are selected.

'pixel' Gray values of the character (WidthCharacter × HeightCharacter features).

'pixel_invar' Gray values of the character with maximum scaling of the gray values (WidthCharacter × HeightCharacter features).

'pixel_binary' Region of the character as a binary image zoomed to a size of WidthCharacter × HeightCharacter (WidthCharacter × HeightCharacter features).

'gradient_8dir' Gradients are computed on the character image. The gradient directions are discretized into 8 directions. The amplitude image is decomposed into 8 channels according to these discretized directions. 25 samples on a 5×5 grid are extracted from each channel. These samples are used as features (200 features).

'projection_horizontal' Horizontal projection of the gray values (see gray_projections, HeightCharacter features).

'projection_horizontal_invar' Maximally scaled horizontal projection of the gray values (HeightCharacter features).

'projection_vertical' Vertical projection of the gray values (see gray_projections, WidthCharacter features).

'projection_vertical_invar' Maximally scaled vertical projection of the gray values (WidthCharacter features).

'ratio' Aspect ratio of the character (see height_width_ratio, 1 feature).

'anisometry' Anisometry of the character (see eccentricity, 1 feature).

'width' Width of the character before scaling the character to the standard size (not scale-invariant, see height_width_ratio, 1 feature).

'height' Height of the character before scaling the character to the standard size (not scale-invariant, see height_width_ratio, 1 feature).

'zoom_factor' Difference in size between the character and the values of WidthCharacter and HeightCharacter (not scale-invariant, 1 feature).

'foreground' Fraction of pixels in the foreground (1 feature).

'foreground_grid_9' Fraction of pixels in the foreground in a 3×3 grid within the smallest enclosing rectangle of the character (9 features).

'foreground_grid_16' Fraction of pixels in the foreground in a 4×4 grid within the smallest enclosing rectangle of the character (16 features).

'compactness' Compactness of the character (see compactness, 1 feature).

'convexity' Convexity of the character (see convexity, 1 feature).

'moments_region_2nd_invar' Normalized 2nd moments of the character (see moments_region_2nd_invar, 3 features).

'moments_region_2nd_rel_invar' Normalized 2nd relative moments of the character (see moments_region_2nd_rel_invar, 2 features).

'moments_region_3rd_invar' Normalized 3rd moments of the character (see moments_region_3rd_invar, 4 features).

'moments_central' Normalized central moments of the character (see `moments_region_central`, 4 features).

'moments_gray_plane' Normalized gray value moments and the angle of the gray value plane (see `moments_gray_plane`, 4 features).

'phi' Orientation (angle) of the character (see `elliptic_axis`, 1 feature).

'num_connect' Number of connected components (see `connect_and_holes`, 1 feature).

'num_holes' Number of holes (see `connect_and_holes`, 1 feature).

'cooc' Values of the binary cooccurrence matrix (see `gen_cooc_matrix`, 12 features).

'num_runs' Number of runs in the region normalized by the height (1 feature).

'chord_histo' Frequency of the runs per row (not scale-invariant, `HeightCharacter` features).

After the classifier has been created, it is trained using `trainf_ocr_class_svm`. After this, the classifier can be saved using `write_ocr_class_svm`. Alternatively, the classifier can be used immediately after training to classify characters using `do_ocr_single_class_svm` or `do_ocr_multi_class_svm`.

A comparison of SVM and the multi-layer perceptron (MLP) (see `create_ocr_class_mlp`) typically shows that SVMs are generally faster at training, especially for huge training sets, and achieve slightly better recognition rates than MLPs. The MLP is faster at classification and should therefore be preferred in time critical applications. Please note that this guideline assumes optimal tuning of the parameters.

Parameters

▷ **`WidthCharacter`** (input_control) . . . integer ⇝ *integer*
Width of the rectangle to which the gray values of the segmented character are zoomed.
Default: 8
Suggested values: `WidthCharacter` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20}
Value range: 4 ≤ `WidthCharacter` ≤ 20

▷ **`HeightCharacter`** (input_control) . . . integer ⇝ *integer*
Height of the rectangle to which the gray values of the segmented character are zoomed.
Default: 10
Suggested values: `HeightCharacter` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20}
Value range: 4 ≤ `HeightCharacter` ≤ 20

▷ **`Interpolation`** (input_control) . . . string ⇝ *string*
Interpolation mode for the zooming of the characters.
Default: 'constant'
List of values: `Interpolation` ∈ {'nearest_neighbor', 'bilinear', 'bicubic', 'constant', 'weighted'}

▷ **`Features`** (input_control) . . . string(-array) ⇝ *string*
Features to be used for classification.
Default: 'default'
List of values: `Features` ∈ {'default', 'pixel', 'pixel_invar', 'pixel_binary', 'gradient_8dir', 'projection_horizontal', 'projection_horizontal_invar', 'projection_vertical', 'projection_vertical_invar', 'ratio', 'anisometry', 'width', 'height', 'zoom_factor', 'foreground', 'foreground_grid_9', 'foreground_grid_16', 'compactness', 'convexity', 'moments_region_2nd_invar', 'moments_region_2nd_rel_invar', 'moments_region_3rd_invar', 'moments_central', 'moments_gray_plane', 'phi', 'num_connect', 'num_holes', 'cooc', 'num_runs', 'chord_histo'}

▷ **`Characters`** (input_control) . . . string-array ⇝ *string*
All characters of the character set to be read.
Default: ['0','1','2','3','4','5','6','7','8','9']

▷ **`KernelType`** (input_control) . . . string ⇝ *string*
The kernel type.
Default: 'rbf'
List of values: `KernelType` ∈ {'linear', 'rbf', 'polynomial_inhomogeneous', 'polynomial_homogeneous'}

▷ **`KernelParam`** (input_control) . . . real ⇝ *real*
Additional parameter for the kernel function.
Default: 0.02
Suggested values: `KernelParam` ∈ {0.01, 0.02, 0.05, 0.1, 0.5}

▷ **Nu** (input_control) ..... real ⇝ *real*
Regularization constant of the SVM.
Default: 0.05
Suggested values: Nu ∈ {0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3}
Restriction: Nu > 0.0 && Nu < 1.0

▷ **Mode** (input_control) ..... string ⇝ *string*
The mode of the SVM.
Default: 'one-versus-one'
List of values: Mode ∈ {'one-versus-all', 'one-versus-one'}

▷ **Preprocessing** (input_control) ..... string ⇝ *string*
Type of preprocessing used to transform the feature vectors.
Default: 'normalization'
List of values: Preprocessing ∈ {'none', 'normalization', 'principal_components', 'canonical_variates'}

▷ **NumComponents** (input_control) ..... integer ⇝ *integer*
Preprocessing parameter: Number of transformed features (ignored for Preprocessing = *'none'* and Preprocessing = *'normalization'*).
Default: 10
Suggested values: NumComponents ∈ {1, 2, 3, 4, 5, 8, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Restriction: NumComponents >= 1

▷ **OCRHandle** (output_control) ..... ocr_svm ⇝ *handle*
Handle of the OCR classifier.

Example

```
read_image (Image, 'letters')
* Segment the image.
binary_threshold(Image,&Region, 'otsu', 'dark', &UsedThreshold);
dilation_circle (Region, RegionDilation, 3.5)
connection (RegionDilation, ConnectedRegions)
intersection (ConnectedRegions, Region, RegionIntersection)
sort_region (RegionIntersection, Characters, 'character', 'true', 'row')
* Generate the training file.
count_obj (Characters, Number)
Classes := []
for J := 0 to 25 by 1
    Classes := [Classes,gen_tuple_const(20,chr(ord('a')+J))]
endfor
Classes := [Classes,gen_tuple_const(20,'.')]
write_ocr_trainf (Characters, Image, Classes, 'letters.trf')
* Generate and train the classifier.
read_ocr_trainf_names ('letters.trf', CharacterNames, CharacterCount)
create_ocr_class_svm (8, 10, 'constant', 'default', CharacterNames, \
                      'rbf', 0.01, 0.01, 'one-versus-all', \
                      'principal_components', 10, OCRHandle)
trainf_ocr_class_svm (OCRHandle, 'letters.trf', 0.001, 'default')
* Re-classify the characters in the image.
do_ocr_multi_class_svm (Characters, Image, OCRHandle, Class)
```

Result

If the parameters are valid the operator create_ocr_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

trainf_ocr_class_svm

Alternatives

create_ocr_class_mlp

See also

do_ocr_single_class_svm, do_ocr_multi_class_svm, clear_ocr_class_svm, create_class_svm, train_class_svm, classify_class_svm

Module

OCR/OCV

deserialize_ocr_class_svm (: : SerializedItemHandle : OCRHandle)

Deserialize a serialized SVM-based OCR classifier.

deserialize_ocr_class_svm deserializes a SVM-based OCR classifier, that was serialized by serialize_ocr_class_svm (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized OCR classifier is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created OCR classifier with the handle OCRHandle.

Parameters

- ▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
  Handle of the serialized item.
- ▷ **OCRHandle** (output_control) ..... ocr_svm ⇝ *handle*
  Handle of the OCR classifier.

Result

If the parameters are valid, the operator deserialize_ocr_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_ocr_class_svm

Possible Successors

do_ocr_single_class_svm, do_ocr_multi_class_svm

See also

create_ocr_class_svm, write_ocr_class_svm, read_class_svm, write_class_svm, serialize_ocr_class_svm

Module

OCR/OCV

do_ocr_multi_class_svm (Character, Image : : OCRHandle : Class)

Classify multiple characters with an SVM-based OCR classifier.

do_ocr_multi_class_svm computes the best class for each of the characters given by the regions Character and the gray values Image with the SVM-based OCR classifier OCRHandle and returns the classes in Class. In contrast to do_ocr_single_class_svm, do_ocr_multi_class_svm can classify multiple characters in one call, and therefore typically is faster than a loop that uses do_ocr_single_class_svm

to classify single characters. However, do_ocr_multi_class_svm can only return the best class of each character. Before calling do_ocr_multi_class_svm, the classifier must be trained with trainf_ocr_class_svm.

Parameters

- ▷ **Character** (input_object) ..... region(-array) ⇝ *object*
  Characters to be recognized.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object* : byte / uint2
  Gray values of the characters.
- ▷ **OCRHandle** (input_control) ..... ocr_svm ⇝ *handle*
  Handle of the OCR classifier.
- ▷ **Class** (output_control) ..... string(-array) ⇝ *string*
  Result of classifying the characters with the SVM.

Result

If the parameters are valid the operator do_ocr_multi_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

trainf_ocr_class_svm, read_ocr_class_svm

Alternatives

do_ocr_single_class_svm

See also

create_ocr_class_svm, classify_class_svm

Module

OCR/OCV

do_ocr_single_class_svm (Character, Image : : OCRHandle, Num : Class)

Classify a single character with an SVM-based OCR classifier.

do_ocr_single_class_svm computes the best Num classes of the character given by the region Character and the gray values Image with the OCR classifier OCRHandle and returns the classes in Class. Because multiple classes may be returned by do_ocr_single_class_svm, Character may only contain a single region (a single character). If multiple characters should be classified in a single call, do_ocr_multi_class_svm must be used. Before calling do_ocr_single_class_svm, the classifier must be trained with trainf_ocr_class_svm.

Parameters

- ▷ **Character** (input_object) ..... region ⇝ *object*
  Character to be recognized.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object* : byte / uint2
  Gray values of the character.
- ▷ **OCRHandle** (input_control) ..... ocr_svm ⇝ *handle*
  Handle of the OCR classifier.
- ▷ **Num** (input_control) ..... integer-array ⇝ *integer*
  Number of best classes to determine.
  Default: 1
  Suggested values: Num ∈ {1, 2, 3, 4, 5}

▷ **Class** (output_control) . . . . . string(-array) ⇝ *string*
Result of classifying the character with the SVM.

Result

If the parameters are valid the operator do_ocr_single_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_svm, read_ocr_class_svm

Alternatives

do_ocr_multi_class_svm

See also

create_ocr_class_svm, classify_class_svm

Module

OCR/OCV

do_ocr_word_svm (Character, Image : : OCRHandle, Expression, NumAlternatives, NumCorrections : Class, Word, Score)

Classify a related group of characters with an OCR classifier.

do_ocr_word_svm works like do_ocr_multi_class_svm insofar as it computes the best class for each of the characters given by the regions Character and the gray values Image with the OCR classifier OCRHandle, and returns the results in Class.

In contrast to do_ocr_multi_class_svm, do_ocr_word_svm treats the group of characters as an entity which yields a Word by concatenating the class names for each character region. This allows to restrict the allowed classification results on a textual level by specifying an Expression describing the expected word.

The Expression may restrict the word to belong to a predefined lexicon created using create_lexicon or import_lexicon, by specifying the name of the lexicon in angular brackets as in *'<mylexicon>'*. If the Expression is of any other form, it is interpreted as a regular expression with the same syntax as specified for tuple_regexp_match. Note that you will usually want to use an expression of the form '^...$' when using variable quantifiers like '*', to ensure that the entire word is used in the expression. Also note that in contrast to tuple_regexp_match, do_ocr_word_svm does **not** support passing extra options in an expression tuple.

If the word derived from the best class for each character does not match the Expression, do_ocr_word_svm attempts to correct it by considering the NumAlternatives best classes for each character. The alternatives used are identical to those returned by do_ocr_single_class_svm for a single character. It does so by testing all possible corrections for which the classification result is changed for at most NumCorrections character regions. Note that NumAlternatives and NumCorrections affect the complexity of the algorithm, so that in some cases internal restrictions are made. See the section 'Complexity' below for further information.

In case the Expression is a lexicon and the above procedure did not yield a result, the most similar word in the lexicon is returned as long as it requires less than NumCorrections edit operations for the correction (see suggest_lexicon).

The resulting word is graded by a Score between 0.0 (no correction found) and 1.0 (original word correct). The Score is lowered by adding a penalty according to the number of corrected characters and another (minor) penalty depending on how many better classes have been ignored in order to match the Expression:

$$\mathtt{Score} = 1.0 - \mathtt{PenaltyCorrections} - \mathtt{PenaltyAlternatives}$$

$$\begin{aligned} \mathtt{PenaltyCorrections} &= \alpha * \mathtt{num\_corr} \\ \mathtt{PenaltyAlternatives} &= \alpha * \beta * \mathtt{num\_alt} \end{aligned}$$

with num_corr being the actual number of applied corrections and num_alt the total number of discarded alternatives.

$$\alpha = \frac{1}{\texttt{NumCorrections} + 1}$$
$$\beta = \frac{1}{\texttt{NumCorrections} * (\texttt{NumAlternatives} - 1) + 2}$$

Parameters

▷ **Character** (input_object) . . . . . region(-array) ⇝ *object*
Characters to be recognized.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Gray values of the characters.

▷ **OCRHandle** (input_control) . . . . . ocr_svm ⇝ *handle*
Handle of the OCR classifier.

▷ **Expression** (input_control) . . . . . string ⇝ *string*
Expression describing the allowed word structure.

▷ **NumAlternatives** (input_control) . . . . . integer ⇝ *integer*
Number of classes per character considered for the internal word correction.
Default: 3
Suggested values: NumAlternatives ∈ {3, 4, 5}
Value range: 1 ≤ NumAlternatives

▷ **NumCorrections** (input_control) . . . . . integer ⇝ *integer*
Maximum number of corrected characters.
Default: 2
Suggested values: NumCorrections ∈ {1, 2, 3, 4, 5}
Value range: 0 ≤ NumCorrections

▷ **Class** (output_control) . . . . . string(-array) ⇝ *string*
Result of classifying the characters with the SVM.
Number of elements: Class == Character

▷ **Word** (output_control) . . . . . string ⇝ *string*
Word text after classification and correction.

▷ **Score** (output_control) . . . . . real ⇝ *real*
Measure of similarity between corrected word and uncorrected classification results.

Complexity

The complexity of checking all possible corrections is of magnitude $O((n * a)^{min(c,n)})$, where a is the number of alternatives, n is the number of character regions, and c is the number of allowed corrections. However, to guard against a near-infinite loop in case of large n, c is internally clipped to 5, 3, or 1 if $a * n >= 30$, 60, or 90, respectively.

Result

If the parameters are valid, the operator do_ocr_word_svm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_svm, read_ocr_class_svm

Alternatives

do_ocr_multi_class_svm

See also

create_ocr_class_svm, classify_class_svm

Module

OCR/OCV

get_features_ocr_class_svm (Character : : OCRHandle, Transform : Features)

Compute the features of a character.

get_features_ocr_class_svm computes the features of the character given by Character with the OCR classifier OCRHandle and returns them in Features. In contrast to do_ocr_single_class_svm and do_ocr_multi_class_svm, the character is passed as a single image object. Hence, before calling get_features_ocr_class_svm, reduce_domain must typically be called. The parameter Transform determines whether the feature transformation specified with Preprocessing in create_ocr_class_svm for the classifier should be applied (Transform = *'true'*) or whether the untransformed features should be returned (Transform = *'false'*). get_features_ocr_class_svm can be used to inspect the features that are used for the classification.

Parameters

- ▷ **Character** (input_object) ..... singlechannelimage ⇝ *object* : byte / uint2
  Input character.
- ▷ **OCRHandle** (input_control) ..... ocr_svm ⇝ *handle*
  Handle of the OCR classifier.
- ▷ **Transform** (input_control) ..... string ⇝ *string*
  Should the feature vector be transformed with the preprocessing?
  Default: 'true'
  List of values: Transform ∈ {'true', 'false'}
- ▷ **Features** (output_control) ..... real-array ⇝ *real*
  Feature vector of the character.

Result

If the parameters are valid the operator get_features_ocr_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_svm

See also

create_ocr_class_svm

Module

OCR/OCV

get_params_ocr_class_svm (: : OCRHandle : WidthCharacter, HeightCharacter, Interpolation, Features, Characters, KernelType, KernelParam, Nu, Mode, Preprocessing, NumComponents)

Return the parameters of an OCR classifier.

get_params_ocr_class_svm returns the parameters of an OCR classifier that were specified when the classifier was created with create_ocr_class_svm. This is particularly useful if the classifier was read with read_ocr_class_svm. The output of get_params_ocr_class_svm can, for example, be used to check whether a character to be read is contained in the classifier. For a description of the parameters, see create_ocr_class_svm.

Parameters

- ▷ **OCRHandle** (input_control) ..... ocr_svm ⇝ *handle*
  Handle of the OCR classifier.
- ▷ **WidthCharacter** (output_control) ..... integer ⇝ *integer*
  Width of the rectangle to which the gray values of the segmented character are zoomed.
- ▷ **HeightCharacter** (output_control) ..... integer ⇝ *integer*
  Height of the rectangle to which the gray values of the segmented character are zoomed.
- ▷ **Interpolation** (output_control) ..... string ⇝ *string*
  Interpolation mode for the zooming of the characters.
- ▷ **Features** (output_control) ..... string(-array) ⇝ *string*
  Features to be used for classification.
- ▷ **Characters** (output_control) ..... string-array ⇝ *string*
  Characters of the character set to be read.
- ▷ **KernelType** (output_control) ..... string ⇝ *string*
  The kernel type.
- ▷ **KernelParam** (output_control) ..... real ⇝ *real*
  Additional parameters for the kernel function.
- ▷ **Nu** (output_control) ..... real ⇝ *real*
  Regularization constant of the SVM.
- ▷ **Mode** (output_control) ..... string ⇝ *string*
  The mode of the SVM.
- ▷ **Preprocessing** (output_control) ..... string ⇝ *string*
  Type of preprocessing used to transform the feature vectors.
- ▷ **NumComponents** (output_control) ..... integer ⇝ *integer*
  Preprocessing parameter: Number of transformed features (ignored for `Preprocessing` = *'none'* and `Preprocessing` = *'normalization'*).

Result

If the parameters are valid the operator `get_params_ocr_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_ocr_class_svm`, `read_ocr_class_svm`

Possible Successors

`do_ocr_single_class_svm`, `do_ocr_multi_class_svm`

See also

`trainf_ocr_class_svm`, `get_params_class_svm`

Module

OCR/OCV

get_prep_info_ocr_class_svm (: : OCRHandle, TrainingFile, Preprocessing : InformationCont, CumInformationCont)

Compute the information content of the preprocessed feature vectors of an SVM-based OCR classifier.

`get_prep_info_ocr_class_svm` computes the information content of the training vectors that have been transformed with the preprocessing given by `Preprocessing`. `Preprocessing` can be set to *'principal_components'* or *'canonical_variates'*. The OCR classifier `OCRHandle` must have been created with `create_ocr_class_svm`. The preprocessing methods are described with `create_class_svm`.

The information content is derived from the variations of the transformed components of the feature vector, i.e., it is computed solely based on the training data, independent of any error rate on the training data. The information content is computed for all relevant components of the transformed feature vectors (NumFeatures for *'principal_components'* and $\min(\texttt{NumClasses} - 1, \texttt{NumFeatures})$ for *'canonical_variates'*, see create_class_svm), and is returned in InformationCont as a number between 0 and 1. To convert the information content into a percentage, it simply needs to be multiplied by 100. The cumulative information content of the first n components is returned in the n-th component of CumInformationCont, i.e., CumInformationCont contains the sums of the first n elements of InformationCont. To use get_prep_info_ocr_class_svm, a sufficient number of samples must be stored in the training files given by TrainingFile (see write_ocr_trainf).

InformationCont and CumInformationCont can be used to decide how many components of the transformed feature vectors contain relevant information. An often used criterion is to require that the transformed data must represent x% (e.g., 90%) of the total data. This can be decided easily from the first value of CumInformationCont that lies above x%. The number thus obtained can be used as the value for NumComponents in a new call to create_ocr_class_svm. The call to get_prep_info_ocr_class_svm already requires the creation of a classifier, and hence the setting of NumComponents in create_ocr_class_svm to an initial value. However, if get_prep_info_ocr_class_svm is called it is typically not known how many components are relevant, and hence how to set NumComponents in this call. Therefore, the following two-step approach should typically be used to select NumComponents: In a first step, a classifier with the maximum number for NumComponents is created (NumFeatures for *'principal_components'* and $\min(\texttt{NumClasses}-1, \texttt{NumFeatures})$ for *'canonical_variates'*). Then, the training samples are saved in a training file using write_ocr_trainf. Subsequently, get_prep_info_ocr_class_svm is used to determine the information content of the components, and with this NumComponents. After this, a new classifier with the desired number of components is created, and the classifier is trained with trainf_ocr_class_svm.

Parameters

- ▷ **OCRHandle** (input_control) ..... ocr_svm ⇝ *handle*
  Handle of the OCR classifier.
- ▷ **TrainingFile** (input_control) ..... filename.read(-array) ⇝ *string*
  Names of the training files.
  Default: 'ocr.trf'
  File extension: .trf, .otr
- ▷ **Preprocessing** (input_control) ..... string ⇝ *string*
  Type of preprocessing used to transform the feature vectors.
  Default: 'principal_components'
  List of values: Preprocessing ∈ {'principal_components', 'canonical_variates'}
- ▷ **InformationCont** (output_control) ..... real-array ⇝ *real*
  Relative information content of the transformed feature vectors.
- ▷ **CumInformationCont** (output_control) ..... real-array ⇝ *real*
  Cumulative information content of the transformed feature vectors.

Example

```
* Create the initial OCR classifier.
read_ocr_trainf_names ('ocr.trf', CharacterNames, CharacterCount)
create_ocr_class_svm (8, 10, 'constant', 'default', CharacterNames, \
                      'rbf', 0.01, 0.01, 'one-versus-one', \
                      'principal_components', 81, OCRHandle)
* Get the information content of the transformed feature vectors.
get_prep_info_ocr_class_svm (OCRHandle, 'ocr.trf', 'principal_components', \
                             InformationCont, CumInformationCont)
* Determine the number of transformed components.
* NumComp = [...]
* Create the final OCR classifier.
create_ocr_class_svm (8, 10, 'constant', 'default', CharacterNames, \
                      'rbf', 0.01, 0.01,'one-versus-one', \
                      'principal_components', NumComp, OCRHandle)
* Train the final classifier.
```

```
trainf_ocr_class_svm (OCRHandle, 'ocr.trf', 0.001, 'default')
write_ocr_class_svm (OCRHandle, 'ocr.osc')
```

Result

If the parameters are valid the operator `get_prep_info_ocr_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

`get_prep_info_ocr_class_svm` may return the error 9211 (Matrix is not positive definite) if `Preprocessing` = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_ocr_class_svm`, `write_ocr_trainf`, `append_ocr_trainf`, `write_ocr_trainf_image`

Possible Successors

`clear_ocr_class_svm`, `create_ocr_class_svm`

Module

OCR/OCV

get_support_vector_num_ocr_class_svm

```
get_support_vector_num_ocr_class_svm ( : : OCRHandle : NumSupportVectors, NumSVPerSVM )
```

Return the number of support vectors of an OCR classifier.

`get_support_vector_num_ocr_class_svm` returns in `NumSupportVectors` the number of support vectors that are stored in the support vector machine (SVM) given by `OCRHandle`. `get_support_vector_num_ocr_class_svm` should be called before the labels of individual support vectors are read out with `get_support_vector_ocr_class_svm`, e.g., for visualizing which of the training data become a SV (see `get_support_vector_ocr_class_svm`). The number of SVs in each classifier is listed in `NumSVPerSVM`. The reason that its sum differs from the number obtained in `NumSupportVectors` is that SV evaluations are reused throughout different binary sub-SVMs.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_svm ⇝ *handle*
OCR handle.

▷ **NumSupportVectors** (output_control) ..... integer ⇝ *integer*
Total number of support vectors.

▷ **NumSVPerSVM** (output_control) ..... integer-array ⇝ *integer*
Number of SV of each sub-SVM.

Result

If the parameters are valid the operator `get_sample_num_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`trainf_ocr_class_svm`

Possible Successors

get_support_vector_ocr_class_svm

See also

create_ocr_class_svm

Module

OCR/OCV

get_support_vector_ocr_class_svm (: : OCRHandle, IndexSupportVector : Index)

Return the index of a support vector from a trained OCR classifier that is based on support vector machines.

The operator get_support_vector_ocr_class_svm maps support vectors of a trained SVM-based OCR classifier (given in OCRHandle) to the original training data set. The index of the SV is specified with IndexSupportVector. The index is counted from 0, i.e., IndexSupportVector must be a number between 0 and NumSupportVectors − 1, where NumSupportVectors can be determined with get_support_vector_num_ocr_class_svm. The index of this SV in the training data is returned in Index. get_support_vector_ocr_class_svm can, for example, be used to visualize the support vectors. To do so, the train file that has been used to train the SVM must be read with read_ocr_trainf. The value returned in Index must be incremented by *1* and can then be used to select the support vectors with select_obj from the training characters. If more than one train file has been used in trainf_ocr_class_svm Index behaves as if all train files had been merged into one train file with concat_ocr_trainf.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_svm ⇝ *handle*
OCR handle.

▷ **IndexSupportVector** (input_control) ..... integer-array ⇝ *integer*
Number of stored support vectors.

▷ **Index** (output_control) ..... real ⇝ *real*
Index of the support vector in the training set.

Result

If the parameters are valid the operator get_support_vector_ocr_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_svm, get_support_vector_num_ocr_class_svm

See also

create_ocr_class_svm, read_ocr_trainf, append_ocr_trainf, concat_ocr_trainf

Module

OCR/OCV

read_ocr_class_svm (: : FileName : OCRHandle)

Read a SVM-based OCR classifier from a file.

read_ocr_class_svm reads an OCR classifier that has been stored with write_ocr_class_svm. Since the training of an OCR classifier can consume a relatively long time, the classifier is typically trained in an offline process and written to a file with write_ocr_class_svm. In the online process the classifier is read

with read_ocr_class_svm and subsequently used for classification with do_ocr_single_class_svm or do_ocr_multi_class_svm.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .osc, .fnt

▷ **OCRHandle** (output_control) . . . . . ocr_svm ⇝ *handle*
Handle of the OCR classifier.

Result

If the parameters are valid the operator read_ocr_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

do_ocr_single_class_svm, do_ocr_multi_class_svm

See also

create_ocr_class_svm, write_ocr_class_svm, read_class_svm, write_class_svm

Module

OCR/OCV

reduce_ocr_class_svm (: : OCRHandle, Method, MinRemainingSV, MaxError : OCRHandleReduced)

Approximate a trained SVM-based OCR classifier by a reduced SVM.

reduce_ocr_class_svm reduces the classification time of an SVM based OCR classifier OCRHandle by returning a reduced copy of it in OCRHandleReduced. The parameters Method, MinRemainingSV and MaxError have the same meaning as in reduce_class_svm and are described there. Please note that classification time can also be significantly reduced with a preprocessing step in create_ocr_class_svm, which possibly introduces less errors.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_svm ⇝ *handle*
Original handle of SVM-based OCR-classifier.

▷ **Method** (input_control) . . . . . string ⇝ *string*
Type of postprocessing to reduce number of SVs.
Default: 'bottom_up'
List of values: Method ∈ {'bottom_up'}

▷ **MinRemainingSV** (input_control) . . . . . integer ⇝ *integer*
Minimum number of remaining SVs.
Default: 2
Suggested values: MinRemainingSV ∈ {2, 3, 4, 5, 7, 10, 15, 20, 30, 50}
Restriction: MinRemainingSV >= 2

▷ **MaxError** (input_control) . . . . . real ⇝ *real*
Maximum allowed error of reduction.
Default: 0.001
Suggested values: MaxError ∈ {0.0001, 0.0002, 0.0005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05}
Restriction: MaxError > 0.0

▷ **OCRHandleReduced** (output_control) . . . . . ocr_svm ⇝ *handle*
SVMHandle of reduced OCR classifier.

Result

If the parameters are valid the operator reduce_ocr_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_svm, get_support_vector_num_ocr_class_svm

Possible Successors

do_ocr_single_class_svm, do_ocr_multi_class_svm, get_support_vector_ocr_class_svm, get_support_vector_num_ocr_class_svm

See also

create_ocr_class_svm

References

Bernhard Schölkopf, Alexander J.Smola: "Learning with Kernels"; The MIT Press, London; 1999.

Module

OCR/OCV

select_feature_set_trainf_svm (: : TrainingFile, FeatureList, SelectionMethod, Width, Height, GenParamName, GenParamValue : OCRHandle, FeatureSet, Score)

Selects an optimal combination of features to classify OCR data.

select_feature_set_trainf_svm selects an optimal combination of features, to classify the data given in the training file TrainingFile with a Support Vector Machine (SVM), for details see create_ocr_class_svm.

Possible features are all OCR features listed and explained in create_ocr_class_svm. All candidates which should be tested can be specified in FeatureList. A subset of these features is returned as selected features in FeatureSet.

select_feature_set_trainf_svm is specialized on OCR problems and only supports the features in the list mentioned before. In order to use other features, please use the more general operator select_feature_set_svm.

The selection method SelectionMethod is either a greedy search *'greedy'* (iteratively add the feature with highest gain) or the dynamically oscillating search *'greedy_oscillating'* (add the feature with highest gain and test then if any of the already added features can be left out without great loss). The method *'greedy'* is generally preferable, since it is faster. Only in cases when a large training set is available the method *'greedy_oscillating'* might return better results.

The optimization criterion is the classification rate of a two-fold cross-validation of the training data. The best achieved value is returned in Score.

The parameters *'nu'* and *'gamma'* for the SVM that is used to classify can be set to *'auto'* by using the parameters GenParamName and GenParamValue. If they are set to *'auto'*, the estimated optimal *'nu'* and/or *'gamma'* is estimated. The automatic estimation of *'nu'* and *'gamma'* can take a substantial amount of time (up to days, depending on the data set and the number of features). Alternatively, a certain value for both can be set the same way. An explanation of the parameters *'nu'* and *'gamma'* as the kernel parameter of the radial basis function (RBF) kernel can be found in create_class_svm.

Attention

This operator may take considerable time, depending on the size of the data set in the training file, and the number of features.

Please note, that this operator should not be called, if only a small set of training data is available. Due to the risk of overfitting the operator `select_feature_set_trainf_svm` may deliver a classifier with a very high score. However, the classifier may perform poorly when tested.

Parameters

▷ **`TrainingFile`** (input_control) . . . . . filename.read(-array) ⇝ *string*
Names of the training files.
Default: ”
File extension: `.trf`, `.otr`

▷ **`FeatureList`** (input_control) . . . . . string(-array) ⇝ *string*
List of features that should be considered for selection.
Default: [’zoom_factor’,’ratio’,’width’,’height’,’foreground’,’foreground_grid_9’,’foreground_grid_16’, ’anisometry’,’compactness’,’convexity’,’moments_region_2nd_invar’,’moments_region_2nd_rel_invar’, ’moments_region_3rd_invar’,’moments_central’,’phi’,’num_connect’,’num_holes’,’projection_horizontal’, ’projection_vertical’,’projection_horizontal_invar’,’projection_vertical_invar’,’chord_histo’,’num_runs’, ’pixel’,’pixel_invar’,’pixel_binary’,’gradient_8dir’,’cooc’,’moments_gray_plane’]
List of values: `FeatureList` ∈ {’default’, ’zoom_factor’, ’ratio’, ’width’, ’height’, ’foreground’, ’foreground_grid_9’, ’foreground_grid_16’, ’anisometry’, ’compactness’, ’convexity’, ’moments_region_2nd_invar’, ’moments_region_2nd_rel_invar’, ’moments_region_3rd_invar’, ’moments_central’, ’phi’, ’num_connect’, ’num_holes’, ’projection_horizontal’, ’projection_vertical’, ’projection_horizontal_invar’, ’projection_vertical_invar’, ’chord_histo’, ’num_runs’, ’pixel’, ’pixel_invar’, ’pixel_binary’, ’gradient_8dir’, ’cooc’, ’moments_gray_plane’}

▷ **`SelectionMethod`** (input_control) . . . . . string ⇝ *string*
Method to perform the selection.
Default: ’greedy’
List of values: `SelectionMethod` ∈ {’greedy’, ’greedy_oscillating’}

▷ **`Width`** (input_control) . . . . . integer ⇝ *integer*
Width of the rectangle to which the gray values of the segmented character are zoomed.
Default: 15

▷ **`Height`** (input_control) . . . . . integer ⇝ *integer*
Height of the rectangle to which the gray values of the segmented character are zoomed.
Default: 16

▷ **`GenParamName`** (input_control) . . . . . string-array ⇝ *string*
Names of generic parameters to configure the selection process and the classifier.
Default: []
List of values: `GenParamName` ∈ {’nu’, ’gamma’}

▷ **`GenParamValue`** (input_control) . . . . . number-array ⇝ *real* / integer / string
Values of generic parameters to configure the selection process and the classifier.
Default: []
Suggested values: `GenParamValue` ∈ {’auto’, ’0.1’, ’0.3’}

▷ **`OCRHandle`** (output_control) . . . . . ocr_svm ⇝ *handle*
Trained OCR-SVM Classifier.

▷ **`FeatureSet`** (output_control) . . . . . string-array ⇝ *string*
Selected feature set, contains only entries from `FeatureList`.

▷ **`Score`** (output_control) . . . . . real-array ⇝ *real*
Achieved score using tow-fold cross-validation.

Result

If the parameters are valid, the operator `select_feature_set_trainf_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Alternatives

select_feature_set_trainf_mlp, select_feature_set_trainf_knn, select_feature_set_trainf_mlp_protected

See also

select_feature_set_trainf_svm_protected, select_feature_set_svm

Module

OCR/OCV

select_feature_set_trainf_svm_protected (: : TrainingFile, Password, FeatureList, SelectionMethod, Width, Height, GenParamName, GenParamValue : OCRHandle, FeatureSet, Score)

Select an optimal combination of features to classify OCR data from a (protected) training file.

select_feature_set_trainf_svm_protected selects an optimal combination of features to classify the data given in the training files TrainingFile with a support vector machine (SVM). Its functionality corresponds to the functionality of select_feature_set_trainf_svm, with the addition that select_feature_set_trainf_svm_protected can process unprotected and protected training files. Protected training files can be used only with the correct user password Password. If the number of passwords Password equals 1, then every input file TrainingFile is checked with that password, otherwise the number of passwords has to be equal to the number of input files and the input file at position n is checked with the password at position n. For unprotected training files the passwords are ignored.

For a more detailed description of the operator's functionality see select_feature_set_trainf_svm. The concept of protecting OCR training data in HALCON is described in protect_ocr_trainf.

Attention

This operator may take considerable time, depending on the size of the data set in the training file, and the number of features.

Please note, that this operator should not be called, if only a small set of training data is available. Due to the risk of overfitting the operator select_feature_set_trainf_svm_protected may deliver a classifier with a very high score. However, the classifier may perform poorly when tested.

Parameters

▷ **TrainingFile** (input_control) ..... filename.read(-array) ⇝ *string*
Names of the training files.
Default: ''
File extension: .trf, .otr

▷ **Password** (input_control) ..... string(-array) ⇝ *string*
Passwords for protected training files.

▷ **FeatureList** (input_control) ..... string(-array) ⇝ *string*
List of features that should be considered for selection.
Default: ['zoom_factor','ratio','width','height','foreground','foreground_grid_9','foreground_grid_16', 'anisometry','compactness','convexity','moments_region_2nd_invar','moments_region_2nd_rel_invar', 'moments_region_3rd_invar','moments_central','phi','num_connect','num_holes','projection_horizontal', 'projection_vertical','projection_horizontal_invar','projection_vertical_invar','chord_histo','num_runs', 'pixel','pixel_invar','pixel_binary','gradient_8dir','cooc','moments_gray_plane']
List of values: FeatureList ∈ {'default', 'zoom_factor', 'ratio', 'width', 'height', 'foreground', 'foreground_grid_9', 'foreground_grid_16', 'anisometry', 'compactness', 'convexity', 'moments_region_2nd_invar', 'moments_region_2nd_rel_invar', 'moments_region_3rd_invar', 'moments_central', 'phi', 'num_connect', 'num_holes', 'projection_horizontal', 'projection_vertical', 'projection_horizontal_invar', 'projection_vertical_invar', 'chord_histo', 'num_runs', 'pixel', 'pixel_invar', 'pixel_binary', 'gradient_8dir', 'cooc', 'moments_gray_plane'}

▷ **SelectionMethod** (input_control) ..... string ⇝ *string*
Method to perform the selection.
Default: 'greedy'
List of values: SelectionMethod ∈ {'greedy', 'greedy_oscillating'}

▷ **Width** (input_control) . . . . . integer ⇝ *integer*
Width of the rectangle to which the gray values of the segmented character are zoomed.
Default: 15

▷ **Height** (input_control) . . . . . integer ⇝ *integer*
Height of the rectangle to which the gray values of the segmented character are zoomed.
Default: 16

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of generic parameters to configure the selection process and the classifier.
Default: []
List of values: GenParamName ∈ {'nu', 'gamma'}

▷ **GenParamValue** (input_control) . . . . . number-array ⇝ *real* / integer / string
Values of generic parameters to configure the selection process and the classifier.
Default: []
Suggested values: GenParamValue ∈ {'auto', '0.1', '0.3'}

▷ **OCRHandle** (output_control) . . . . . ocr_svm ⇝ *handle*
Trained OCR-SVM Classifier.

▷ **FeatureSet** (output_control) . . . . . string-array ⇝ *string*
Selected feature set, contains only entries from FeatureList.

▷ **Score** (output_control) . . . . . real-array ⇝ *real*
Achieved score using tow-fold cross-validation.

Result

If the parameters are valid, the operator select_feature_set_trainf_svm_protected returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

protect_ocr_trainf

Alternatives

select_feature_set_trainf_mlp_protected

See also

select_feature_set_trainf_svm, select_feature_set_svm

Module

OCR/OCV

serialize_ocr_class_svm (: : OCRHandle : SerializedItemHandle)

Serialize a SVM-based OCR classifier

serialize_ocr_class_svm serializes a SVM-based OCR classifier (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_ocr_class_svm is converted to a serialized item. The OCR classifier is defined by the handle OCRHandle. The serialized OCR classifier is returned by the handle SerializedItemHandle and can be deserialized by deserialize_ocr_class_svm.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_svm ⇝ *handle*
Handle of the OCR classifier.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_ocr_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`trainf_ocr_class_svm`

Possible Successors

`clear_ocr_class_svm`, `fwrite_serialized_item`, `send_serialized_item`, `deserialize_ocr_class_svm`

See also

`create_ocr_class_svm`, `read_ocr_class_svm`, `write_class_svm`, `read_class_svm`, `deserialize_ocr_class_svm`

Module

OCR/OCV

trainf_ocr_class_svm (: : OCRHandle, TrainingFile, Epsilon, TrainMode :)

Train an OCR classifier.

`trainf_ocr_class_svm` trains the OCR classifier `OCRHandle` with the training characters stored in the OCR training files given by `TrainingFile`. The training files must have been created, e.g., using `write_ocr_trainf`, before calling `trainf_ocr_class_svm`. The parameters `Epsilon` and `TrainMode` have the same meaning as in `train_class_svm`. Please, note that training characters that have no corresponding class in the classifier `OCRHandle` are discarded.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_svm ⇝ *handle*
Handle of the OCR classifier.

▷ **TrainingFile** (input_control) . . . . . filename.read(-array) ⇝ *string*
Names of the training files.
Default: 'ocr.trf'
File extension: `.trf`, `.otr`

▷ **Epsilon** (input_control) . . . . . real ⇝ *real*
Stop parameter for training.
Default: 0.001
Suggested values: `Epsilon` ∈ {0.00001, 0.0001, 0.001, 0.01, 0.1}

▷ **TrainMode** (input_control) . . . . . number ⇝ *string* / integer
Mode of training.
Default: 'default'
List of values: `TrainMode` ∈ {'default', 'add_sv_to_train_set'}

Example

```
* Train an OCR classifier
```

```
read_ocr_trainf_names ('ocr.trf', CharacterNames, CharacterCount)
create_ocr_class_svm (8, 10, 'constant', 'default', CharacterNames, \
                      'rbf', 0.01, 0.01, 'one-versus-one', \
                      'normalization', 81, OCRHandle)
trainf_ocr_class_svm (OCRHandle, 'ocr.trf', 0.001, 'default')
write_ocr_class_svm (OCRHandle, 'ocr.osc')
```

Result

If the parameters are valid the operator `trainf_ocr_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

`trainf_ocr_class_svm` may return the error 9211 (Matrix is not positive definite) if `Preprocessing` = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class. In this case we recommend to change `Preprocessing` to *'normalization'*. Another solution can be to add more training samples.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_ocr_class_svm`, `write_ocr_trainf`, `append_ocr_trainf`, `write_ocr_trainf_image`

Possible Successors

`do_ocr_single_class_svm`, `do_ocr_multi_class_svm`, `write_ocr_class_svm`

Alternatives

`read_ocr_class_svm`

See also

`train_class_svm`

Module

OCR/OCV

trainf_ocr_class_svm_protected (: : OCRHandle, TrainingFile, Password, Epsilon, TrainMode :)

Train an OCR classifier with data from a (protected) training file.

`trainf_ocr_class_svm_protected` trains the OCR classifier `OCRHandle` with the training data stored in the OCR training files given by `TrainingFile`. Its functionality corresponds to the functionality of `trainf_ocr_class_svm`, with the addition that `trainf_ocr_class_svm_protected` can process unprotected and protected training files. Protected training files can be used only with the correct user password `Password`. If the number of passwords `Password` equals 1, then every input file `TrainingFile` is checked with that password, otherwise the number of passwords has to be equal to the number of input files and the input file at position n is checked with the password at position n. For unprotected training files the passwords are ignored.

For a more detailed description of the operator's functionality see `trainf_ocr_class_svm`. The concept of protecting OCR training data in HALCON is described in `protect_ocr_trainf`.

Parameters

▷ **OCRHandle** (input_control) . . . . . ocr_svm ⇝ *handle*
Handle of the OCR classifier.

▷ **TrainingFile** (input_control) . . . . . filename.read(-array) ⇝ *string*
Names of the training files.
Default: 'ocr.trf'
File extension: `.trf`, `.otr`

▷ **Password** (input_control) . . . . . string(-array) ⇝ *string*
Passwords for protected training files.

▷ **Epsilon** (input_control) . . . . . real ⇝ *real*
Stop parameter for training.
Default: 0.001
Suggested values: `Epsilon` ∈ {0.00001, 0.0001, 0.001, 0.01, 0.1}

▷ **TrainMode** (input_control) . . . . . number ⇝ *string* / integer
Mode of training.
Default: 'default'
List of values: `TrainMode` ∈ {'default', 'add_sv_to_train_set'}

Result

If the parameters are valid the operator `trainf_ocr_class_svm_protected` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

`trainf_ocr_class_svm_protected` may return the error 9211 (Matrix is not positive definite) if `Preprocessing` = *'canonical_variates'* is used. This typically indicates that not enough training samples have been stored for each class. In this case we recommend to change `Preprocessing` to *'normalization'*. Another solution can be to add more training samples.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- OCRHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_ocr_class_svm`, `write_ocr_trainf`, `append_ocr_trainf`, `write_ocr_trainf_image`, `protect_ocr_trainf`

Possible Successors

`do_ocr_single_class_svm`, `do_ocr_multi_class_svm`, `write_ocr_class_svm`

Alternatives

`read_ocr_class_svm`

See also

`trainf_ocr_class_svm`, `train_class_svm`

Module

OCR/OCV

write_ocr_class_svm (: : OCRHandle, FileName :)

Write an OCR classifier to a file.

`write_ocr_class_svm` writes the OCR classifier `OCRHandle` to the file given by `FileName`. If a file extension is not specified in `FileName`, the default extension `'.osc'` is appended to

FileName. write_ocr_class_svm is typically called after the classifier has been trained with trainf_ocr_class_svm. The classifier can be read with read_ocr_class_svm.

Parameters

▷ **OCRHandle** (input_control) ..... ocr_svm ⇝ *handle*
Handle of the OCR classifier.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
File name.
File extension: .osc

Result

If the parameters are valid the operator write_ocr_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

trainf_ocr_class_svm

Possible Successors

clear_ocr_class_svm

See also

create_ocr_class_svm, read_ocr_class_svm, write_class_svm, read_class_svm

Module

OCR/OCV

21.8 Training Files

append_ocr_trainf (Character, Image : : Class, TrainingFile :)

Add characters to a training file.

The operator append_ocr_trainf serves to prepare the training with the operator trainf_ocr_class_mlp or trainf_ocr_class_svm. Hereby regions, representing characters, including their gray values (region and pixel) and the corresponding class name will be written into a file. An arbitrary number of regions within one image is supported. For each character (region) in Character the corresponding class name must be specified in Class. The gray values are passed via the parameter Image. In contrast to the operator write_ocr_trainf the characters are appended to an existing file using the same training file format as this file. If the file does not exist, a new file is generated. In this case, the file format can be chosen by the parameter *'ocr_trainf_version'* of the operator set_system. If no file extension is specified in TrainingFile, the extension '.trf' is appended to the name.

Parameters

▷ **Character** (input_object) ..... region(-array) ⇝ *object*
Characters to be trained.

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
Gray values of the characters.

▷ **Class** (input_control) ..... string(-array) ⇝ *string*
Class (name) of the characters.

▷ **TrainingFile** (input_control) ..... filename.write ⇝ *string*
Name of the training file.
Default: 'train_ocr'
File extension: .trf, .otr

Example

```
char       name[128];
char       class[128];

read_image(&Image,"character.tiff");
binary_threshold(Image,&Dark, 'otsu', 'dark', &UsedThreshold);
connection(Dark,&Character);
count_obj(Character,&num);
create_tuple(&Class,num);
open_window(0,0,-1,-1,0,"","",&WindowHandle);
set_color(WindowHandle,"red");
for (i=0; i<num; i++) {
  select_obj(Character,&SingleCharacter,i);
  disp_region(SingleCharacter,WindowHandle);
  printf("class of character %d ?\n",i);
  scanf("%s\n",class);
  append_ocr_trainf(Character,Image,name,class);
}
```

Result

If the parameters are correct, the operator `append_ocr_trainf` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`threshold`, `connection`, `read_ocr_trainf`

Possible Successors

`trainf_ocr_class_mlp`, `trainf_ocr_class_svm`, `write_ocr_trainf`

Alternatives

`write_ocr_trainf`, `write_ocr_trainf_image`

Module

OCR/OCV

`concat_ocr_trainf ( : : SingleFiles, ComposedFile : )`

Concat training files.

The operator `concat_ocr_trainf` stores all characters which are contained in the files `SingleFiles` into a new file with the name `ComposedFile`. The file format can be defined by the parameter *'ocr_trainf_version'* of the operator `set_system`. If no file extension is specified in `ComposedFile`, the extension '.trf' is appended to the file name. It is not possible to use any of the files in `SingleFiles` as the new file `ComposedFile`.

Parameters

▷ **`SingleFiles`** (input_control) ..... filename.read(-array) ⇝ *string*
Names of the single training files.
Default: ''
File extension: `.trf`, `.otr`

▷ **`ComposedFile`** (input_control) ..... filename.write ⇝ *string*
Name of the composed training file.
Default: 'all_characters'
File extension: `.trf`

Result

If the parameters are correct, the operator `concat_ocr_trainf` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`write_ocr_trainf`, `append_ocr_trainf`

Possible Successors

`trainf_ocr_class_mlp`, `trainf_ocr_class_svm`, `write_ocr_trainf`

Module

OCR/OCV

`protect_ocr_trainf` (: : `TrainingFile`, `Password`, `TrainingFileProtected` :)

Protection of training data.

Obtaining training data for an OCR classifier can be a complicated and time consuming activity. Thus, training data is valuable and there might be the necessity to protect its usage. The general idea is to prevent protected OCR training data from being reviewed or modified in any way and to ensure that its usage, e.g., training an OCR classifier, is restricted by a user password.

The approach in HALCON is to obtain OCR training data in the usual way, then write it to an unprotected OCR training file using, e.g., `write_ocr_trainf`, and afterwards create a new file containing the OCR training data protected with a user password. Protected training files can be used with a set of special operators that also require the correct password in order to be able to process the data, e.g., for training an OCR classifier.

It is important to keep in mind that data in a protected OCR training file cannot be reviewed or modified in any way, thus it is important to keep the original, unprotected, training data files.

Protected OCR training data can be used for training of multilayer perceptron (MLP) classifiers and support vector machine (SVM) classifiers.

`protect_ocr_trainf` protects the OCR training data contained in the file `TrainingFile` and writes it to the protected training data file `TrainingFileProtected`. The protection is done with the user password `Password`. The same password has to be used afterwards in order to use the protected OCR training data, e.g., by the operators `trainf_ocr_class_mlp_protected` or `read_ocr_trainf_names_protected`. Empty passwords (or passwords containing the empty string) are not allowed. If the number of passwords `Password` equals 1, then every output file `TrainingFileProtected` is protected with that password, otherwise the number of passwords has to be equal to the number of input and output files and the output file at position n is protected with the password at position n.

Parameters

▷ **`TrainingFile`** (input_control) ..... filename.read ⇝ *string*
Names of the training files.
Default: ”
File extension: `.trf`, `.otr`

▷ **`Password`** (input_control) ..... string(-array) ⇝ *string*
Passwords for protecting the training files.

▷ **`TrainingFileProtected`** (input_control) ..... filename.write ⇝ *string*
Names of the protected training files.
File extension: `.trf`

Result

If the parameters are correct, the operator `protect_ocr_trainf` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

write_ocr_trainf, append_ocr_trainf, concat_ocr_trainf

Possible Successors

read_ocr_trainf_names_protected, trainf_ocr_class_mlp_protected, trainf_ocr_class_svm_protected, select_feature_set_trainf_mlp_protected, select_feature_set_trainf_svm_protected

See also

select_feature_set_trainf_mlp

Module

OCR/OCV

read_ocr_trainf (: Characters : TrainingFile : CharacterNames)

Read training characters from files and convert to images.

read_ocr_trainf reads all characters from the specified file names and converts them into images. The default HALCON file extension for the OCR training file is 'trf'. The domain is defined according to the foreground of the characters (as specified in write_ocr_trainf). The names of the characters are returned in CharacterNames. If more than one file name is given the files are processed in the order of the file names.

Parameters

▷ **Characters** (output_object) ..... image-array ⇝ *object :* byte / uint2
Images read from file.

▷ **TrainingFile** (input_control) ..... filename.read(-array) ⇝ *string*
Names of the training files.
Default: ''
File extension: .trf, .otr

▷ **CharacterNames** (output_control) ..... string-array ⇝ *string*
Names of the read characters.

Result

If the parameter values are correct the operator read_ocr_trainf returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

write_ocr_trainf

Possible Successors

disp_image, select_obj, zoom_image_size

Alternatives

read_ocr_trainf_select

Module

OCR/OCV

read_ocr_trainf_names (: : TrainingFile : CharacterNames, CharacterCount)

Query which characters are stored in a training file.

read_ocr_trainf_names extracts the names and frequency of all characters in the specified training files.

Parameters

▷ **TrainingFile** (input_control) ..... filename.read(-array) ⇝ *string*
Names of the training files.
Default: ”
File extension: .trf, .otr

▷ **CharacterNames** (output_control) ..... string(-array) ⇝ *string*
Names of the read characters.

▷ **CharacterCount** (output_control) ..... integer(-array) ⇝ *integer*
Number of characters.

Result

If the parameter values are correct the operator read_ocr_trainf_names returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

write_ocr_trainf

See also

trainf_ocr_class_svm, trainf_ocr_class_mlp

Module

OCR/OCV

read_ocr_trainf_names_protected (: : TrainingFile, Password : CharacterNames, CharacterCount)

Query which characters are stored in a (protected) training file.

read_ocr_trainf_names_protected extracts the names and frequency of all characters in the specified training files. Its functionality corresponds to the functionality of read_ocr_trainf_names, with the addition that read_ocr_trainf_names_protected can process unprotected and protected training files. Protected training files can be used only with the correct user password Password. If the number of passwords Password equals 1, then every input file TrainingFile is checked with that password, otherwise the number of passwords has to be equal to the number of input files and the input file at position n is checked with the password at position n. For unprotected training files the passwords are ignored.

For a more detailed description of the operator’s functionality see read_ocr_trainf_names. The concept of protecting OCR training data in HALCON is described in protect_ocr_trainf.

Parameters

▷ **TrainingFile** (input_control) ..... filename.read(-array) ⇝ *string*
Names of the training files.
Default: ”
File extension: .trf, .otr

▷ **Password** (input_control) ..... string(-array) ⇝ *string*
Passwords for protected training files.

▷ **CharacterNames** (output_control) ..... string(-array) ⇝ *string*
Names of the read characters.

▷ **CharacterCount** (output_control) . . . . . integer(-array) ⇝ *integer*
Number of characters.

Result

If the parameter values are correct the operator `read_ocr_trainf_names_protected` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`protect_ocr_trainf`

See also

`read_ocr_trainf_names`

Module

OCR/OCV

read_ocr_trainf_select (: Characters : TrainingFile, SearchNames : FoundNames)

Read training specific characters from files and convert to images.

`read_ocr_trainf_select` reads the characters given in `SearchNames` from the specified files and converts them into images. It works similar to `read_ocr_trainf` but here the characters which are extracted can be specified.

Parameters

▷ **Characters** (output_object) . . . . . image-array ⇝ *object :* byte / uint2
Images read from file.

▷ **TrainingFile** (input_control) . . . . . filename.read(-array) ⇝ *string*
Names of the training files.
Default: ”
File extension: `.trf`, `.otr`

▷ **SearchNames** (input_control) . . . . . string(-array) ⇝ *string*
Names of the characters to be extracted.
Default: ’0’

▷ **FoundNames** (output_control) . . . . . string(-array) ⇝ *string*
Names of the read characters.

Result

If the parameter values are correct the operator `read_ocr_trainf_select` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`write_ocr_trainf`

Possible Successors

`disp_image`, `select_obj`, `zoom_image_size`

Alternatives

`read_ocr_trainf`

Module

OCR/OCV

write_ocr_trainf (Character, Image : : Class, TrainingFile :)

Storing of training characters into a file.

The operator write_ocr_trainf serves to prepare the training with the operator trainf_ocr_class_mlp or trainf_ocr_class_svm. Hereby regions, representing characters, including their gray values (region and pixel) and the corresponding class name will be written into a file. An arbitrary number of regions within one image is supported. For each character (region) in Character the corresponding class name must be specified in Class. The gray values are passed via the parameter Image. If no file extension is specified in TrainingFile the extension '.trf' is appended to the file name. The version of the file format used for writing data can be defined by the parameter *'ocr_trainf_version'* of the operator set_system.

Parameters

- ▷ **Character** (input_object) ..... region(-array) ⇝ *object*
  Characters to be trained.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
  Gray values of the characters.
- ▷ **Class** (input_control) ..... string(-array) ⇝ *string*
  Class (name) of the characters.
- ▷ **TrainingFile** (input_control) ..... filename.write ⇝ *string*
  Name of the training file.
  Default: 'train_ocr'
  File extension: .trf

Result

If the parameters are correct, the operator write_ocr_trainf returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, connection, read_ocr_trainf

Possible Successors

trainf_ocr_class_mlp, trainf_ocr_class_svm

Module

OCR/OCV

write_ocr_trainf_image (Character : : Class, TrainingFile :)

Write characters into a training file.

The operator write_ocr_trainf_image is used to prepare the training with the operator trainf_ocr_class_mlp or trainf_ocr_class_svm. Hereby regions, representing characters, including their gray values (region and pixel) and the corresponding class name will be written into a file. An arbitrary number of regions within one image is supported. For each character (region) in Character the corresponding class name must be specified in Class. If no file extension is specified in TrainingFile the extension '.trf' is appended to the file name. In contrast to write_ocr_trainf one image per character is passed. The domain of this image defines the pixels which belong to the character. The file format can be defined by the parameter *'ocr_trainf_version'* of the operator set_system.

Parameters

▷ **Character** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2
Characters to be trained.

▷ **Class** (input_control) . . . . . string(-array) ⇝ *string*
Class (name) of the characters.

▷ **TrainingFile** (input_control) . . . . . filename.write ⇝ *string*
Name of the training file.
Default: 'train_ocr'
File extension: `.trf`

Result

If the parameters are correct, the operator `write_ocr_trainf_image` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`threshold`, `connection`, `read_ocr_trainf`

Possible Successors

`trainf_ocr_class_mlp`, `trainf_ocr_class_svm`

Alternatives

`write_ocr_trainf`, `append_ocr_trainf`

Module

OCR/OCV

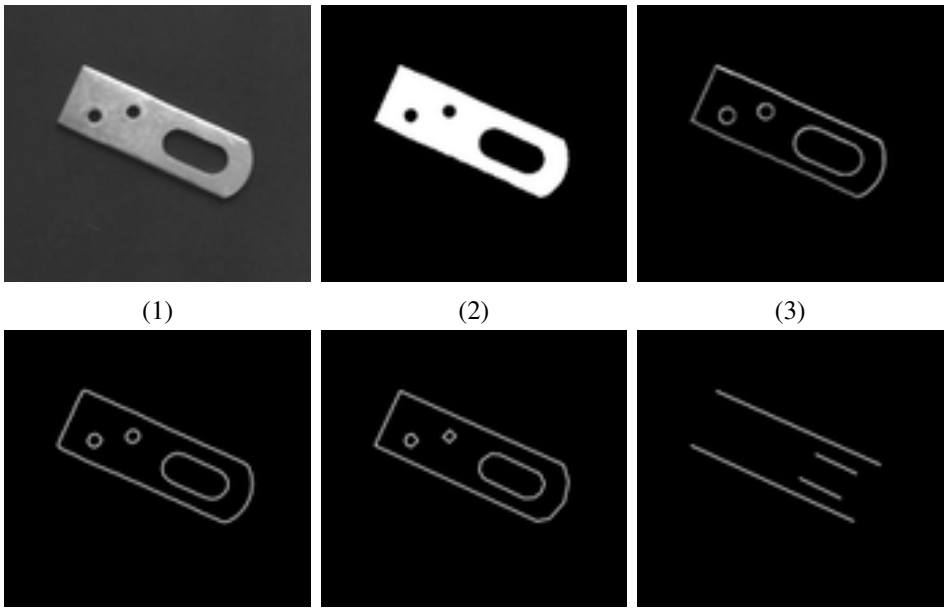
Chapter 22

Object

This chapter contains operators that can be used to either query information related to iconic objects or to manipulate iconic objects.

Available Iconic Objects

HALCON provides different kinds of iconic objects that are suitable for different needs:



(1) (2) (3)

(4) (5) (6)

Within the image of a metal part (1) a region (2), an ROI that is created using the dilated boundary of the region (3), contours (4), polygons (5), and parallels (6) are extracted.

Images: An image consists of one to multiple channels, i.e., matrices of similar size that contain gray values of various pixel types. Images are the major source for most machine vision tasks.

Regions: A region is defined as a set of pixels. The pixels of a region do not need to be connected. This means that even an arbitrary collection of pixels can be handled as a single region. Region processing is suitable, e.g., to apply a blob analysis within images or to define a region of interest (ROI) for following subpixel-precise operations.

XLDs: XLD is the abbreviation for eXtended Line Description and comprises all contour and polygon based data. Note that XLD contours are returned or used by many HALCON operators whereas XLD polygons and XLD parallels are needed only in special cases. A contour is a sequence of subpixel-accurate 2D control points that are connected by lines. Typically, the distance between control points is about one pixel. XLD objects contain, besides the control points, so-called local and global attributes. Typical examples for these

are, e.g., the edge amplitude of a control point or the regression parameters of a contour segment. Contour processing is suitable, e.g., for subpixel-precise measurements. Subpixel-precise operations can be faster if they are applied within an ROI.

Further Information

See the "Quick Guide" for further details about the available data structures.

22.1 Information

compare_obj (Objects1, Objects2 : : Epsilon : IsEqual)

Compare iconic objects regarding equality.

The operator compare_obj compares iconic objects regarding equality. The iconic objects are passed in the two input parameters Objects1 and Objects2 in form of (possibly mixed) tuples of images, regions, or XLDs. The n-th object in Objects1 is compared to the n-th object in Objects2 (for all n). If the two passed object tuples have the same length and if all objects are equal, the parameter IsEqual is set to 1, otherwise to 0.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Depending on the type of the input objects, different conditions must be met for the equality of two objects. In the following, the conditions for each possible iconic object are listed:

Images: For all channels, the gray values of the respective pixels must not differ by more than Epsilon. Note that images are only compared within their domain (ROI) and they are only equal if they have the same domain (ROI).

Regions: The regions must be equal. The parameter Epsilon has no effect for the comparison of regions.

XLD contours: XLD contours must have the same number of points and attributes. The point coordinates and the attribute values must not differ by more than Epsilon.

XLD polygons: XLD polygons must have the same number of line segments. The length and the orientation of these segments as well as the coordinates of the control points must not differ by more than Epsilon.

XLD parallels and extended XLD parallels: XLD parallels must have an identical index of the start and end line segment of the parallels of the first polygon (P1) and the second polygon (P2) as well as identical pointers to the underlying contours.

Modified XLD parallels: For modified XLD parallels, the same conditions apply as for XLD parallels and extended XLD parallels. Furthermore, the distances between the line segments of the parallel polygons must be identical.

Note that compare_obj compares all objects regarding their actual content. In contrast test_equal_obj compares only regions regarding their actual content while for all other objects their location in memory is compared.

Parameters

▷ **Objects1** (input_object) ..... object(-array) ⇝ *object*
Reference objects.

▷ **Objects2** (input_object) ..... object(-array) ⇝ *object*
Test objects.

▷ **Epsilon** (input_control) ..... number ⇝ *real* / integer
Maximum allowed difference between two gray values or coordinates etc.
Default: 0.0
Suggested values: Epsilon ∈ {0.0, 1.e-5}

▷ **IsEqual** (output_control) ..... integer ⇝ *integer*
Boolean result value.

Result

The operator compare_obj returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input objects available) is set via the operator set_system(::'no_object_result',<Result>:). If the number of objects differs an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`test_equal_obj`, `test_equal_region`

Module

Foundation

count_obj (Objects : : : Number)

Number of objects in a tuple.

The operator `count_obj` determines for the object parameter `Objects` the number of objects it contains. In this connection it should be noted that object is not the same as connection component (see `connection`). For example, the number of objects of one region consisting of three parts that are not connected is 1.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Parameters

▷ **Objects** (input_object) ..... object-array ⇝ *object*
Objects to be examined.

▷ **Number** (output_control) ..... integer ⇝ *integer*
Number of objects in the tuple `Objects`.

Complexity

Runtime complexity: $O(|\texttt{Objects}|)$.

Result

`count_obj` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`copy_obj`, `obj_to_integer`, `connection`, `set_system`

Module

Foundation

get_channel_info (Object : : Request, Channel : Information)

Information about the components of an image object.

The operator `get_channel_info` gives information about the components of an image object. The following requests (`Request`) are currently possible:

'creator' Output of the names of the procedures which initially created the image components (not the object).

'type' Output of the type of image component (`byte`, `int1`, `int2`, `uint2` `int4`, `real`, `direction`, `cyclic`, `complex`, `vector_field`). The component 0 is of type 'region' or 'xld'.

In the tuple Channel the numbers of the components about which information is required are stated. After carrying out get_channel_info, Information contains a tuple of strings (one string per entry in Channel) with the required information.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Parameters

▷ **Object** (input_object) . . . . . object ⇝ *object*
Image object to be examined.

▷ **Request** (input_control) . . . . . string ⇝ *string*
Required information about object components.
Default: 'creator'
List of values: Request ∈ {'creator', 'type'}

▷ **Channel** (input_control) . . . . . channel(-array) ⇝ *integer*
Components to be examined (0 for region/XLD).
Default: 0
Suggested values: Channel ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8}

▷ **Information** (output_control) . . . . . string(-array) ⇝ *string*
Requested information.

Result

If the parameters are correct the operator get_channel_info returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_image

See also

count_relation

Module

Foundation

get_obj_class (Object : : : Class)

Name of the class of an image object.

get_obj_class returns the name of the corresponding class to each object. The following classes are possible:

'image' Object with region (definition domain) and at least one channel.

'region' Object with a region without gray values.

'xld_cont' XLD object as contour

'xld_poly' XLD object as polygon

'xld_parallel' XLD object with parallel polygons

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Parameters

▷ **Object** (input_object) ..... object(-array) ⇝ *object*
Image objects to be examined.
▷ **Class** (output_control) ..... string(-array) ⇝ *string*
Name of class.

Result

If the parameter values are correct the operator `get_obj_class` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`disp_image`, `disp_region`, `disp_xld`

See also

`get_channel_info`, `count_relation`

Module

Foundation

test_equal_obj (Objects1, Objects2 : : : IsEqual)

Compare image objects regarding equality.

The operator `test_equal_obj` compares the regions and gray value components of all objects of the two input parameters. The n-th object in `Objects1` is compared to the n-th object in `Objects2` (for all n). If all corresponding regions are equal and the number of regions is also identical the parameter `IsEqual` is set to TRUE, otherwise FALSE.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Attention

Image matrices and XLDs are not compared regarding their contents. Thus, two images or XLDs, respectively, are “equal” if they are located at the same place in the storage. By contrast, regions that are not located at the same place in the storage are compared regarding their actual contents. If the input parameters are empty and the behavior was set via the operator `set_system(::'no_object_result','true':)`, the parameter `IsEqual` is set to TRUE, since all input (= empty set) is equal.

Parameters

▷ **Objects1** (input_object) ..... object-array ⇝ *object*
Test objects.
▷ **Objects2** (input_object) ..... object-array ⇝ *object*
Comparative objects.
▷ **IsEqual** (output_control) ..... integer ⇝ *integer*
Boolean result value.

Complexity

If F is the area of a region the runtime complexity is $O(1)$ or $O(sqrt(F))$ if the result is TRUE and $O(sqrt(F))$ if the result is FALSE.

Result

The operator `test_equal_obj` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. The behavior in case of empty input (no input objects available) is set via the operator `set_system(::'no_object_result',<Result>:)`. If the number of objects differs an exception is raised. Else `test_equal_obj` returns `2` (`H_MSG_TRUE`)

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

compare_obj, test_equal_region

Module

Foundation

22.2 Manipulation

clear_obj (Objects : : :)

Delete an iconic object from the HALCON database.

clear_obj deletes iconic objects, which are no longer needed, from the HALCON database. It should be noted that clear_obj is the only way to delete objects from the database, and hence to reclaim their memory, in HALCON/C. In all other HALCON language interfaces, clear_obj **must not** be used because objects are destroyed automatically through appropriate destructors.

Images and regions are normally used by several iconic objects at the same time (uses less memory!). This has the consequence that a region or an image is only deleted if all objects using it have been deleted.

The operator reset_obj_db can be used to reset the system and clear all remaining iconic objects.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Attention

Regarding the use of local variables in HALCON/C: When exiting a subroutine, the local variables are deleted, but the HALCON database is not updated. To update the database and thus free the memory, you must explicitly clear the local objects from the database before exiting the subroutine.

Parameters

▷ **Objects** (input_object) ..... object(-array) ⇝ *object*
Objects to be deleted.

Result

clear_obj returns 2 (H_MSG_TRUE) if all objects are contained in the HALCON database. If not all objects are valid (e.g., already cleared), an exception is raised, which also clears all valid objects. The operator set_check ('~clear') can be used to suppress the raising of this exception. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

reset_obj_db

See also

set_check

Module

none

concat_obj (Objects1, Objects2 : ObjectsConcat : :)

Concatenate two iconic object tuples.

concat_obj concatenates the two tuples of iconic objects Objects1 and Objects2 into a new tuple of iconic objects ObjectsConcat. Hence, this tuple contains all the iconic objects of the two input tuples:

ObjectsConcat = [Objects1,Objects2]

In ObjectsConcat, the objects of Objects1 are stored first, followed by the objects of Objects2, i.e., the order of the objects is preserved. Note that only references to the corresponding images and regions are stored in ObjectsConcat, i.e., no new memory is allocated. Furthermore, this means that modifications of input images, e.g., with set_grayval, overpaint_gray, or overpaint_region directly affect the images of the output tuple ObjectsConcat and vice versa.

concat_obj should not be confused with union1 or union2, in which regions are merged, i.e., in which the number of objects is modified.

concat_obj can be used to concatenate objects of different image object types (e.g., images and XLD contours) into a single object. This is only recommended if it is necessary to accumulate in a single object variable, for example, the results of an image processing sequence. It should be noted that the only operators that can handle such object tuples of mixed type are concat_obj, copy_obj, select_obj, and disp_obj.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Parameters

- ▷ **Objects1** (input_object) ..... object(-array) ⇝ *object*
  Object tuple 1.
- ▷ **Objects2** (input_object) ..... object(-array) ⇝ *object*
  Object tuple 2.
- ▷ **ObjectsConcat** (output_object) ..... object-array ⇝ *object*
  Concatenated objects.

Example

```
/* generate a tuple of a circle and a rectangle */

gen_circle(&Circle,200.0,400.0,23.0);
gen_rectangle1(&Rectangle,23.0,44.0,203.0,201.0);
concat_obj(Circle,Rectangle,&CircleAndRectangle);
disp_region(CircleAndRectangle,WindowHandle);
```

Complexity

Runtime complexity: $O(|\text{Objects1}| + |\text{Objects2}|)$;

Memory complexity of the result objects: $O(|\text{Objects1}| + |\text{Objects2}|)$

Result

concat_obj returns 2 (H_MSG_TRUE) if all objects are contained in the HALCON database. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

count_obj, copy_obj, select_obj, disp_obj

Module

Foundation

copy_obj (Objects : ObjectsSelected : Index, NumObj :)

Copy an iconic object in the HALCON database.

copy_obj copies NumObj iconic objects beginning with index Index (starting with 1) from the iconic input object tuple Objects to the output object ObjectsSelected. If -1 is passed for NumObj all objects beginning with Index are copied. No new storage is allocated for the regions and images. Instead, new objects containing references to the existing objects are created. The number of objects in an object tuple can be queried with the operator count_obj.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Parameters

▷ **Objects** (input_object) ..... object(-array) ⇝ *object*
Objects to be copied.

▷ **ObjectsSelected** (output_object) ..... object(-array) ⇝ *object*
Copied objects.

▷ **Index** (input_control) ..... integer ⇝ *integer*
Starting index of the objects to be copied.
Default: 1
Suggested values: Index ∈ {1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000}
Value range: 1 ≤ Index
Restriction: Index <= number(Objects)

▷ **NumObj** (input_control) ..... integer ⇝ *integer*
Number of objects to be copied or -1.
Default: 1
Suggested values: NumObj ∈ {-1, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000}
Value range: -1 ≤ NumObj
Restriction: NumObj + Index - 1 <= number(Objects) && NumObj != 0

Example

```
gen_circle (Circles, [100,200,400], [200,100,400], [100,100,50])
copy_obj (Circles, CircleCopy, 2, 3)
erosion_circle (CircleCopy, CircleErosion, 15.5)
dev_set_color ('red')
dev_set_draw ('fill')
dev_display(Circles)
dev_set_color ('white')
dev_set_draw ('margin')
dev_display (CircleCopy)
dev_set_color ('green')
dev_display (CircleErosion)
```

Complexity

Runtime complexity: $O(|\text{Objects}| + \text{NumObj})$;

Memory complexity of the result object: $O(\text{NumObj})$

Result

copy_obj returns 2 (H_MSG_TRUE) if all objects are contained in the HALCON database and all parameters are correct. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

count_obj

Alternatives

select_obj

See also

count_obj, concat_obj, obj_to_integer, copy_image

Module

Foundation

gen_empty_obj (: EmptyObject : :)

Create an empty object tuple.

The operator gen_empty_obj creates an empty tuple. This means that the output parameter does not contain any objects. Thus, the operator count_obj returns 0. However, clear_obj can be called for the output. It should be noted that no objects must not be confused with an empty region. In case of an empty region, i.e. a region with 0 pixels count_obj returns the value 1.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Parameters

▷ **EmptyObject** (output_object) ..... object ⇝ *object*
No objects.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

insert_obj (Objects, ObjectsInsert : ObjectsExtended : Index :)

Insert objects into an iconic object tuple.

insert_obj inserts the elements of ObjectsInsert at Index into the object tuple Objects and returns the result in ObjectsExtended.

Index must contain a single integer value. Note that, in contrast to control tuples, indices of object tuple elements start at 1. The length of ObjectsExtended is the sum of the length of Objects and ObjectsInsert. It is allowed to mix images, regions and XLDs in Objects and ObjectsInsert.

Parameters

▷ **Objects** (input_object) ..... object(-array) ⇝ *object*
Input object tuple.

▷ **ObjectsInsert** (input_object) ..... object(-array) ⇝ *object*
Object tuple to insert.

▷ **ObjectsExtended** (output_object) ..... object(-array) ⇝ *object*
Extended object tuple.

▷ **Index** (input_control) ..... number ⇝ *integer*
Index to insert objects.

Example

```
gen_image_const (Image1, 'byte', 1, 1)
gen_image_const (Image3, 'byte', 3, 3)
concat_obj (Image1, Image3, Images)
gen_image_const (Image, 'byte', 2, 2)
insert_obj (Images, Image, Images, 2)
```

Complexity

Typical runtime complexity: $O(|\texttt{Objects}| + |\texttt{ObjectsInsert}|)$.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

count_obj

Possible Successors

count_obj

Alternatives

concat_obj

See also

count_obj, select_obj, copy_obj, remove_obj

Module

Foundation

integer_to_obj (: Objects : SurrogateTuple :)

Convert an "integer number" into an iconic object.

integer_to_obj is the inverse operator to obj_to_integer. All surrogates of objects passed in SurrogateTuple are stored as objects. In contrast to obj_to_integer, the objects are duplicated. integer_to_obj is intended especially for use in HALCON/C, because iconic objects and control parameters are treated differently in C. Please note that if you pass the value *0* in SurrogateTuple, the program will crash because *0* is no valid pointer.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Attention

The objects are duplicated in the database.

Parameters

▷ **Objects** (output_object) ..... object(-array) ⇝ *object*
Created objects.

▷ **SurrogateTuple** (input_control) ..... pointer(-array) ⇝ *integer*
Tuple of object surrogates.

Result

integer_to_obj returns 2 (H_MSG_TRUE) if all parameters are correct, i.e., if they are valid object keys. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

obj_to_integer

Module

Foundation

obj_diff (Objects, ObjectsSub : ObjectsDiff : :)

Calculate the difference of two object tuples.

obj_diff calculates the set-theoretic difference of two object tuples:

(objects in Objects) − (objects in ObjectsSub)

The resulting object tuple ObjectsDiff is defined as the input tuple Objects with all objects from ObjectsSub removed.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Attention

Image matrices and XLDs are not compared regarding their contents. Thus, two images or XLDs, respectively, are "equal" if they are located at the same place in the storage. By contrast, regions that are not located at the same place in the storage are compared regarding their actual contents.

Parameters

- ▷ **Objects** (input_object) ..... object(-array) ⇝ *object*
  Object tuple 1.
- ▷ **ObjectsSub** (input_object) ..... object(-array) ⇝ *object*
  Object tuple 2.
- ▷ **ObjectsDiff** (output_object) ..... object(-array) ⇝ *object*
  Objects from Objects that are not part of ObjectsSub.

Result

obj_diff always returns 2 (H_MSG_TRUE).

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

test_equal_obj, count_obj, copy_obj, select_obj

Module

Foundation

obj_to_integer (Objects : : Index, Number : SurrogateTuple)

Convert an iconic object into an "integer number."

obj_to_integer stores Number, starting at index Index, of the database keys of the input object Objects as integer numbers in the output parameter SurrogateTuple. If -1 is passed for Number all objects beginning with Index are copied. This facilitates a direct access to an arbitrary element of Objects. In conjunction with count_obj (returns the number of objects in Objects) the elements of Objects can be processed successively. The objects are not duplicated by obj_to_integer and thus must not be cleared by clear_obj.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Attention

The objects' data is not duplicated.

Parameters

▷ **Objects** (input_object) . . . . . object(-array) ⇝ *object*
Objects for which the surrogates are to be returned.

▷ **Index** (input_control) . . . . . integer ⇝ *integer*
Starting index of the surrogates to be returned.
Default: 1
Value range: 1 ≤ Index

▷ **Number** (input_control) . . . . . integer ⇝ *integer*
Number of surrogates to be returned.
Default: -1
Restriction: Number == -1 || Number + Index <= number(Objects)

▷ **SurrogateTuple** (output_control) . . . . . pointer(-array) ⇝ *integer*
Tuple containing the surrogates.

Example

```
* Access the i-th element:
obj_to_integer(Objects,i,1,Surrogat)
```

Complexity

Runtime complexity: $O(|\texttt{Objects}| + \texttt{Number})$

Result

`obj_to_integer` returns 2 (`H_MSG_TRUE`) if all parameters are correct. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`copy_obj`, `select_obj`, `copy_image`, `gen_image_proto`

See also

`integer_to_obj`, `count_obj`

Module

Foundation

remove_obj (Objects : ObjectsReduced : Index :)

Remove objects from an iconic object tuple.

`remove_obj` removes the elements referred by `Index` from the tuple `Objects` and returns the rest in the tuple `ObjectsReduced`. `Index` determines the index of the elements to remove. Note that, in contrast to control tuples, indices for the object tuple elements start at 1, i.e. the first tuple element has got the index 1. Duplicates and indices out of range are ignored.

Parameters

▷ **Objects** (input_object) . . . . . object(-array) ⇝ *object*
Input object tuple.

▷ **ObjectsReduced** (output_object) . . . . . object(-array) ⇝ *object*
Remaining object tuple.

▷ **Index** (input_control) . . . . . number(-array) ⇝ *integer*
Indices of the objects to be removed.

Example

```
gen_empty_obj (Images)
for Index := 1 to 10 by 1
  gen_image_const (Image, 'byte', Index, Index)
  concat_obj (Images, Image, Images)
endfor

remove_obj (Images, Images, [7, 4])
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

select_obj

See also

count_obj, select_obj, copy_obj, insert_obj

Module

Foundation

replace_obj (Objects, ObjectsReplace : Replaced : Index :)

Replaces one or more elements of an iconic object tuple.

replace_obj replaces one or more elements of the iconic input tuple Objects and returns them with Replaced. At this, Index determines the indices of the elements and ObjectsReplace the corresponding objects to replace. The parameter Index must contain one or more integer values (any floating point number must represent an integer value without fraction). Indices of iconic object elements start at 1. If ObjectsReplace contains only one value, this value will be replaced at all indices of Index. If a value of Index is greater than the length of the iconic input object tuple Objects, Replaced will be extended accordingly and initialized with empty regions.

Exception: Empty input object

If either Index or ObjectsReplace is empty and the other is not, an exception is raised. If both are empty, the output object tuple Replaced corresponds to the input Objects. If both are not empty, but the input Objects is, the empty object will be extended as described above.

Parameters

- ▷ **Objects** (input_object) ..... object(-array) ⇝ *object*
  Iconic Input Object.
- ▷ **ObjectsReplace** (input_object) ..... object(-array) ⇝ *object*
  Element(s) to replace.
- ▷ **Replaced** (output_object) ..... object(-array) ⇝ *object*
  Tuple with replaced elements.
- ▷ **Index** (input_control) ..... number(-array) ⇝ *integer*
  Index/Indices of elements to be replaced.

Example

```
gen_empty_obj (Images)
for Index := 1 to 10 by 1
  gen_image_const (Image, 'byte', Index, Index)
  concat_obj (Images, Image, Images)
```

```
endfor
gen_empty_obj (Replace)

replace_obj (Images, Replace, Images, 1)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

select_obj

See also

count_obj, select_obj, copy_obj, insert_obj

Module

Foundation

select_obj (Objects : ObjectSelected : Index :)

Select objects from an object tuple.

select_obj copies the iconic objects with the indices given by Index (starting with 1) from the iconic input object tuple Objects to the output object ObjectSelected. No new storage is allocated for the regions and images. Instead, new objects containing references to the existing objects are created. The number of objects in an object tuple can be queried with the operator count_obj.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Parameters

▷ **Objects** (input_object) ..... object(-array) ⇝ *object*
Input objects.

▷ **ObjectSelected** (output_object) ..... object(-array) ⇝ *object*
Selected objects.

▷ **Index** (input_control) ..... integer(-array) ⇝ *integer*
Indices of the objects to be selected.
Default: 1
Suggested values: Index ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 50, 100, 200, 500, 1000, 2000, 5000}
Restriction: Index >= 1

Example

```
/* Access to all Regions */

count_obj(Regions,&Num);
for (i=1; i<=Num; i++)
{
  select_obj(Regions,&Single,i);
  T_get_region_polygon(Single,5.0,&Row,&Column);
  T_disp_polygon(WindowHandleTuple,Row,Column);
  destroy_tuple(Row);
  destroy_tuple(Column);
}
```

Complexity

Runtime complexity: $O(|\text{Objects}|)$

Result

`select_obj` returns `2` (`H_MSG_TRUE`) if all objects are contained in the HALCON database and all parameters are correct. If the input is empty the behavior can be set via `set_system(::'no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Supports objects on compute devices.
- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`count_obj`

Alternatives

`copy_obj`

See also

`count_obj`, `concat_obj`, `obj_to_integer`

Module

Foundation

Chapter 23

Regions

To learn more about the basic concept of regions in HALCON you can also see the example program `halcon_basic_concepts.hdev`.

23.1 Access

get_region_contour (Region : : : Rows, Columns)

Access the contour of an object.

The operator `get_region_contour` returns the contour of a region. A contour is a result of line (`Rows`) and column coordinates (`Columns`), describing the boundary of the region. The contour lies on the region. It starts at the smallest line number. In that line at the pixel with the largest column index. The rotation direction is clockwise. The first pixel of the contour is identical with the last. Holes of the region are ignored. The operator `get_region_contour` returns the coordinates in the form of tuples. An empty region is passed as empty tuple.

Attention

Holes of the region are ignored. Only one region may be passed, and this region must have exactly one connection component.

Parameters

- ▷ **Region** (input_object) ..... region ⇝ *object*
  Output region.
- ▷ **Rows** (output_control) ..... contour.y-array ⇝ *integer*
  Line numbers of the contour pixels.
- ▷ **Columns** (output_control) ..... contour.x-array ⇝ *integer*
  Column numbers of the contour pixels.
  Number of elements: Columns == Rows

Result

The operator `get_region_contour` normally returns the value `2` (`H_MSG_TRUE`). If more than one connection component is passed an exception is raised. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`sobel_amp`, `threshold`, `skeleton`, `edges_image`, `gen_rectangle1`, `gen_circle`

See also

copy_obj, get_region_polygon

Module

Foundation

get_region_convex (Region : : : Rows, Columns)

Access convex hull as contour.

The operator get_region_convex returns the convex hull of a region as polygon. The polygon is the minimum result of line (Rows) and column coordinates (Columns) describing the hull of the region. The polygon pixels lie on the region. The polygon starts at the smallest line number; in this line at the pixel with the largest column index. The rotation direction is clockwise. The first pixel of the polygon is identical with the last. The operator get_region_convex returns the coordinates in the form of tuples. An empty region is passed as empty tuple.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
Output region.

▷ **Rows** (output_control) . . . . . contour.y-array ⇝ *integer*
Line numbers of contour pixels.

▷ **Columns** (output_control) . . . . . contour.x-array ⇝ *integer*
Column numbers of the contour pixels.
Number of elements: Columns == Rows

Result

The operator get_region_convex returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, skeleton, dyn_threshold

Possible Successors

disp_polygon

Alternatives

shape_trans

See also

select_obj, get_region_contour

Module

Foundation

get_region_points (Region : : : Rows, Columns)

Access the pixels of a region.

The operator get_region_points returns the region data in the form of coordinate lists. The coordinates are sorted in the following order:

$$(r_1, c_1) \leq (r_2, c_2) := (r_1 < r_2) \quad \vee \quad (r_1 = r_2) \wedge (c1 \leq c2)$$

get_region_points returns the coordinates in the form of tuples. An empty region is passed as empty tuple.

Attention

Only one region may be passed.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
This region is accessed.

▷ **Rows** (output_control) . . . . . coordinates.y-array ⇝ *integer*
Line numbers of the pixels in the region

▷ **Columns** (output_control) . . . . . coordinates.x-array ⇝ *integer*
Column numbers of the pixels in the region.
Number of elements: Columns == Rows

Result

The operator `get_region_points` normally returns the value `2` (`H_MSG_TRUE`). If more than one connection component is passed an exception is raised. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`sobel_amp`, `threshold`, `connection`

Alternatives

`get_region_runs`

See also

`copy_obj`, `gen_region_points`

Module

Foundation

get_region_polygon (Region : : Tolerance : Rows, Columns)

Polygon approximation of a region.

The operator `get_region_polygon` calculates a polygon to approximate the edge of a region. A polygon is a sequence of line (`Rows`) and column coordinates (`Columns`). It describes the contour of the region. Only the base points of the polygon are returned. The parameter `Tolerance` indicates how large the maximum distance between the polygon and the edge of the region may be. Holes of the region are ignored. The operator `get_region_polygon` returns the coordinates in the form of tuples.

Attention

Holes of the region are ignored. Only one region may be passed, and this region must have exactly one connection component.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
Region to be approximated.

▷ **Tolerance** (input_control) . . . . . number ⇝ *real* / integer
Maximum distance between the polygon and the edge of the region.
Default: 5.0
Suggested values: `Tolerance` ∈ {0.0, 2.0, 5.0, 10.0}
Value range: 0.0 ≤ Tolerance (lin)
Minimum increment: 0.01
Recommended increment: 1.0

▷ **Rows** (output_control) . . . . . polygon.y-array ⇝ *integer*
Line numbers of the base points of the contour.

▷ **Columns** (output_control) ..... polygon.x-array ⇝ *integer*
Column numbers of the base points of the contour.
Number of elements: Columns == Rows

Result

The operator `get_region_polygon` normally returns the value `2` (`H_MSG_TRUE`). If more than one connection component is passed an exception is raised. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`sobel_amp`, `threshold`, `skeleton`, `edges_image`

See also

`copy_obj`, `gen_region_polygon`, `disp_polygon`, `get_region_contour`, `set_line_approx`

Module

Foundation

get_region_runs (Region : : : Row, ColumnBegin, ColumnEnd)

Access the runlength coding of a region.

The operator `get_region_runs` returns the region data in the form of chord tuples. The chord representation is caused by examining a region line by line with ascending line number (= from "top" to "bottom"). Every line is passed from left to right (ascending column number); storing all starting and ending points of region segments (= chords). Thus a region can be described by a sequence of chords, a chord being defined by line number, starting and ending points (column number). The operator `get_region_runs` returns the three components of the chords in the form of tuples. In case of an empty region three empty tuples are returned.

Attention

Only one region may be passed.

Parameters

▷ **Region** (input_object) ..... region ⇝ *object*
Output region.

▷ **Row** (output_control) ..... chord.y-array ⇝ *integer*
Line numbers of the chords.

▷ **ColumnBegin** (output_control) ..... chord.x1-array ⇝ *integer*
Column numbers of the starting points of the chords.
Number of elements: ColumnBegin == Row

▷ **ColumnEnd** (output_control) ..... chord.x2-array ⇝ *integer*
Column numbers of the ending points of the chords.
Number of elements: ColumnEnd == Row

Result

The operator `get_region_runs` normally returns the value `2` (`H_MSG_TRUE`). If more than one region is passed an exception is raised. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, connection

Alternatives

get_region_points

See also

copy_obj, gen_region_runs

Module

Foundation

23.2 Creation

gen_checker_region (: RegionChecker : WidthRegion, HeightRegion, WidthPattern, HeightPattern :)

Create a checkered region.

The operator gen_checker_region returns a checkered region. Every black field of the checkerboard belongs to the region. The horizontal and vertical expansion of the region is limited by WidthRegion, HeightRegion respectively, the size of the fields of the checkerboard by WidthPattern × HeightPattern.

Attention

If a very small pattern is chosen (WidthPattern < 4) the created region requires much storage.

Parameters

▷ **RegionChecker** (output_object) . . . . . region ⇝ *object*
Created checkerboard region.

▷ **WidthRegion** (input_control) . . . . . extent.x ⇝ *integer*
Largest occurring x value of the region.
Default: 511
Suggested values: WidthRegion ∈ {10, 20, 31, 63, 127, 255, 300, 400, 511}
Value range: 1 ≤ WidthRegion (lin)
Minimum increment: 1
Recommended increment: 10

▷ **HeightRegion** (input_control) . . . . . extent.y ⇝ *integer*
Largest occurring y value of the region.
Default: 511
Suggested values: HeightRegion ∈ {10, 20, 31, 63, 127, 255, 300, 400, 511}
Value range: 1 ≤ HeightRegion (lin)
Minimum increment: 1
Recommended increment: 10

▷ **WidthPattern** (input_control) . . . . . extent.x ⇝ *integer*
Width of a field of the checkerboard.
Default: 64
Suggested values: WidthPattern ∈ {1, 2, 4, 8, 16, 20, 32, 64, 100, 128, 200, 300, 500}
Value range: 1 ≤ WidthPattern (lin)
Minimum increment: 1
Recommended increment: 10
Restriction: WidthPattern < WidthRegion

▷ **HeightPattern** (input_control) . . . . . extent.y ⇝ *integer*
Height of a field of the checkerboard.
Default: 64
Suggested values: HeightPattern ∈ {1, 2, 4, 8, 16, 20, 32, 64, 100, 128, 200, 300, 500}
Value range: 1 ≤ HeightPattern (lin)
Minimum increment: 1
Recommended increment: 10
Restriction: HeightPattern < HeightRegion

Example

```
gen_checker_region(Checker,512,512,32,64)
dev_set_draw('fill')
dev_display (Checker)
```

Complexity

The required storage (in bytes) for the region is:

$O((\texttt{WidthRegion} * \texttt{HeightRegion})/\texttt{WidthPattern})$

Result

The operator gen_checker_region returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

paint_region

Alternatives

gen_grid_region, gen_region_polygon_filled, gen_region_points, gen_region_runs, gen_rectangle1, concat_obj, gen_random_region, gen_random_regions

See also

reduce_domain

Module

Foundation

gen_circle (: Circle : Row, Column, Radius :)

Create a circle.

The operator gen_circle generates one or more circles described by the center and Radius. If several circles shall be generated the coordinates must be passed in the form of tuples.

gen_circle only creates symmetric circles. To achieve this, the radius is internally rounded down to a multiple of 0.5. If an integer number is specified for the radius (i.e., 1, 2, 3, ...) an even diameter is obtained, and hence the circle can only be symmetric with respect to a center with coordinates that have a fractional part of 0.5. Consequently, internally the coordinates of the center are adapted to the closest coordinates that have a fractional part of 0.5. Here, integer coordinates are rounded down to the next smaller values with a fractional part of 0.5. For odd diameters (i.e., radius = 1.5, 2.5, 3.5, ...), the circle can only be symmetric with respect to a center with integer coordinates. Hence, internally the coordinates of the center are rounded to the nearest integer coordinates. It should be noted that the above algorithm may lead to the fact that circles with an even diameter are *not* contained in circles with the next larger odd diameter, even if the coordinates specified in Row and Column are identical.

If the circle extends beyond the image edge it is clipped to the current image format if the value of the system flag *'clip_region'* is set to *'true'* (set_system).

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Attention

For speed reasons, the resulting region may contain additional pixels at the border and some individual pixels at the border may be missing. This may lead to an inconsistency between the operators smallest_circle and gen_circle.

Parameters

▷ **Circle** (output_object) . . . . . region(-array) ⇝ *object*
Generated circle.

▷ **Row** (input_control) . . . . . circle.center.y(-array) ⇝ *real* / integer
Line index of center.
Default: 200.0
Suggested values: Row ∈ {0.0, 10.0, 50.0, 100.0, 200.0, 300.0}
Value range: 1.0 ≤ Row ≤ 1024.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Column** (input_control) . . . . . circle.center.x(-array) ⇝ *real* / integer
Column index of center.
Default: 200.0
Suggested values: Column ∈ {0.0, 10.0, 50.0, 100.0, 200.0, 300.0}
Value range: 1.0 ≤ Column ≤ 1024.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Radius** (input_control) . . . . . circle.radius(-array) ⇝ *real* / integer
Radius of circle.
Default: 100.5
Suggested values: Radius ∈ {1.0, 1.5, 2.0, 2.5, 3, 3.5, 4, 4.5, 5.5, 6.5, 7.5, 9.5, 11.5, 15.5, 20.5, 25.5, 31.5, 50.5}
(lin)
Restriction: Radius > 0.0

Example

```
read_image(Image,'fabrik')
gen_circle(Circle,300.0,200.0,150.5)
reduce_domain(Image,Circle,Mask)
dev_clear_window ()
dev_display (Mask)
```

Complexity

Runtime complexity: $O(\texttt{Radius} * 2)$

Storage complexity (byte): $O(\texttt{Radius} * 8)$

Result

If the parameter values are correct, the operator gen_circle returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>). If an empty region is created by clipping (the circle is completely outside of the image format) the operator set_system ('store_empty_region',<'true'/'false'>) determines whether the empty region is put out.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

paint_region, reduce_domain

Alternatives

gen_ellipse, gen_region_polygon_filled, gen_region_points, gen_region_runs, draw_circle

See also

disp_circle, set_shape, smallest_circle, reduce_domain

Module

Foundation

gen_circle_sector (: CircleSector : Row, Column, Radius, StartAngle, EndAngle :)

Create a circle sector.

The operator gen_circle_sector generates one or more circles sectors described by the center, Radius, StartAngle and EndAngle. If several circle sectors shall be generated the coordinates must be passed in the form of tuples.

gen_circle_sector only creates symmetric circle sectors with respect to the center of coordinates. To achieve this, the radius is internally rounded down to a multiple of 0.5. If an integer number is specified for the radius (i.e., 1, 2, 3, ...) an even diameter is obtained, and hence the circle can only be symmetric with respect to a center with coordinates that have a fractional part of 0.5. Consequently, internally the coordinates of the center are adapted to the closest coordinates that have a fractional part of 0.5. Here, integer coordinates are rounded down to the next smaller values with a fractional part of 0.5. For odd diameters (i.e., radius = 1.5, 2.5, 3.5, ...), the circle can only be symmetric with respect to a center with integer coordinates. Hence, internally the coordinates of the center are rounded to the nearest integer coordinates. It should be noted that the above algorithm may lead to the fact that circles with an even diameter are *not* contained in circles with the next larger odd diameter, even if the coordinates specified in Row and Column are identical.
The angles are given in radians in mathematically positive direction. See the examples illustrated in the figure below. Note, *'rad(360)'* is equivalent to *0*. As a consequence a sector with StartAngle = *0* and EndAngle = *'rad(360)'* results in an empty region.

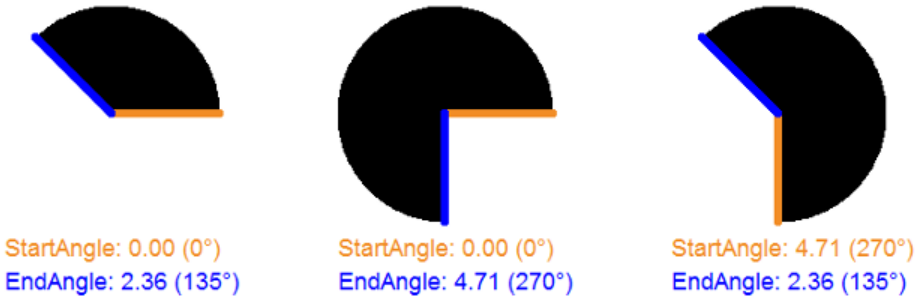


Illustration of different possible angle combinations.

If the circle extends beyond the image edge it is clipped to the current image format if the value of the system flag *'clip_region'* is set to *'true'* (set_system).

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

▷ **CircleSector** (output_object) . . . . . region(-array) ⇝ *object*
Generated circle sector.

▷ **Row** (input_control) . . . . . coordinates.y(-array) ⇝ *real* / integer
Line index of center.
Default: 200.0
Suggested values: Row ∈ {0.0, 10.0, 50.0, 100.0, 200.0, 300.0}
Value range: 1.0 ≤ Row ≤ 1024.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Column** (input_control) . . . . . coordinates.x(-array) ⇝ *real* / integer
Column index of center.
Default: 200.0
Suggested values: Column ∈ {0.0, 10.0, 50.0, 100.0, 200.0, 300.0}
Value range: 1.0 ≤ Column ≤ 1024.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Radius** (input_control) . . . . . number(-array) ⇝ *real* / integer
Radius of circle.
Default: 100.5
Suggested values: Radius ∈ {1.0, 1.5, 2.0, 2.5, 3, 3.5, 4, 4.5, 5.5, 6.5, 7.5, 9.5, 11.5, 15.5, 20.5, 25.5, 31.5, 50.5}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: Radius > 0.0

▷ **StartAngle** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
Start angle of the circle sector.
Default: 0.0
Suggested values: StartAngle ∈ {0.0, 0.785398, 1.5708, 2.35619, 3.14159, 3.92699, 4.71239, 5.49779, 6.28318}
Value range: 0 ≤ StartAngle ≤ 6.28318 (lin)

▷ **EndAngle** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
End angle of the circle sector.
Default: 3.14159
Suggested values: EndAngle ∈ {0.0, 0.785398, 1.5708, 2.35619, 3.14159, 3.92699, 4.71239, 5.49779, 6.28318}
Value range: 0 ≤ EndAngle ≤ 6.28318 (lin)

Example

```
read_image(Image,'fabrik')
gen_circle_sector(CircleSector,300.0,200.0,150.5,0,rad(120))
reduce_domain(Image,CircleSector,Mask)
dev_clear_window ()
dev_display (Mask)
```

Complexity

Runtime complexity: $O(\texttt{Radius} * 2)$

Storage complexity (byte): $O(\texttt{Radius} * 8)$

Result

If the parameter values are correct, the operator gen_circle_sector returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>). If an empty region is created by clipping (the circle is completely outside of the image format) the operator set_system ('store_empty_region',<'true'/'false'>) determines whether the empty region is put out.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

paint_region, reduce_domain

Alternatives

gen_ellipse, gen_ellipse_sector, gen_region_polygon_filled, gen_region_points, gen_region_runs, draw_circle

See also

disp_circle, disp_region, set_shape, smallest_circle, reduce_domain

Module

Foundation

gen_ellipse (: Ellipse : Row, Column, Phi, Radius1, Radius2 :)

Create an ellipse.

The operator gen_ellipse generates one or more ellipses with the center (Row, Column), the orientation Phi and the half-radii (half axes) Radius1 and Radius2. The angle is indicated in arc measure according to the x axis in mathematically positive direction. More than one region can be created by passing tuples of parameter values.

The center must be located within the image coordinates. The coordinate system runs from (0,0) (upper left corner) to (Width-1,Height-1). See get_system and reset_obj_db in this context. If the ellipse reaches beyond the edge of the image it is clipped to the current image format according to the value of the system flag *'clip_region'* (set_system).

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

▷ **Ellipse** (output_object) . . . . . region(-array) ⇝ *object*
Created ellipse(s).

▷ **Row** (input_control) . . . . . ellipse.center.y(-array) ⇝ *real* / integer
Line index of center.
Default: 200.0
Suggested values: Row ∈ {0.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Value range: 1.0 ≤ Row ≤ 1024.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Column** (input_control) . . . . . ellipse.center.x(-array) ⇝ *real* / integer
Column index of center.
Default: 200.0
Suggested values: Column ∈ {0.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Value range: 1.0 ≤ Column ≤ 1024.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Phi** (input_control) . . . . . ellipse.angle.rad(-array) ⇝ *real* / integer
Orientation of the longer radius (Radius1).
Default: 0.0
Suggested values: Phi ∈ {-1.178097, -0.785398, -0.392699, 0.0, 0.392699, 0.785398, 1.178097}
Value range: -1.178097 ≤ Phi ≤ 1.178097 (lin)

▷ **Radius1** (input_control) . . . . . ellipse.radius1(-array) ⇝ *real* / integer
Longer radius.
Default: 100.0
Suggested values: Radius1 ∈ {2.0, 5.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: Radius1 > 0

▷ **Radius2** (input_control) . . . . . ellipse.radius2(-array) ⇝ *real* / integer
Shorter radius.
Default: 60.0
Suggested values: Radius2 ∈ {1.0, 2.0, 4.0, 5.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: Radius2 > 0 && Radius2 <= Radius1

Example

```
read_image (Image, 'fabrik')
```

```
gen_ellipse (Ellipse, 200, 200, rad(10), 160, 90)
reduce_domain (Image, Ellipse, ImageReduced)
dev_clear_window ()
dev_display (ImageReduced)
```

Complexity

Runtime complexity: $O(\texttt{Radius1} * 2)$

Storage complexity (byte): $O(\texttt{Radius1} * 8)$

Result

If the parameter values are correct, the operator `gen_ellipse` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised. The clipping according to the current image format is set via the operator `set_system ('clip_region',<'true'/'false'>)`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`paint_region`, `reduce_domain`

Alternatives

`gen_circle`, `gen_region_polygon_filled`, `draw_ellipse`

See also

`disp_ellipse`, `set_shape`, `smallest_circle`, `reduce_domain`

Module

Foundation

gen_ellipse_sector (: EllipseSector : Row, Column, Phi, Radius1, Radius2, StartAngle, EndAngle :)

Create an ellipse sector.

The operator `gen_ellipse_sector` generates one or more ellipse sectors with the center (`Row`, `Column`), the orientation `Phi`, the half-radii `Radius1` and `Radius2`, and the start and end angles `StartAngle` and `EndAngle`. The angles are given in radians in mathematically positive direction. An illustration showing this can be found in the reference of `gen_circle_sector`. Note, *'rad(360)'* is equivalent to *0*. As a consequence a sector with `StartAngle` = *0* and `EndAngle` = *'rad(360)'* results in an empty region. More than one region can be created by passing tuples of parameter values.

The center must be located within the image coordinates. The coordinate system runs from (0,0) (upper left corner) to (Width-1,Height-1). See `get_system` and `reset_obj_db` in this context. If the ellipse reaches beyond the edge of the image it is clipped to the current image format according to the value of the system flag *'clip_region'* (`set_system`).

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

▷ **EllipseSector** (output_object) ..... region(-array) ⇝ *object*
Created ellipse(s).

▷ **Row** (input_control) ..... coordinates.y(-array) ⇝ *real* / integer
Line index of center.
Default: 200.0
Suggested values: `Row` ∈ {0.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Value range: 1.0 ≤ `Row` ≤ 1024.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Column** (input_control) . . . . . coordinates.x(-array) ⇝ *real* / integer
Column index of center.
Default: 200.0
Suggested values: Column ∈ {0.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Value range: 1.0 ≤ Column ≤ 1024.0 (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Phi** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
Orientation of the longer radius (Radius1).
Default: 0.0
Suggested values: Phi ∈ {-1.5707, -1.1781, -0.785398, -0.392699, 0.0, 0.392699, 0.785398, 1.1781, 1.5707}
Value range: -1.570796 ≤ Phi ≤ 1.570796 (lin)
Restriction: - pi / 2 <= Phi <= pi / 2

▷ **Radius1** (input_control) . . . . . number(-array) ⇝ *real* / integer
Longer radius.
Default: 100.0
Suggested values: Radius1 ∈ {2.0, 5.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: Radius1 > 0

▷ **Radius2** (input_control) . . . . . number(-array) ⇝ *real* / integer
Shorter radius.
Default: 60.0
Suggested values: Radius2 ∈ {1.0, 2.0, 4.0, 5.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: Radius2 > 0 && Radius2 <= Radius1

▷ **StartAngle** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
Start angle of the sector.
Default: 0.0
Suggested values: StartAngle ∈ {0.0, 0.785398, 1.5708, 2.35619, 3.14159, 3.92699, 4.71239, 5.49779, 6.28318}
Value range: 0 ≤ StartAngle ≤ 6.28318 (lin)

▷ **EndAngle** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
End angle of the sector.
Default: 3.14159
Suggested values: EndAngle ∈ {0.0, 0.785398, 1.5708, 2.35619, 3.14159, 3.92699, 4.71239, 5.49779, 6.28318}
Value range: 0 ≤ EndAngle ≤ 6.28318 (lin)

Example

```
read_image (Image, 'fabrik')
gen_ellipse_sector (EllipseSector, 200, 200, rad(30),\
                    150, 90, rad(45), rad(280))
reduce_domain (Image, EllipseSector, ImageReduced)
dev_clear_window ()
dev_display (ImageReduced)
```

Complexity

Runtime complexity: $O(\texttt{Radius1} * 2)$

Storage complexity (byte): $O(\texttt{Radius1} * 8)$

Result

If the parameter values are correct, the operator gen_ellipse returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system ('clip_region',<'true'/'false'>).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

paint_region, reduce_domain

Alternatives

gen_circle, gen_circle_sector, gen_region_polygon_filled, draw_ellipse, gen_ellipse

See also

disp_ellipse, set_shape, smallest_circle, reduce_domain

Module

Foundation

gen_empty_region (: EmptyRegion : :)

Create an empty region.

The operator gen_empty_region creates an empty region. This means that the output parameter contains an object. Thus, count_obj returns 1. The area of the region is 0. Most of the shape features are undefined (0). It should be noted that an empty region must not be confused with the empty tuple.

Parameters

▷ **EmptyRegion** (output_object) ..... region ⇝ *object*
Empty region (no pixels).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

gen_grid_region (: RegionGrid : RowSteps, ColumnSteps, Type, Width, Height :)

Create a region from lines or pixels.

The operator gen_grid_region creates a grid constructed of lines (Type = 'lines') or pixels (Type = 'points'). In case of 'lines' continuous lines are returned, in case of 'points' only the intersections of the lines. Starting from the pixel (0,0) to the pixel (Height-1,Width-1) the grid is built up at stepping width RowSteps in row direction and ColumnSteps in column direction. The step widths have to be greater than *1*. Exceptions are in 'lines' mode RowSteps, where RowSteps or ColumnSteps can be set equal to *0*. In such a case, only columns, rows respectively, are created.

Attention

If a very small pattern is chosen (RowSteps $<$ 4 or ColumnSteps $<$ 4) the created region requires much storage.

In the 'points' mode RowSteps and ColumnSteps must not be set to zero.

Parameters

▷ **RegionGrid** (output_object) . . . . . region ⇝ *object*
Created lines/pixel region.

▷ **RowSteps** (input_control) . . . . . extent.y ⇝ *integer* / real
Step width in line direction or zero.
Default: 10
Suggested values: RowSteps ∈ {0, 2, 3, 4, 5, 7, 10, 15, 20, 30, 50, 100}
(lin)
Minimum increment: 1
Recommended increment: 10
Restriction: RowSteps > 1 || RowSteps == 0

▷ **ColumnSteps** (input_control) . . . . . extent.x ⇝ *integer* / real
Step width in column direction or zero.
Default: 10
Suggested values: ColumnSteps ∈ {0, 2, 3, 4, 5, 7, 10, 15, 20, 30, 50, 100}
(lin)
Minimum increment: 1
Recommended increment: 10
Restriction: ColumnSteps > 1 || ColumnSteps == 0

▷ **Type** (input_control) . . . . . string ⇝ *string*
Type of created pattern.
Default: 'lines'
List of values: Type ∈ {'lines', 'points'}

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Maximum width of pattern.
Default: 512
Suggested values: Width ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Maximum height of pattern.
Default: 512
Suggested values: Height ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

Example

```
read_image(Image,'fabrik')
gen_grid_region(Raster,10,10,'lines',512,512)
reduce_domain(Image,Raster,Mask)
sobel_amp(Mask,GridSobel,'sum_abs',3)
dev_display(GridSobel)
```

Complexity

The necessary storage (in bytes) for the region is:

$O((ImageWidth/ColumnSteps) * (ImageHeight/RowSteps))$

Result

If the parameter values are correct the operator gen_grid_region returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

reduce_domain, paint_region

Alternatives

gen_region_line, gen_region_polygon, gen_region_points, gen_region_runs

See also

gen_checker_region, reduce_domain

Module

Foundation

gen_random_region (: RegionRandom : Width, Height :)

Create a random region.

The operator gen_random_region returns a random region. During this process every pixel in the image area [0...Width-1] [0...Height-1] is adapted into the region with the probability 0.5. The created region can be imagined as the threshold formation in an image with noise.

This procedure is particularly important for the creation of uncorrelated binary patterns.

The random pattern is generated using the C function "nrand48()". See the parameter *'seed_rand'* of set_system for information on the used random seed.

Attention

If Width and Height are chosen large (> 100) the created region may require much storage space due to the internally used runlength coding. The gray values of the output region are undefined.

Parameters

▷ **RegionRandom** (output_object) ..... region ⇝ *object*
Created random region with expansion Width x Height.

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Maximum horizontal expansion of random region.
Default: 128
Suggested values: Width ∈ {16, 32, 50, 64, 100, 128, 256, 300, 400, 512}
(lin)
Minimum increment: 1
Recommended increment: 10
Restriction: Width > 0

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Maximum vertical expansion of random region.
Default: 128
Suggested values: Height ∈ {16, 32, 50, 64, 100, 128, 256, 300, 400, 512}
(lin)
Minimum increment: 1
Recommended increment: 10
Restriction: Height > 0

Complexity

The worst case for the storage complexity for the created region (in byte) is: $O(Width * Height * 2)$.

Result

If the parameter values are correct, the operator gen_random_region returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

paint_region, reduce_domain

See also

gen_checker_region, add_noise_distribution, add_noise_white, reduce_domain

Module

Foundation

gen_random_regions (: Regions : Type, WidthMin, WidthMax, HeightMin, HeightMax, PhiMin, PhiMax, NumRegions, Width, Height :)

Create random regions like circles, rectangles and ellipses.

The operator gen_random_regions generates regions, whose parameters are determined at random within the given limits, and returns them in Regions.

The parameter NumRegions determines how many regions are created.

The position is always random and cannot be determined by parameters, but the center of every region lies in the pixel area [0...Width-1] [0...Height-1].

The parameter Type defines the type of the regions. The meaning of the lower and upper limits depends on Type:

- *'rectangle1'*: Minimum/maximum width/height.
  PhiMin and PhiMax are without further significance.
- *'rectangle2'*: Minimum/maximum half edge lengths, minimum/maximum rotation angle. Thereby the larger value is used for the first half edge length. The parameters correspond to the parameters Length1, Length2, and Phi of gen_rectangle2.
- *'circle'*: WidthMin and WidthMax determine the minimum and maximum radii. The parameters correspond to the parameter Radius in gen_circle.
  HeightMin and HeightMax as well as PhiMin and PhiMax are without further significance.
- *'ellipse'*: Minimum/maximum half-radii and minimal/maximum rotation angle. Thereby the larger value is used for the first half-radius. The parameters correspond to the parameters Radius1, Radius2, and Phi in gen_ellipse.
- *'ring'*: Minimum/maximum radii, whereby the larger one determines the radius of the larger circle and the smaller one the radius of the hole.
  PhiMin and PhiMax are without further significance.

The random parameters are generated using the C function "nrand48()". See the parameter *'seed_rand'* of set_system for information on the used random seed.

Parameters

▷ **Regions** (output_object) ..... region-array ⇝ *object*
Created regions.

▷ **Type** (input_control) ..... string ⇝ *string*
Type of regions to be created.
Default: 'circle'
List of values: Type ∈ {'circle', 'ring', 'ellipse', 'rectangle1', 'rectangle2'}

▷ **WidthMin** (input_control) ..... number ⇝ *real* / integer
Minimum object characteristic, depending on type and value.
Default: 10.0
Suggested values: WidthMin ∈ {1.0, 3.0, 5.0, 10.0, 20.0, 40.0, 80.0}
Value range: 0 ≤ WidthMin (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **WidthMax** (input_control) . . . . . number ⇝ *real* / integer
Maximum object characteristic, depending on type and value.
Default: 20.0
Suggested values: WidthMax ∈ {1.0, 3.0, 5.0, 10.0, 20.0, 40.0, 80.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: WidthMin <= WidthMax

▷ **HeightMin** (input_control) . . . . . number ⇝ *real* / integer
Minimum object characteristic, depending on type and value.
Default: 10.0
Suggested values: HeightMin ∈ {1.0, 3.0, 5.0, 10.0, 20.0, 40.0, 80.0}
Value range: 0 ≤ HeightMin (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **HeightMax** (input_control) . . . . . number ⇝ *real* / integer
Maximum object characteristic, depending on type and value.
Default: 30.0
Suggested values: HeightMax ∈ {1.0, 3.0, 5.0, 10.0, 20.0, 40.0, 80.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: HeightMin <= HeightMax

▷ **PhiMin** (input_control) . . . . . number ⇝ *real* / integer
Minimum rotation angle of the region.
Default: -0.7854
Suggested values: PhiMin ∈ {0.0, 0.1, 0.3, 0.6, 0.9, 1.2, 1.5}
Value range: 0.0 ≤ PhiMin ≤ 6.28 (lin)
Restriction: PhiMin > 0

▷ **PhiMax** (input_control) . . . . . number ⇝ *real* / integer
Maximum rotation angle of the region.
Default: 0.7854
Suggested values: PhiMax ∈ {0.0, 0.1, 0.3, 0.6, 0.9, 1.2, 1.5}
Value range: 0.0 ≤ PhiMax ≤ 6.28 (lin)
Restriction: PhiMax > 0

▷ **NumRegions** (input_control) . . . . . integer ⇝ *integer*
Number of regions.
Default: 100
Suggested values: NumRegions ∈ {1, 5, 20, 100, 200, 500, 1000, 2000}
Value range: 1 ≤ NumRegions (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Width** (input_control) . . . . . integer ⇝ *integer*
Maximum horizontal expansion of the centers.
Default: 512
Suggested values: Width ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Width (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Height** (input_control) . . . . . integer ⇝ *integer*
Maximum vertical expansion of the centers.
Default: 512
Suggested values: Height ∈ {128, 256, 512, 1024}
Value range: 1 ≤ Height (lin)
Minimum increment: 1
Recommended increment: 10

Result

If the parameter values are correct gen_random_regions returns the value 2 (H_MSG_TRUE). Otherwise

an exception is raised. The clipping according to the current image format is determined by the operator `set_system('clip_region',<'true'/'false'>)`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`paint_region`

Module

Foundation

gen_rectangle1 (: Rectangle : Row1, Column1, Row2, Column2 :)

Create a rectangle parallel to the coordinate axes.

The operator `gen_rectangle1` generates one or more rectangles parallel to the coordinate axes which are described by the upper left corner (`Row1`, `Column1`) and the lower right corner (`Row2`, `Column2`). More than one region can be created by passing a tuple of corner points. The coordinate system runs from (0,0) (upper left corner) to (Width-1,Height-1). See `get_system` and `reset_obj_db` in this context.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

▷ **Rectangle** (output_object) . . . . . region(-array) ⇝ *object*
Created rectangle.

▷ **Row1** (input_control) . . . . . rectangle.origin.y(-array) ⇝ *real* / integer
Line of upper left corner point.
Default: 30.0
Suggested values: `Row1` ∈ {0.0, 10.0, 20.0, 50.0, 100.0, 200.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Column1** (input_control) . . . . . rectangle.origin.x(-array) ⇝ *real* / integer
Column of upper left corner point.
Default: 20.0
Suggested values: `Column1` ∈ {0.0, 10.0, 20.0, 50.0, 100.0, 200.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Row2** (input_control) . . . . . rectangle.corner.y(-array) ⇝ *real* / integer
Line of lower right corner point.
Default: 100.0
Suggested values: `Row2` ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0, 511.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: Row2 >= Row1

▷ **Column2** (input_control) . . . . . rectangle.corner.x(-array) ⇝ *real* / integer
Column of lower right corner point.
Default: 200.0
Suggested values: Column2 ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0, 511.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0
Restriction: Column2 >= Column1

Example

```
* Contrast improvement in a rectangular region of interest
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
read_image(Image,'mreut')
dev_display (Image)
draw_rectangle1(WindowHandle,Row1,Column1,Row2,Column2)
gen_rectangle1(Rectangle,Row1,Column1,Row2,Column2)
reduce_domain(Image,Rectangle,Mask)
emphasize(Mask,Emphasize,9,9,1.0)
dev_display (Emphasize)
```

Result

If the parameter values are correct, the operator gen_rectangle1 returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

paint_region, reduce_domain

Alternatives

gen_rectangle2, gen_region_polygon, fill_up, gen_region_runs, gen_region_points, gen_region_line

See also

draw_rectangle1, reduce_domain, smallest_rectangle1

Module

Foundation

gen_rectangle2 (: Rectangle : Row, Column, Phi, Length1, Length2 :)

Create a rectangle of any orientation.

The operator gen_rectangle2 generates one or more rectangles with the center (Row, Column) , the orientation Phi and the half edge lengths Length1 and Length2. The orientation is given in arc measure and indicates the angle between the horizontal axis and the edge (with Length1) (mathematically positive). The coordinate system runs from (0,0) (upper left corner) to (Width-1,Height-1). See get_system and reset_obj_db in this context. More than one region can be created by passing one tuple of corner points.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Attention

The gray values of the output objects are undefined. For speed reasons, the resulting region may contain additional

pixels at the border and some individual pixels at the border may be missing. This may lead to an inconsistency between the operators smallest_rectangle2 and gen_rectangle2.

Parameters

▷ **Rectangle** (output_object) . . . . . region(-array) ⇝ *object*
Created rectangle.

▷ **Row** (input_control) . . . . . rectangle2.center.y(-array) ⇝ *real* / integer
Line index of the center.
Default: 300.0
Suggested values: Row ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Column** (input_control) . . . . . rectangle2.center.x(-array) ⇝ *real* / integer
Column index of the center.
Default: 200.0
Suggested values: Column ∈ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Phi** (input_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real* / integer
Angle of the first edge to the horizontal (in radians).
Default: 0.0
Suggested values: Phi ∈ {-1.178097, -0.785398, -0.392699, 0.0, 0.392699, 0.785398, 1.178097}
(lin)
Restriction: - pi / 2 < Phi && Phi <= pi / 2

▷ **Length1** (input_control) . . . . . rectangle2.hwidth(-array) ⇝ *real* / integer
Half width.
Default: 100.0
Suggested values: Length1 ∈ {3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0, 300.0, 500.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Length2** (input_control) . . . . . rectangle2.hheight(-array) ⇝ *real* / integer
Half height.
Default: 20.0
Suggested values: Length2 ∈ {1.0, 2.0, 3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0}
(lin)
Minimum increment: 1.0
Recommended increment: 10.0

Result

If the parameter values are correct the operator gen_rectangle2 returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

paint_region, reduce_domain

Alternatives

gen_rectangle1, gen_region_polygon_filled, gen_region_polygon, gen_region_points, fill_up

See also

draw_rectangle2, reduce_domain, smallest_rectangle2, gen_ellipse

Module

Foundation

gen_region_contour_xld (Contour : Region : Mode :)

Create a region from an XLD contour.

gen_region_contour_xld creates a region Region from a subpixel XLD contour Contour. The contour is sampled according to the Bresenham algorithm and influenced by the parameter *'neighborhood'* of the operator set_system. Open contours are closed before converting them to regions. Finally, the parameter Mode defines whether the region is filled up (*'filled'*) or returned by its contour (*'margin'*).

Please note that the coordinates of the contour points are rounded to their nearest integer pixel coordinates during the conversion. This may lead to unexpected results when passing the contour obtained by the operator gen_contour_region_xld to gen_region_contour_xld: When setting Mode of gen_contour_region_xld to *'border'*, the input region of gen_contour_region_xld and the output region of gen_region_contour_xld differ. For example, let us assume that the input region of gen_contour_region_xld consists of the single pixel (1,1). Then, the resulting contour that is obtained when calling gen_contour_region_xld with Mode set to *'border'* consists of the five points (0.5,0.5), (0.5,1.5), (1.5,1.5), (1.5,0.5), and (0.5,0.5). Consequently, when passing this contour again to gen_region_contour_xld, the resulting region consists of the points (1,1), (1,2), (2,2), and (2,1).

Parameters

▷ **Contour** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contour(s).

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Created region(s).

▷ **Mode** (input_control) ..... string ⇝ *string*
Fill mode of the region(s).
Default: 'filled'
Suggested values: Mode ∈ {'filled', 'margin'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contour_polygon_xld, gen_contour_polygon_rounded_xld

Alternatives

gen_region_polygon, gen_region_polygon_xld

See also

set_system

Module

Foundation

gen_region_histo (: Region : Histogram, Row, Column, Scale :)

Convert a histogram into a region.

gen_region_histo converts a histogram created with gray_histo, gray_histo_range ,or tuple_histo_range into a region. Row and Column define the position of the center of the graphic. Scale allows scaling of the graphic, whereby *1* means displaying all 256 values, *2* means displaying 128 values, *3* means displaying only 64 values, etc.

Parameters

▷ **Region** (output_object) . . . . . region ⇝ *object*
Region containing the histogram.

▷ **Histogram** (input_control) . . . . . histogram-array ⇝ *integer*
Input histogram.

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row coordinate of the center of the histogram.
Default: 255
Suggested values: Row ∈ {100, 200, 255, 300, 400}
Value range: 0 ≤ Row ≤ 511

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column coordinate of the center of the histogram.
Default: 255
Suggested values: Column ∈ {100, 200, 255, 300, 400}
Value range: 0 ≤ Column ≤ 511

▷ **Scale** (input_control) . . . . . integer ⇝ *integer*
Scale factor for the histogram.
Default: 1
Suggested values: Scale ∈ {1, 2, 3, 4, 5, 6, 7}
Value range: 1 ≤ Scale ≤ 10 (lin)
Minimum increment: 1
Recommended increment: 1

Result

gen_region_histo returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gray_histo, gray_histo_range, tuple_histo_range

See also

disp_channel

Module

Foundation

gen_region_hline (: Regions : Orientation, Distance :)

Store input lines described in Hesse normal form as regions.

The operator gen_region_hline stores the lines described in Hesse normal form as regions. A line is determined by the distance from the line to the origin (Distance, corresponds to the length of the normal vector) and the direction of the normal vector (Orientation, corresponds to the orientation of the line $\pm\pi/2$). The directions were defined in such a way that at Orientation = 0 the normal vector lies in the direction of the X axis, which corresponds to a vertical line. At Orientation = $\pi/2$ the normal vector points in the direction of the Y axis, i.e. a horizontal line is described.

Attention

The lines are clipped to the current maximum image format.

Parameters

▷ **Regions** (output_object) . . . . . region(-array) ⇝ *object*
Created regions (one for every line), clipped to maximum image format.
Number of elements: Regions == Distance

▷ **Orientation** (input_control) . . . . . hesseline.angle.rad(-array) ⇝ *real* / integer
Orientation of the normal vector in radians.
Number of elements: Orientation == Distance
Default: 0.0
Suggested values: Orientation ∈ {-0.78, 0.0, 0.78, 1.57}
(lin)
Recommended increment: 0.02

▷ **Distance** (input_control) . . . . . hesseline.distance(-array) ⇝ *real* / integer
Distance from the line to the coordinate origin (0.0).
Default: 200
Suggested values: Distance ∈ {10, 50, 100, 200, 300, 400}
(lin)
Recommended increment: 1

Result

The operator gen_region_hline always returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

gen_region_line

See also

hough_lines

Module

Foundation

gen_region_line (: RegionLines : BeginRow, BeginCol, EndRow, EndCol :)

Store input lines as regions.

The operator gen_region_line stores the given lines (with starting point [BeginRow,BeginCol] and ending point [EndRow, EndCol]) as region.

Parameters

▷ **RegionLines** (output_object) . . . . . region(-array) ⇝ *object*
Created regions.

▷ **BeginRow** (input_control) . . . . . line.begin.y(-array) ⇝ *integer* / real
Line coordinates of the starting points of the input lines.
Default: 100
Suggested values: BeginRow ∈ {10, 50, 100, 200, 300, 400}
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **BeginCol** (input_control) . . . . . line.begin.x(-array) ⇝ *integer* / real
Column coordinates of the starting points of the input lines.
Default: 50
Suggested values: BeginCol ∈ {10, 50, 100, 200, 300, 400}
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **EndRow** (input_control) . . . . . line.end.y(-array) ⇝ *integer* / real
Line coordinates of the ending points of the input lines.
Default: 150
Suggested values: EndRow ∈ {50, 100, 200, 300, 400, 500}
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **EndCol** (input_control) . . . . . line.end.x(-array) ⇝ *integer* / real
Column coordinates of the ending points of the input lines.
Default: 250
Suggested values: EndCol ∈ {50, 100, 200, 300, 400, 500}
(lin)
Minimum increment: 1
Recommended increment: 1

Result

The operator gen_region_line always returns the value 2 (H_MSG_TRUE). The clipping according to the current image format is determined by the operator set_system('clip_region',<'true'/'false'>).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

split_skeleton_lines

Alternatives

gen_region_hline

Module

Foundation

gen_region_points (: Region : Rows, Columns :)

Store individual pixels as image region.

The operator gen_region_points creates a region described by a number of pixels. The pixels do not have to be stored in a fixed order, but the best runtime behavior is obtained when the pixels are stored in ascending order. The order is as follows:

$$(l_1, c_1) \leq (l_2, c_2) := (l_1 < l_2) \quad \vee \quad (l_1 = l_2) \wedge (c_1 \leq c_2)$$

The indicated coordinates stand for two consecutive pixels in the tuple.

Parameters

▷ **Region** (output_object) . . . . . region ⇝ *object*
Created region.

▷ **Rows** (input_control) . . . . . coordinates.y(-array) ⇝ *integer* / real
Lines of the pixels in the region.
Default: 100
Suggested values: Rows ∈ {0, 10, 30, 50, 100, 200, 300, 500}
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Columns** (input_control) . . . . . coordinates.x(-array) ⇝ *integer / real*
Columns of the pixels in the region.
Number of elements: Columns == Rows
Default: 100
Suggested values: Columns ∈ {0, 10, 30, 50, 100, 200, 300, 500}
(lin)
Minimum increment: 1
Recommended increment: 1

Complexity

F shall be the number of pixels. If the pixels are sorted in ascending order the runtime complexity is: $O(F)$, otherwise $O(log(F) * F)$.

Result

The operator gen_region_points returns the value 2 (H_MSG_TRUE) if the pixels are located within the image format. Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>). If an empty region is created (by the clipping or by an empty input) the operator set_system('store_empty_region',<'true'/'false'>) determines whether the region is returned or an empty object tuple.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

get_region_points

Possible Successors

paint_region, reduce_domain

Alternatives

gen_region_polygon, gen_region_runs, gen_region_line

See also

reduce_domain

Module

Foundation

gen_region_polygon (: Region : Rows, Columns :)

Store a polygon as a region.

The operator gen_region_polygon creates a region from a polygon row described by a series of line and column coordinates. The created region consists of the pixels of the routes defined thereby, wherein it is linearly interpolated between the base points.

Attention

The region is not automatically closed and not filled.

Parameters

▷ **Region** (output_object) . . . . . region ⇝ *object*
Created region.

▷ **Rows** (input_control) . . . . . polygon.y-array ⇝ *integer*
Line indices of the base points of the region contour.
Default: 100
Suggested values: Rows ∈ {0, 10, 30, 50, 100, 200, 300, 500}
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Columns** (input_control) . . . . . polygon.x-array ⇝ *integer*
Column indices of the base points of the region contour.
Number of elements: Columns == Rows
Default: 100
Suggested values: Columns ∈ {0, 10, 30, 50, 100, 200, 300, 500}
(lin)
Minimum increment: 1
Recommended increment: 1

Example

```
* Polygon-approximation
get_region_polygon(Region,7,Row,Column)
* store it as a region
gen_region_polygon(Pol,Row,Column)
* fill up the hole
fill_up(Pol,Filled)
```

Result

If the base points are correct the operator gen_region_polygon returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>). If an empty region is created (by the clipping or by an empty input) the operator set_system('store_empty_region',<'true'/'false'>) determines whether the region is returned or an empty object tuple.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

get_region_polygon, draw_polygon

Alternatives

gen_region_polygon_filled, gen_region_points, gen_region_runs

See also

fill_up, reduce_domain, get_region_polygon, draw_polygon

Module

Foundation

gen_region_polygon_filled (: Region : Rows, Columns :)

Store a polygon as a "filled" region.

The operator gen_region_polygon_filled creates a region from a polygon containing the corner points of the region (line and column coordinates) either clockwise or anti-clockwise. Contrary to gen_region_polygon a "filled" region is returned here. Note that for subpixel coordinates Rows and Columns the coordinates are rounded before the polygon containing the corner points is determined. Therefore there might be pixels in the Region which do not lie on the orignial (subpixel) polygon. An simple alternative in the case of polygons without holes would be to use the operators gen_region_polygon and fill_up.

Parameters

▷ **Region** (output_object) . . . . . region ⇝ *object*
Created region.

▷ **Rows** (input_control) . . . . . polygon.y-array ⇝ *integer* / real
Line indices of the base points of the region contour.
Default: 100
Suggested values: Rows ∈ {0, 10, 30, 50, 100, 200, 300, 500}
(lin)
Minimum increment: 1
Recommended increment: 1

▷ **Columns** (input_control) . . . . . polygon.x-array ⇝ *integer* / real
Column indices of the base points of the region contour.
Number of elements: Columns == Rows
Default: 100
Suggested values: Columns ∈ {0, 10, 30, 50, 100, 200, 300, 500}
(lin)
Minimum increment: 1
Recommended increment: 1

Example

```
/* Polygon approximation */
T_get_region_polygon(Region,7,&Row,&Column);
T_gen_region_polygon_filled(&Pol,Row,Column);
/* fill up with original gray value */
reduce_domain(Image,Pol,&New);
```

Result

If the base points are correct the operator gen_region_polygon_filled returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system('clip_region',<'true'/'false'>). If an empty region is created (by the clipping or by an empty input) the operator set_system('store_empty_region',<'true'/'false'>) determines whether the region is returned or an empty object tuple.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_region_polygon, draw_polygon

Alternatives

gen_region_polygon, gen_region_points, draw_polygon

See also

gen_region_polygon, reduce_domain, get_region_polygon, gen_region_runs

Module

Foundation

gen_region_polygon_xld (Polygon : Region : Mode :)

Create a region from an XLD polygon.

gen_region_polygon_xld creates a region Region from a subpixel XLD polygon Polygon. The polygon is sampled according to the Bresenham algorithm and influenced by the parameter *'neighborhood'* of the operator set_system. Open polygons are closed before converting them to regions. Finally, the parameter Mode defines whether the region is filled up (*'filled'*) or returned by its contour (*'margin'*).

Parameters

▷ **Polygon** (input_object) . . . . . xld_poly(-array) ⇝ *object*
Input polygon(s).

▷ **Region** (output_object) . . . . . region(-array) ⇝ *object*
Created region(s).

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Fill mode of the region(s).
Default: 'filled'
Suggested values: Mode ∈ {'filled', 'margin'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_polygons_xld

Alternatives

gen_region_polygon, gen_region_contour_xld

See also

set_system

Module

Foundation

gen_region_runs (: Region : Row, ColumnBegin, ColumnEnd :)

Create a region from a runlength coding.

The operator gen_region_runs creates a region described by the input runlength structure. The runlength representation is created by examining a region line by line with ascending line number (= from "top" to "bottom"). Every line runs through from left to right (ascending column number). All starting and ending points being stored by region segments (=runs). Thus a region can be described by a sequence of runs, a run being defined by line number as well as starting and ending points (column number).

The storing is fastest when the runs are sorted. The order is as follows:

$$(l_1, b_1, e_1) \leq (l_2, b_2, e_2) := (l_1 < l_2) \quad \vee \quad (l_1 = l_2) \wedge (b_1 \leq b_2)$$

.

Attention

For runtime reasons, it is not checked whether the restriction ColumnEnd >= ColumnBegin is fulfilled. Note that if this restriction is violated, subsequent operations will likely lead to errors or unexpected behavior.

Parameters

▷ **Region** (output_object) . . . . . region ⇝ *object*
Created region.

▷ **Row** (input_control) . . . . . chord.y(-array) ⇝ *integer*
Lines of the runs.
Default: 100
Suggested values: Row ∈ {0, 50, 100, 200, 300, 500}
(lin)
Minimum increment: 1
Recommended increment: 10

▷ **ColumnBegin** (input_control) . . . . . chord.x1(-array) ⇝ *integer*
Columns of the starting points of the runs.
Number of elements: ColumnBegin == Row
Default: 50
Suggested values: ColumnBegin ∈ {0, 50, 100, 200, 300, 500}
(lin)
Minimum increment: 1
Recommended increment: 10

▷ **ColumnEnd** (input_control) . . . . . chord.x2(-array) ⇝ *integer*
Columns of the ending points of the runs.
Number of elements: ColumnEnd == Row
Default: 200
Suggested values: ColumnEnd ∈ {50, 100, 200, 300, 500}
(lin)
Minimum increment: 1
Recommended increment: 10
Restriction: ColumnEnd >= ColumnBegin

Complexity

F shall be the number of pixels. If the pixels are sorted in ascending order the runtime complexity is: $O(F)$, otherwise it is $O(log(F) * F)$.

Result

If the data is correct the operator gen_region_runs returns the value 2 (H_MSG_TRUE), otherwise an exception is raised. The clipping according to the current image format is set via the operator set_system ('clip_region',<'true'/'false'>). If an empty region is created (by the clipping or by an empty input) the operator set_system('store_empty_region',<'true'/'false'>) determines whether the region is returned or an empty object tuple.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

get_region_runs

Alternatives

gen_region_points, gen_region_polygon, gen_region_line, gen_region_polygon_filled

See also

reduce_domain

Module

Foundation

label_to_region (LabelImage : Regions : :)

Extract regions with equal gray values from an image.

label_to_region segments an image into regions of equal gray value. One output region is generated for each gray value occurring in the image. This is similar to calling threshold multiple times, and accumulating the results with concat_obj. Another related operator is regiongrowing. However, label_to_region does not perform a connection operation on the resulting regions, i.e., they may be disconnected. A typical application of label_to_region is the segmentation of label images, hence its name.

The number of output regions is limited by the system parameter *'max_outp_obj_par'*, which can be read via get_system(::'max_outp_obj_par':<number>).

Attention

`label_to_region` is not implemented for images of type `real`. The input images must not contain negative gray values.

Parameters

▷ **`LabelImage`** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / int2 / int4 / int8
Label image.

▷ **`Regions`** (output_object) . . . . . region-array ⇝ *object*
Regions having a constant gray value.

Complexity

Let $x1$ be the minimum x-coordinate, $x2$ the maximum x-coordinate, $y1$ be the minimum y-coordinate, and $y2$ the maximum y-coordinate of a particular gray value. Furthermore, let N be the number of different gray values in the image. Then the runtime complexity is $O(N * (x2 - x1 + 1) * (y2 - y1 + 1))$

Result

`label_to_region` returns 2 (`H_MSG_TRUE`) if the gray values lie within a correct range. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`min_max_gray`, `sobel_amp`, `binomial_filter`, `gauss_filter`, `reduce_domain`, `diff_of_gauss`

Possible Successors

`connection`, `dilation1`, `erosion1`, `opening`, `closing`, `rank_region`, `shape_trans`, `skeleton`

See also

`threshold`, `concat_obj`, `regiongrowing`, `region_to_label`

Module

Foundation

23.3 Features

This chapter contains operators that allow you to access the different features of regions.

List of Features

In the following, the available features are illustrated.

'area': Area of the object

'row': Row index of the center

'column': Column index of the center

'row1': Row index of upper left corner

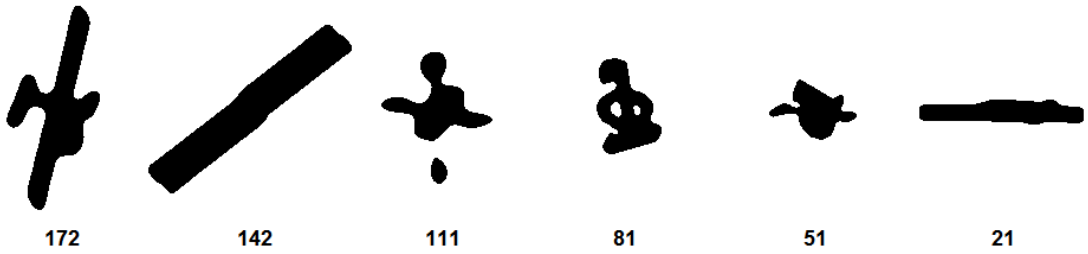
'column1': Column index of upper left corner

'row2': Row index of lower right corner

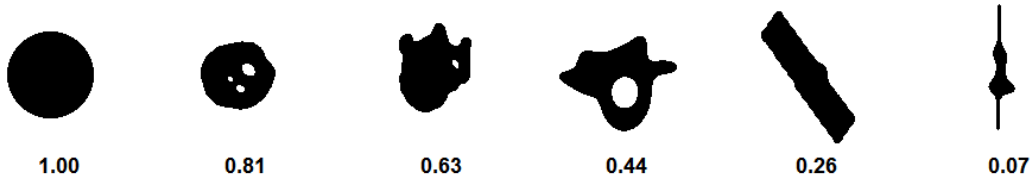
'column2': Column index of lower right corner

'width': Width of the region (parallel to the coordinate axis)

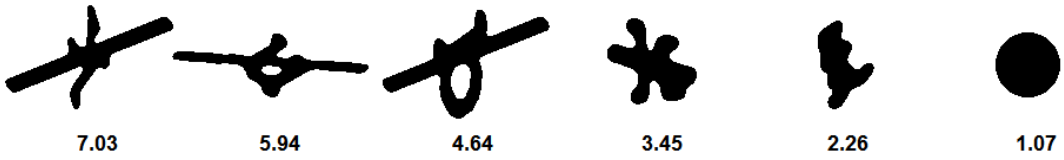
'height': Height of the region (parallel to the coordinate axis)



'circularity': Circularity (see `circularity`)

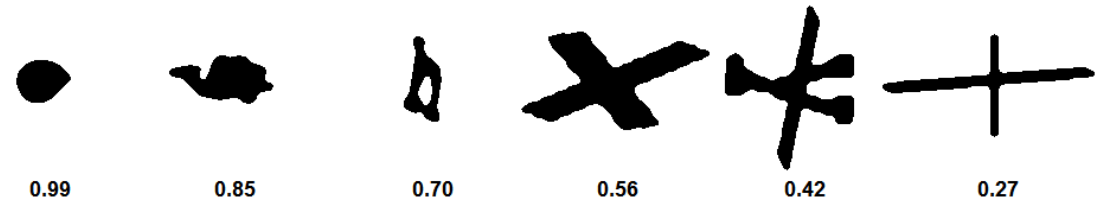


'compactness': Compactness (see `compactness`)



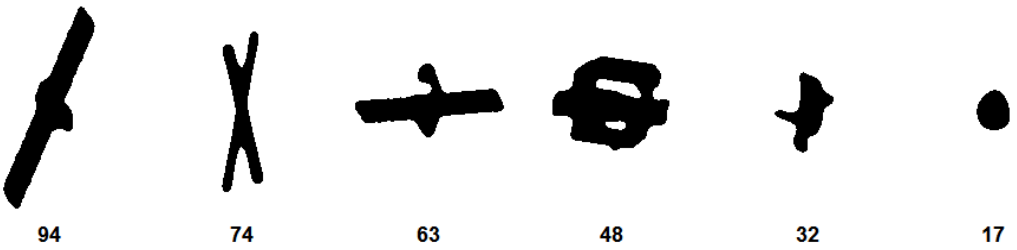
'contlength': Total length of contour (see operator `contlength`)

'convexity': Convexity (see `convexity`)

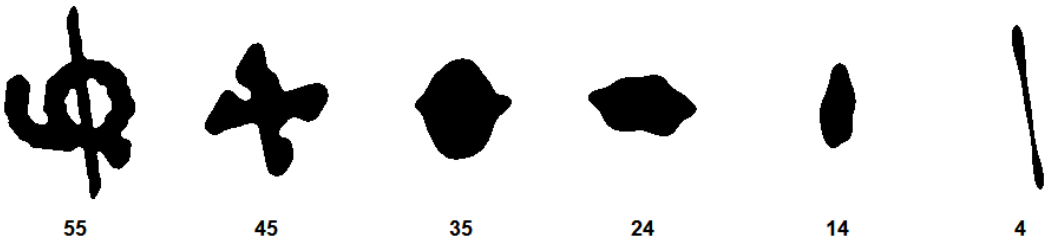


'rectangularity': Rectangularity (see `rectangularity`)

'ra': Main radius of the equivalent ellipse (see `elliptic_axis`)

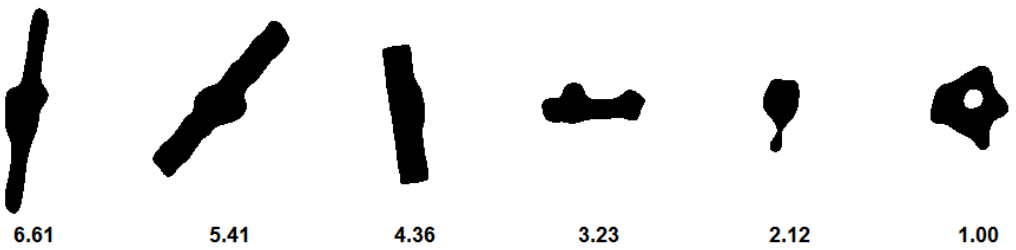


'rb': Secondary radius of the equivalent ellipse (see `elliptic_axis`)

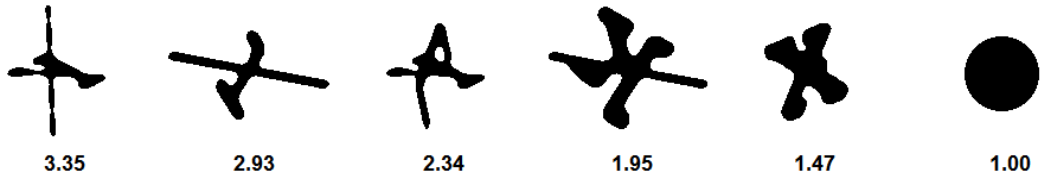


'phi': Orientation of the equivalent ellipse see `elliptic_axis`)

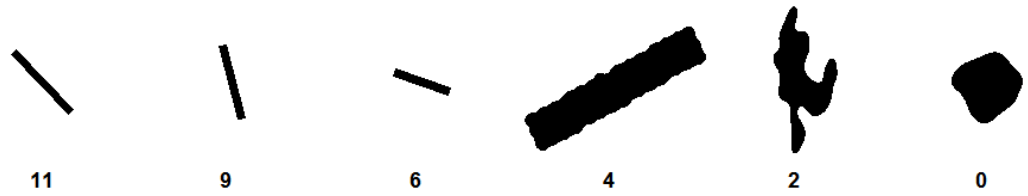
'anisometry': Anisometry (see `eccentricity`)



'bulkiness': Bulkiness (see operator `eccentricity`)



'struct_factor': Structure Factor (see operator `eccentricity`)



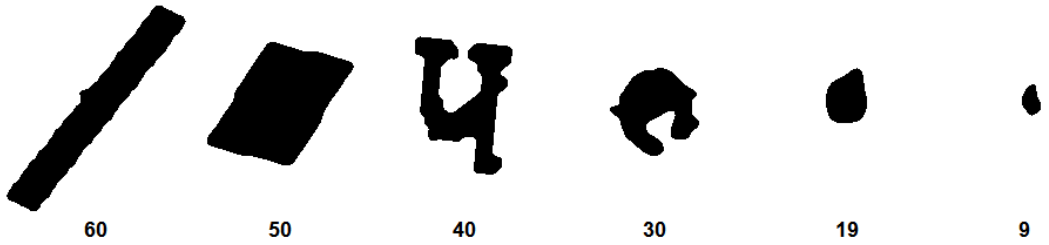
'outer_radius': Radius of smallest surrounding circle (see `smallest_circle`)

'inner_radius': Radius of largest inner circle (see `inner_circle`)

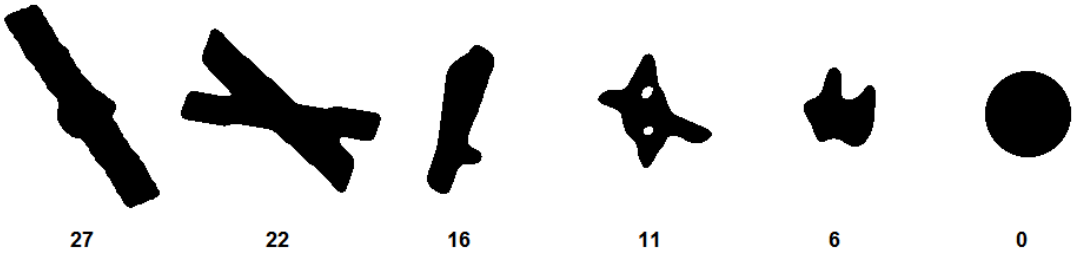
'inner_width': Width of the largest axis-parallel rectangle that fits into the region (see `inner_rectangle1`)

'inner_height': Height of the largest axis-parallel rectangle that fits into the region (see `inner_rectangle1`)

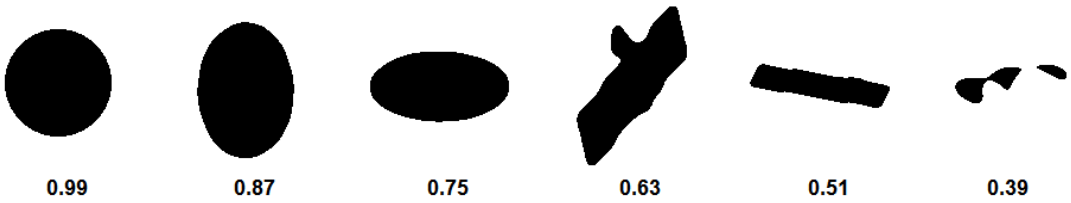
'dist_mean': Mean distance from the region border to the center (see operator `roundness`)



'dist_deviation': Deviation of the distance from the region border to the center (see operator `roundness`)



'roundness': Roundness (see operator `roundness`)



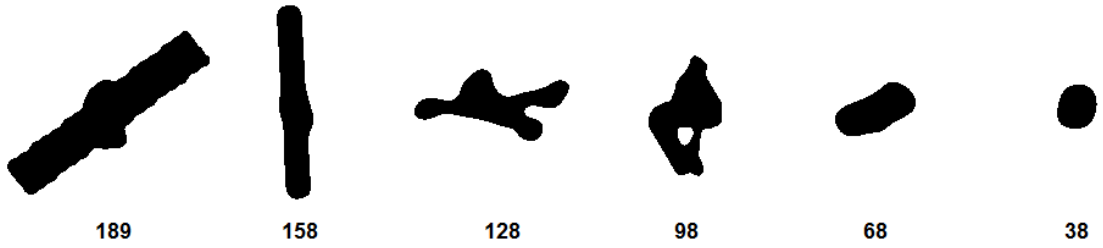
'num_sides': Number of polygon sides (see operator `roundness`)

'connect_num': Number of connection components (see operator `connect_and_holes`)

'holes_num': Number of holes (see operator `connect_and_holes`)

'area_holes': Area of the holes of the object (see operator `area_holes`)

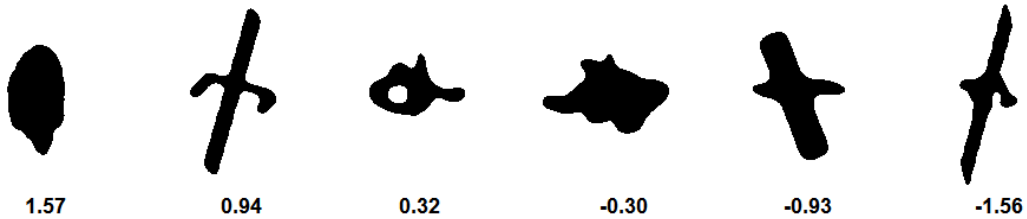
'max_diameter': Maximum diameter of the region (see operator `diameter_region`)



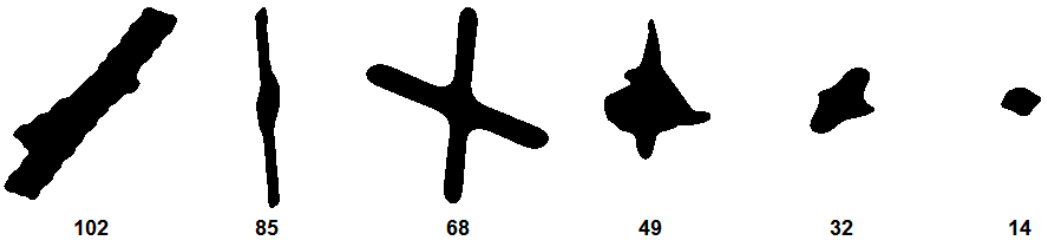
'orientation': Orientation of the region (see operator `orientation_region`)

'euler_number': Euler number (see operator `euler_number`)

'rect2_phi': Orientation of the smallest surrounding rectangle (see operator `smallest_rectangle2`)



'rect2_len1': Half the length of the smallest surrounding rectangle (see operator `smallest_rectangle2`)



'rect2_len2': Half the width of the smallest surrounding rectangle (see operator smallest_rectangle2)

'moments_m11': Geometric moments of the region (see operator moments_region_2nd)

'moments_m20': Geometric moments of the region (see operator moments_region_2nd)

'moments_m02': Geometric moments of the region (see operator moments_region_2nd)

'moments_ia': Geometric moments of the region (see operator moments_region_2nd)

'moments_ib': Geometric moments of the region (see operator moments_region_2nd)

'moments_m11_invar': Geometric moments of the region (see operator moments_region_2nd_invar)

'moments_m20_invar': Geometric moments of the region (see operator moments_region_2nd_invar)

'moments_m02_invar': Geometric moments of the region (see operator moments_region_2nd_invar)

'moments_phi1': Geometric moments of the region (see operator moments_region_2nd_rel_invar)

'moments_phi2': Geometric moments of the region (see operator moments_region_2nd_rel_invar)

'moments_m21': Geometric moments of the region (see operator moments_region_3rd)

'moments_m12': Geometric moments of the region (see operator moments_region_3rd)

'moments_m03': Geometric moments of the region (see operator moments_region_3rd)

'moments_m30': Geometric moments of the region (see operator moments_region_3rd)

'moments_m21_invar': Geometric moments of the region (see operator moments_region_3rd_invar)

'moments_m12_invar': Geometric moments of the region (see operator moments_region_3rd_invar)

'moments_m03_invar': Geometric moments of the region (see operator moments_region_3rd_invar)

'moments_m30_invar': Geometric moments of the region (see operator moments_region_3rd_invar)

'moments_i1': Geometric moments of the region (see operator moments_region_central)

'moments_i2': Geometric moments of the region (see operator moments_region_central)

'moments_i3': Geometric moments of the region (see operator moments_region_central)

'moments_i4': Geometric moments of the region (see operator moments_region_central)

'moments_psi1': Geometric moments of the region (see operator moments_region_central_invar)

'moments_psi2': Geometric moments of the region (see operator moments_region_central_invar)

'moments_psi3': Geometric moments of the region (see operator moments_region_central_invar)

'moments_psi4': Geometric moments of the region (see operator moments_region_central_invar)

area_center (Regions : : : Area, Row, Column)

Area and center of regions.

The operator area_center calculates the area and the center of the input regions. The area is defined as the number of pixels of a region. The center is calculated as the mean value of the line or column coordinates, respectively, of all pixels.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their area.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of the input region. In case of empty region all parameters have the value 0.0 if no other behavior was set (see set_system).

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Region(s) to be examined.
- ▷ **Area** (output_control) ..... integer(-array) ⇝ *integer*
  Area of the region.
- ▷ **Row** (output_control) ..... point.y(-array) ⇝ *real*
  Line index of the center.
- ▷ **Column** (output_control) ..... point.x(-array) ⇝ *real*
  Column index of the center.

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

main()
{
  Tuple   area, row, column;

  HImage   img ("monkey");
  HWindow  w;

  img.Display (w);
  w.Click ();

  HRegionArray   reg = (img >= 164).Connection ();

  reg.Display (w);
  w.Click ();

  area = reg.AreaCenter (&row, &column);

  for (int i = 0; i < reg.Num (); i++)
  {
    cout << "Row    [" << i << "]" << "= " << row[i].D ();
    cout << "\t\tColumn [" << i << "]" << "= " << column[i].D () << endl;
  }

  cout << "Total number of regions: " << reg.Num () << endl;
  return(0);
}
```

Complexity

If F is the area of a region the mean runtime complexity is $O(sqrt(F))$.

Result

The operator `area_center` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

See also

`select_shape`

Module

Foundation

area_holes (Regions : : : Area)

Compute the area of holes of regions.

The operator `area_holes` calculates the area of the holes in the input regions. The area is defined as the number of pixels contained in the holes. If a region has more than one hole the sum of the areas of all holes in this region are returned. The neighborhood type is set via `set_system('neighborhood',<4/8>)` (default: 8-neighborhood).

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in the area of their holes.

If more than one region is passed the results are stored in tuples with the index of a value in the tuple corresponding to the index of the input region. In case of an empty region the area has the value 0.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Area** (output_control) ..... integer(-array) ⇝ *integer*
Area(s) of holes of the region(s).

Example

```
read_image(Image,'modules/modules_01')
threshold(Image,Region,50,250)
area_holes(Region,Area)
```

Result

The operator `area_holes` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, regiongrowing, connection

See also

area_center, fill_up

Module

Foundation

circularity (Regions : : : Circularity)

Shape factor for the circularity (similarity to a circle) of a region.

The operator circularity calculates the similarity of the input region with a circle.

Calculation: If F is the area of the region and max is the maximum distance from the center to all contour pixels, the shape factor C is defined as:

$$C' = \frac{F}{(max^2 * \pi)}$$

$$C = \min(1.0, C')$$

The shape factor C of a circle is 1. If the region is long or has holes, C is smaller than 1. The operator circularity especially responds to large bulges, holes and unconnected regions. The value of C is clipped to 1.0, because the pixel area of a region can only be an approximation of a real circle's area. This approximation error is bigger for small regions than for large regions.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their circularity.

In case of an empty region the operator circularity returns the value 0 (if no other behavior was set (see set_system)). If more than one region is passed the numerical values of the shape factor are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Circularity** (output_control) ..... real(-array) ⇝ *real*
Circularity of the input region(s).
Assertion: 0 <= Circularity && Circularity <= 1.0

Example

```
* Comparison between shape factors of rectangle, circle and ellipse:
gen_rectangle1(R1,10,10,20,20)
gen_rectangle2(R2,100,100,0.0,100,20)
gen_ellipse(E100,100,100,0.0,100,20)
gen_circle(C,100,100,20)
circularity(R1,M_R1)
circularity(R2,M_R2)
circularity(E100,M_E)
circularity(C,M_C)
fwrite_string(FileId,['quadrate:  ',M_R1])
fnew_line(FileId)
fwrite_string(FileId,['rectangle: ',M_R2])
fnew_line(FileId)
fwrite_string(FileId,['ellipse:   ',M_E])
fnew_line(FileId)
fwrite_string(FileId,['circle:    ',M_C])
fnew_line(FileId)
```

Result

The operator circularity returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

roundness, compactness, convexity, eccentricity

See also

area_center, select_shape

Module

Foundation

compactness (Regions : : : Compactness)

Shape factor for the compactness of a region.

The operator compactness calculates the compactness of the input regions.

Calculation: If L is the length of the contour (see contlength) and F the area of the region the shape factor C is defined as:

$$C' = \frac{L^2}{4F\pi}$$

$$C = \max(1, C')$$

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their compactness.

The shape factor C of a circle is 1. If the region is long or has holes C is larger than 1. The operator compactness responds to the course of the contour (roughness) and to holes. The value of C is clipped to 1.0, because the pixel area of a region can only be an approximation of a real circle's area. This approximation error is bigger for small regions than for large regions.

In case of an empty region the operator compactness returns the value 0 if no other behavior was set (see set_system). If more than one region is passed the numerical values of the shape factor are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Compactness** (output_control) ..... real(-array) ⇝ *real*
Compactness of the input region(s).
Assertion: Compactness >= 1.0 || Compactness == 0

Result

The operator compactness returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

convexity, eccentricity

See also

contlength, area_center, select_shape

Module

Foundation

connect_and_holes (Regions : : : NumConnected, NumHoles)

Number of connection components and holes

The operator connect_and_holes calculates the number of connection components and the number of holes of each region of Regions.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in the number of their connection components and the number of their holes.

If more than one region is passed the numerical values of the output control parameters NumConnected and NumHoles are each stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **NumConnected** (output_control) ..... integer(-array) ⇝ *integer*
Number of connection components of a region.

▷ **NumHoles** (output_control) ..... integer(-array) ⇝ *integer*
Number of holes of a region.

Result

The operator connect_and_holes returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

euler_number

See also

connection, fill_up, fill_up_shape, union1

Module

Foundation

contlength (Regions : : : ContLength)

Contour length of a region.

The operator contlength calculates the total length of the contour (sum of all connection components of the region) for each region of Regions. The distance between two neighboring contour points parallel to the coordinate axes is rated 1, the distance in the diagonal is rated $\sqrt{2}$.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in the length of their contours.

If more than one region is passed the numerical values of the contour length are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple. In case of an empty region the operator contlength returns the value 0.

Attention

The contour of holes is not calculated.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **ContLength** (output_control) ..... real(-array) ⇝ *real*
Contour length of the input region(s).
Assertion: ContLength >= 0

Example

```
#include "HalconCpp.h"
#include <iostream>
using namespace HalconCpp;

int main (int argc, char *argv[])
{
  HWindow        w(10,10,512,512);
  HRegion        reg;
  int            NumOfElements = 3;

  std::cout << "Draw " << NumOfElements << " regions " << std::endl;
  GenEmptyObj(&reg);
  for (int i=0; i < NumOfElements; i++)
  {
    reg = reg.ConcatObj(w.DrawRegion());
  }

  HTuple circ = reg.Circularity ();
  HTuple cont = reg.Contlength ();

  for (int i = 0; i < NumOfElements; i++)
  {
    std::cout << "region " << i+1 << ": \tcircularity = " << circ[i].D() <<
                 "\tcontour length =" << cont[i].D() << std::endl;
  }
  std::cout << "Click into the graphics window to end." << std::endl;
  w.Click ();
  return(0);
}
```

Result

The operator contlength returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Possible Successors

get_region_contour

Alternatives

compactness

See also

area_center, get_region_contour

Module

Foundation

convexity (Regions : : : Convexity)

Shape factor for the convexity of a region.

The operator convexity calculates the convexity of each input region of Regions.

Calculation: If F_c is the area of the convex hull and F_o the original area of the region the shape factor C is defined as:

$$C = \frac{F_o}{F_c}$$

The shape factor C is 1 if the region is convex (e.g., rectangle, circle etc.). If there are indentations or holes C is smaller than 1.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their convexity.

In case of an empty region the operator convexity returns the value 0 (if no other behavior was set (see set_system)). If more than one region is passed the numerical values of the contour length are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Convexity** (output_control) ..... real(-array) ⇝ *real*
Convexity of the input region(s).
Assertion: Convexity <= 1

Result

The operator convexity returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

See also

select_shape, area_center, shape_trans

Module

Foundation

diameter_region (Regions : : : Row1, Column1, Row2, Column2, Diameter)

Maximal distance between two boundary points of a region.

The operator diameter_region calculates the maximal distance between two boundary points of a region. The coordinates of these two extremes and the distance between them will be returned.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their diameter.

Attention

If the region is empty, the results of Row1, Column1, Row2 and Column2 (all of them = 0) may lead to confusion.

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **Row1** (output_control) ..... line.begin.y(-array) ⇝ *integer*
  Row index of the first extreme point.
- ▷ **Column1** (output_control) ..... line.begin.x(-array) ⇝ *integer*
  Column index of the first extreme point.
- ▷ **Row2** (output_control) ..... line.end.y(-array) ⇝ *integer*
  Row index of the second extreme point.
- ▷ **Column2** (output_control) ..... line.end.x(-array) ⇝ *integer*
  Column index of the second extreme point.
- ▷ **Diameter** (output_control) ..... number(-array) ⇝ *real*
  Distance of the two extreme points.

Complexity

If F is the area of a region, the runtime complexity amounts to $O(sqrt(F))$ on average.

Result

The operator diameter_region returns the value 2 (H_MSG_TRUE), if the input is not empty. The reaction to empty input (no input regions are available) may be determined with the help of the operator set_system ('no_object_result',<Result>). The reaction concerning an empty region (region is the empty set) will be determined by the operator set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, runlength_features

Possible Successors

disp_line

Alternatives

smallest_rectangle2

Module

Foundation

eccentricity (Regions : : : Anisometry, Bulkiness, StructureFactor)

Shape features derived from the ellipse parameters.

The operator eccentricity calculates the three shape features Anisometry, Bulkiness, and StructureFactor for the given Regions:

$$\begin{aligned} \texttt{Anisometry} &= \frac{Ra}{Rb} \\ \texttt{Bulkiness} &= \frac{\pi \cdot Ra \cdot Rb}{A} \\ \texttt{StructureFactor} &= \texttt{Anisometry} \cdot \texttt{Bulkiness} - 1 \end{aligned}$$

where Ra and Rb denote the ellipse radii (see elliptic_axis) and A denotes the area of the region (see area_center).

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their anisometry, bulkiness and structure factor.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see set_system).

Attention

It should be noted that, like for all region-moments-based operators, the region's pixels are regarded as mathematical, infinitely small points that are represented by the center of the pixels (see the documentation of elliptic_axis). This can lead to non-empty regions that have Rb = *0*. In these cases, the output features that require a division by Rb are set to *0*. In particular, regions that contain a single point or regions whose points lie exactly on a straight line (e.g., one pixel high horizontal regions or one pixel wide vertical regions) have an anisometry of *0*.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Anisometry** (output_control) ..... real(-array) ⇝ *real*
Shape feature (in case of a circle = 1.0).
Assertion: Anisometry >= 1.0

▷ **Bulkiness** (output_control) ..... real(-array) ⇝ *real*
Calculated shape feature.

▷ **StructureFactor** (output_control) ..... real(-array) ⇝ *real*
Calculated shape feature.

Complexity

If F is the area of the region the mean runtime complexity is $O(sqrt(F))$.

Result

The operator eccentricity returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

See also

elliptic_axis, moments_region_2nd, select_shape, area_center

Module

Foundation

elliptic_axis (Regions : : : Ra, Rb, Phi)

Calculate the parameters of the equivalent ellipse.

The operator elliptic_axis calculates the radii Ra and Rb and the orientation Phi of the ellipse having the same orientation and the same aspect ratio as the input region in Regions. Several input regions can be passed as tuples. Ra represents the main radius of the ellipse whereas the radius Rb represents the secondary radius of the ellipse. The orientation of the main axis with regard to the x-axis is returned in Phi which is represented in radians. The principle axis of the ellipse is equivalent to the principle axis of the inertia moment of the input region.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their Phi, Ra and Rb.

Calculation:

If the moments M_{20}, M_{02} and M_{11} are normalized and passed to the area (see moments_region_2nd), the radii Ra and Rb are calculated as:

$$\mathtt{Ra} = \frac{\sqrt{8(M_{20} + M_{02} + \sqrt{(M_{20} - M_{02})^2 + 4M_{11}^2})}}{2}$$

$$\mathtt{Rb} = \frac{\sqrt{8(M_{20} + M_{02} - \sqrt{(M_{20} - M_{02})^2 + 4M_{11}^2})}}{2}$$

The orientation Phi is defined by:

$$\mathtt{Phi} = -0.5 atan2(2M_{11}, M_{02} - M_{20})$$

If more than one region is passed, the results are stored in tuples. The index of an resulting tuple element corresponds to the index of the respective input region.

If an empty region is passed, all parameters have the value 0.0 if no other behavior was set (see set_system ('no_object_result',<Result>)).

Attention

It should be noted that, like for all region-moments-based operators, the region's pixels are regarded as mathematical, infinitely small points that are represented by the center of the pixels. This means that Ra and Rb can assume the value *0*. In particular, for an empty region and a region containing a single point Ra = Rb = *0* is returned. Furthermore, for regions whose points lie exactly on a straight line (e.g., one pixel high horizontal regions or one pixel wide vertical regions), Rb = *0* is returned.

Parameters

- ▷ **Regions** (input_object) . . . region(-array) ⇝ *object*
  Input regions.
- ▷ **Ra** (output_control) . . . real(-array) ⇝ *real*
  Main radius (normalized to the area).
  Assertion: Ra >= 0.0
- ▷ **Rb** (output_control) . . . real(-array) ⇝ *real*
  Secondary radius (normalized to the area).
  Assertion: Rb >= 0.0 && Rb <= Ra
- ▷ **Phi** (output_control) . . . angle.rad(-array) ⇝ *real*
  Angle between main radius and x-axis in radians.
  Assertion: - pi / 2 < Phi && Phi <= pi / 2

Example

```
read_image(Image,'fabrik')
regiongrowing(Image,Seg,5,5,6,100)
elliptic_axis(Seg,Ra,Rb,Phi)
area_center(Seg,_,Row,Column)
gen_ellipse(Ellipses,Row,Column,Phi,Ra,Rb)
dev_set_draw('margin')
dev_display(Ellipses)
```

Complexity

If F is the area of a region the mean runtime complexity is $O(sqrt(F))$.

Result

The operator `elliptic_axis` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Possible Successors

`gen_ellipse`

Alternatives

`smallest_rectangle2`, `orientation_region`

See also

`moments_region_2nd`, `select_shape`, `set_shape`

References

R. Haralick, L. Shapiro “Computer and Robot Vision” Addison-Wesley, 1992, pp. 73-75

Module

Foundation

euler_number (Regions : : : EulerNumber)

Calculate the Euler number.

The operator `euler_number` calculates the Euler number, i.e., the difference between the number of connection components and the number of holes.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their Euler number.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **EulerNumber** (output_control) ..... integer(-array) ⇝ *integer*
Calculated Euler number.

Result

The operator euler_number returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

connect_and_holes

Module

Foundation

find_neighbors (Regions1, Regions2 : : MaxDistance : RegionIndex1, RegionIndex2)

Search direct neighbors.

The operator find_neighbors determines neighboring regions with Regions1 and Regions2 containing the regions to be examined. Regions1 can have three different states:

- Regions1 is empty:
  In this case all regions in Regions2 are permutatively checked for neighborhood.
- Regions1 consists of one region:
  The regions of Regions1 are compared to all regions in Regions2.
- Regions1 consists of the same number of regions as Regions2:
  Here all regions at the n-th position in Regions1 and Regions2 are checked for the neighboring relation.

The operator find_neighbors uses the chessboard distance between neighboring regions. It can be specified by the parameter MaxDistance. Neighboring regions are located at the n-th position in RegionIndex1 and RegionIndex2, i.e., the region with index RegionIndex1[n] from Regions1 is the neighbor of the region with index RegionIndex2[n] from Regions2.

Attention

Covered regions are not found!

Parameters

▷ **Regions1** (input_object) . . . . . region(-array) ⇝ *object*
Starting regions.

▷ **Regions2** (input_object) . . . . . region(-array) ⇝ *object*
Comparative regions.

▷ **MaxDistance** (input_control) . . . . . integer ⇝ *integer*
Maximal distance of regions.
Default: 1
Suggested values: MaxDistance ∈ {1, 2, 3, 4, 5, 6, 7, 8, 10, 15, 20, 50}
Value range: 1 ≤ MaxDistance ≤ 255
Minimum increment: 1
Recommended increment: 1

▷ **RegionIndex1** (output_control) . . . . . integer-array ⇝ *integer*
Indices of the found regions from Regions1.

▷ **RegionIndex2** (output_control) . . . . . integer-array ⇝ *integer*
Indices of the found regions from Regions2.

Result

The operator find_neighbors returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, regiongrowing, connection

See also

spatial_relation, select_region_spatial, expand_region, distance_transform, interjacent, boundary

Module

Foundation

get_region_index (Regions : : Row, Column : Index)

Index of all regions containing a given pixel.

The operator get_region_index returns the index of all regions in Regions (range of values: 1 to n) containing the test pixel (Row,Column), i.e.:

$$|\texttt{Regions}[n] \cap \{(\texttt{Row}, \texttt{Column})\}| \neq \emptyset$$

The returned indices can be used, e.g., in select_obj to select the regions containing the test pixel.

Attention

If the regions overlap more than one region might contain the pixel. In this case all these regions are returned. If no region contains the indicated pixel the empty tuple (= no region) is returned.

Parameters

▷ **Regions** (input_object) . . . . . region-array ⇝ *object*
Regions to be examined.

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Line index of the test pixel.
Default: 100
(lin)

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Column index of the test pixel.
Default: 100
(lin)

▷ **Index** (output_control) . . . . . integer(-array) ⇝ *integer*
Index of the regions containing the test pixel.

Complexity

If F is the area of the region and N is the number of regions the mean runtime complexity is $O(ln(sqrt(F)) * N)$.

Result

The operator get_region_index returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

select_region_point

See also

get_mbutton, get_mposition, test_region_point

Module

Foundation

get_region_thickness (Region : : : Thickness, Histogramm)

Access the thickness of a region along the main axis.

The operator get_region_thickness calculates the thickness of the regions along the main axis (see elliptic_axis) for each pixel of the section. The thickness at one point on the main axis is defined as the distance between the intersections of the contour with the plumb on the main axis in the respective point which are the furthest apart. Additionally the operator get_region_thickness returns the Histogramm of the thicknesses of the region. The length of the histogram corresponds to the largest occurring thickness in the observed region.

Attention

Only one region may be passed. If the region has several connection components, only the first one is investigated. All other components are ignored.

Parameters

▷ **Region** (input_object) ..... region ⇝ *object*
Region to be analyzed.

▷ **Thickness** (output_control) ..... integer-array ⇝ *integer*
Thickness of the region along its main axis.

▷ **Histogramm** (output_control) ..... integer-array ⇝ *integer*
Histogram of the thickness of the region along its main axis.

Result

The operator get_region_thickness returns the value 2 (H_MSG_TRUE) if exactly one region is passed. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

sobel_amp, threshold, connection, select_shape, select_obj

See also

copy_obj, elliptic_axis

Module

Foundation

hamming_distance (Regions1, Regions2 : : : Distance, Similarity)

Hamming distance between two regions.

The operator hamming_distance returns the hamming distance between two regions, i.e., the number of pixels of the regions which are different (Distance), i.e., the number of pixels contained in one region but not in the other:

$$\texttt{Distance} = |\texttt{Regions1} \cap \overline{\texttt{Regions2}}| + |\texttt{Regions2} \cap \overline{\texttt{Regions1}}|$$

The parameter Similarity describes the similarity between the two regions based on the hamming distance Distance:

$$\texttt{Similarity} = 1 - \frac{\texttt{Distance}}{|\texttt{Regions1}| + |\texttt{Regions2}|}$$

If both regions are empty Similarity is set to 0. The regions with the same index from both input parameters are always compared.

Attention

In both input parameters the same number of regions must be passed.

Parameters

- ▷ **Regions1** (input_object) . . . . . region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **Regions2** (input_object) . . . . . region(-array) ⇝ *object*
  Comparative regions.
- ▷ **Distance** (output_control) . . . . . integer(-array) ⇝ *integer*
  Hamming distance of two regions.
  Assertion: Distance >= 0
- ▷ **Similarity** (output_control) . . . . . real(-array) ⇝ *real*
  Similarity of two regions.
  Assertion: 0 <= Similarity && Similarity <= 1

Complexity

If F is the area of a region the mean runtime complexity is $O(sqrt(F))$.

Result

hamming_distance returns the value 2 (H_MSG_TRUE) if the number of objects in both parameters is the same and is not 0. The behavior in case of empty input (no input objects available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

intersection, complement, area_center

See also

hamming_change_region

Module

Foundation

hamming_distance_norm (Regions1, Regions2 : : Norm : Distance, Similarity)

Hamming distance between two regions using normalization.

The operator hamming_distance_norm returns the hamming distance between two regions, i.e., the number of pixels of the regions which are different (Distance). Before calculating the difference the region in Regions1 is normalized onto the regions in Regions2. The result is the number of pixels contained in one region but not in the other:

$$\texttt{Distance} = |Norm(\texttt{Regions1}) \cap \overline{\texttt{Regions2}}| + |\texttt{Regions2} \cap \overline{Norm(\texttt{Regions1})}|$$

The parameter Similarity describes the similarity between the two regions based on the hamming distance Distance:

$$\texttt{Similarity} = 1 - \frac{\texttt{Distance}}{|Norm(\texttt{Regions1})| + |\texttt{Regions2}|}$$

The following types of normalization are available:

'center': The region is moved so that both regions have the save center of gravity.

If both regions are empty Similarity is set to 0. The regions with the same index from both input parameters are always compared.

Attention

In both input parameters the same number of regions must be passed.

Parameters

- ▷ **Regions1** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **Regions2** (input_object) ..... region(-array) ⇝ *object*
  Comparative regions.
- ▷ **Norm** (input_control) ..... string(-array) ⇝ *string*
  Type of normalization.
  Default: 'center'
  List of values: Norm ∈ {'center'}
- ▷ **Distance** (output_control) ..... integer(-array) ⇝ *integer*
  Hamming distance of two regions.
  Assertion: Distance >= 0
- ▷ **Similarity** (output_control) ..... real(-array) ⇝ *real*
  Similarity of two regions.
  Assertion: 0 <= Similarity && Similarity <= 1

Complexity

If F is the area of a region the mean runtime complexity is $O(sqrt(F))$.

Result

hamming_distance_norm returns the value 2 (H_MSG_TRUE) if the number of objects in both parameters is the same and is not 0. The behavior in case of empty input (no input objects available) is set via the operator set_system('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

intersection, complement, area_center

See also

hamming_change_region

Module

Foundation

height_width_ratio (Regions : : : Height, Width, Ratio)

Compute the width, height, and aspect ratio of the surrounding rectangle parallel to the coordinate axes.

The operator height_width_ratio calculates the surrounding rectangle of all input regions (parallel to the coordinate axes). The surrounding rectangle is described by the coordinates of the corner pixels (Row1,Column1,Row2,Column2) (see smallest_rectangle1). Based on these values, height_width_ratio computes the width, height, and aspect ratio of the smallest surrounding rectangle as follows:

$$\begin{aligned} \texttt{Width} &= \texttt{Column2} - \texttt{Column1} + 1 \\ \texttt{Height} &= \texttt{Row2} - \texttt{Row1} + 1 \\ \texttt{Ratio} &= \texttt{Height}/\texttt{Width} \end{aligned}$$

If more than one region is passed in Regions, the results are stored in tuples in the same order as the respective regions in the Regions. In case of empty regions, all parameters have the value 0 if no other behavior was set (see set_system).

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **Height** (output_control) ..... extent.y(-array) ⇝ *integer*
  Height of the surrounding rectangle of the region.
- ▷ **Width** (output_control) ..... extent.x(-array) ⇝ *integer*
  Width of the surrounding rectangle of the region.
- ▷ **Ratio** (output_control) ..... real(-array) ⇝ *real*
  Aspect ratio of the surrounding rectangle of the region.

Complexity

If F is the area of the region the mean runtime complexity is $O(sqrt(F))$.

Result

The operator height_width_ratio returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, runlength_features

Alternatives

smallest_rectangle1, smallest_rectangle2, region_features

See also

select_shape, smallest_circle, elliptic_axis, area_center

Module

Foundation

inner_circle (Regions : : : Row, Column, Radius)

Largest inner circle of a region.

The operator inner_circle determines the largest inner circle of a region. This is the biggest discrete circle region that completely fits into the region. For this circle the center (Row, Column) and the radius (Radius) are calculated. If the position of the circle is ambiguous, the "first possible" position (as far upper left as possible) is returned.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions with varying inner circles.

The output of the operator is chosen in such a way that it can be used as an input for the operators disp_circle, gen_circle, and gen_ellipse_contour_xld.

If several regions are passed in Regions corresponding tuples are returned as output parameters. In case of an empty input region all parameters have the value *0.0* if no other behavior was set with set_system.

Attention

If several inner circles are present at a region only the most upper left solution is returned.

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **Row** (output_control) ..... circle.center.y(-array) ⇝ *real*
  Line index of the center.
- ▷ **Column** (output_control) ..... circle.center.x(-array) ⇝ *real*
  Column index of the center.
- ▷ **Radius** (output_control) ..... circle.radius(-array) ⇝ *real*
  Radius of the inner circle.
  Assertion: Radius >= 0

Example

```
read_image(Image,'fabrik')
regiongrowing(Image,Seg,5,5,6,100)
select_shape(Seg,H,'area','and',100,2000)
inner_circle(H,Row,Column,Radius)
gen_circle(Circles,Row,Column,Radius)
dev_set_draw('margin')
dev_display(Circles)
```

Complexity

If F is the area of the region and R is the radius of the inner circle the runtime complexity is $O(sqrt(F) * R)$.

Result

The operator inner_circle returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>), the behavior in case of empty region is set via set_system ('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, runlength_features

Possible Successors

gen_circle, disp_circle

Alternatives

erosion_circle, inner_rectangle1

See also

set_shape, select_shape, smallest_circle

Module

Foundation

inner_rectangle1 (Regions : : : Row1, Column1, Row2, Column2)

Largest inner rectangle of a region.

The operator inner_rectangle1 determines the largest axis-parallel rectangle that fits into a region. The rectangle is described by the coordinates of the corner pixels (Row1, Column1, Row2, Column2).

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in the width and height of their inner rectangle.

If more than one region is passed in Regions the results are stored in tuples, the index of a value in the tuple corresponding to the index of the input region. For empty regions all parameters have the value 0 if no other behavior was set (see set_system).

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Region to be examined.
- ▷ **Row1** (output_control) ..... rectangle.origin.y(-array) ⇝ *integer*
  Row coordinate of the upper left corner point.
- ▷ **Column1** (output_control) ..... rectangle.origin.x(-array) ⇝ *integer*
  Column coordinate of the upper left corner point.
- ▷ **Row2** (output_control) ..... rectangle.corner.y(-array) ⇝ *integer*
  Row coordinate of the lower right corner point.
- ▷ **Column2** (output_control) ..... rectangle.corner.x(-array) ⇝ *integer*
  Column coordinate of the lower right corner point.

Result

The operator inner_rectangle1 returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

threshold, regiongrowing, connection

Possible Successors

disp_rectangle1, gen_rectangle1

Alternatives

inner_circle

See also

smallest_rectangle1, select_shape

Module

Foundation

moments_region_2nd (Regions : : : M11, M20, M02, Ia, Ib)

Calculate the geometric moments of regions.

moments_region_2nd calculates the geometric moments M11, M20 and M02 of the input regions in Regions. Furthermore the major and minor axis of the input regions are calculated and returned in Ia and Ib.

The covariance matrix is given by:

$$\begin{bmatrix} M20 & M11 \\ M11 & M02 \end{bmatrix}$$

The row-dependent moment of 2nd order is returned in `M20` and the column-dependent moment of 2nd order is returned in `M02`. The moment `M11` represents the covariance between the row and column coordinates of the region points.

Calculation: r_0 and c_0 are the coordinates of the center of gravity of a region R. Then the moments M_{ij} are defined by:

$$M_{ij} = \sum_{(r,c) \in R} (r_0 - r)^i (c_0 - c)^j$$

wherein r and c run over all pixels of the region R.

Furthermore the length of the major and minor axes are defined by:

$$Ia = h + \sqrt{h^2 - M20 * M02 + M11^2}$$

$$Ib = h - \sqrt{h^2 - M20 * M02 + M11^2}$$

wherein h is defined by:

$$h = \frac{M20 + M02}{2}$$

The equation for the major and minor axes can be derived from the definition of the moments by diagonalizing the covariance matrix and reforming the resulting formula.

If more than one region is passed, the results are returned in tuples. The index of a tuple element corresponds to the index of the respective input region.

If an empty region is passed, 0.0 is returned for all parameters, if no other behavior was set (see `set_system`).

Parameters

- ▷ **`Regions`** (input_object) ..... region(-array) ⇝ *object*
  Input regions.
- ▷ **`M11`** (output_control) ..... real(-array) ⇝ *real*
  Product of inertia of the axes through the center parallel to the coordinate axes.
- ▷ **`M20`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order (row-dependent).
- ▷ **`M02`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order (column-dependent).
- ▷ **`Ia`** (output_control) ..... real(-array) ⇝ *real*
  Length of the major axis of the input region.
- ▷ **`Ib`** (output_control) ..... real(-array) ⇝ *real*
  Length of the minor axis of the input region.

Complexity

If F is the area of the region the mean runtime complexity is $O(sqrt(F))$.

Result

The operator `moments_region_2nd` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system ('no_object_result',<Result>)`. The behavior in case of empty region (region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

moments_region_2nd_invar

See also

elliptic_axis

References

R. Haralick, L. Shapiro "Computer and Robot Vision" Addison-Wesley, 1992, pp. 73-75

Module

Foundation

moments_region_2nd_invar (Regions : : : M11, M20, M02)

Geometric moments of regions.

The operator moments_region_2nd_invar calculates the scaled moments (M20, M02) and the product of inertia of the axes through the center parallel to the coordinate axes (M11).

Calculation: Z_0 and S_0 are the coordinates of the center of a region R with the area F. Then the moments M_{ij} are defined by:

$$M_{ij} = \frac{1}{F^2} \sum_{(Z,S) \in R} (Z_0 - Z)^i (S_0 - S)^j$$

,

wherein Z and S run through all pixels of the region R.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see set_system).

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **M11** (output_control) ..... real(-array) ⇝ *real*
  Product of inertia of the axes through the center parallel to the coordinate axes.
- ▷ **M20** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order (line-dependent).
- ▷ **M02** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order (column-dependent).

Complexity

If F is the area of the region the mean runtime complexity is $O(sqrt(F))$.

Result

The operator moments_region_2nd_invar returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

moments_region_2nd

See also

elliptic_axis

Module

Foundation

moments_region_2nd_rel_invar (Regions : : : PHI1, PHI2)

Geometric moments of regions.

The operator moments_region_2nd_rel_invar calculates the scaled relative moments (PHI1, PHI2).

Calculation: The moments PHI_1 and PHI_2 are defined by:

$$PHI_1 = M_{20} + M_{02}$$

$$PHI_2 = (M_{20} - M_{02})^2 + 4M_{11}^2$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see set_system).

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **PHI1** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order.
- ▷ **PHI2** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order.

Result

The operator moments_region_2nd_rel_invar returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

moments_region_2nd

See also

elliptic_axis

Module

Foundation

moments_region_3rd (Regions : : : M21, M12, M03, M30)

Geometric moments of regions.

The operator moments_region_3rd calculates the translation-invariant central moments (M21, M12, M03, M30) of order $(p + q)$.

Calculation: x and y are the coordinates of the center of a region R with the area Z.
Then the moments M_{pq} are defined by:

$$M_{pq} = \sum_{i=1} MZ(x_i, y_i)(x_i - x)^p (y_i - y)^q$$

,
wherein are $x = \frac{m_{10}}{m_{00}}$ and $y = \frac{m_{01}}{m_{00}}$.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see set_system).

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **M21** (output_control) ..... real(-array) ⇝ *real*
  Moment of 3rd order (line-dependent).
- ▷ **M12** (output_control) ..... real(-array) ⇝ *real*
  Moment of 3rd order (column-dependent).
- ▷ **M03** (output_control) ..... real(-array) ⇝ *real*
  Moment of 3rd order (column-dependent).
- ▷ **M30** (output_control) ..... real(-array) ⇝ *real*
  Moment of 3rd order (line-dependent).

Complexity

If Z is the area of the region the mean runtime complexity is $O(sqrt(Z))$.

Result

The operator moments_region_3rd returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

moments_region_2nd

See also

elliptic_axis

Module

Foundation

moments_region_3rd_invar (Regions : : : M21, M12, M03, M30)

Geometric moments of regions.

The operator moments_region_3rd_invar calculates the scale-invariant moments (M21, M12, M03, M30).

Calculation: Then the moments M_{ij} are defined by:

$$M_{pq} = \frac{\mu_{pq}}{\mu^3}$$

,
wherein $p + q >= 2$ and $\mu = \mu_{00} = m_{00}$
.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see set_system).

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **M21** (output_control) ..... real(-array) ⇝ *real*
  Moment of 3rd order (line-dependent).
- ▷ **M12** (output_control) ..... real(-array) ⇝ *real*
  Moment of 3rd order (column-dependent).
- ▷ **M03** (output_control) ..... real(-array) ⇝ *real*
  Moment of 3rd order (column-dependent).
- ▷ **M30** (output_control) ..... real(-array) ⇝ *real*
  Moment of 3rd order (line-dependent).

Complexity

If Z is the area of the region the mean runtime complexity is $O(sqrt(Z))$.

Result

The operator moments_region_3rd_invar returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

moments_region_2nd

See also

elliptic_axis

Module

Foundation

moments_region_central (Regions : : : I1, I2, I3, I4)

Geometric moments of regions.

The operator moments_region_central calculates the central moments (I1, I2, I3, I4).

Calculation: Then the moments I_i are defined by:

$$
\begin{aligned}
I_1 &= \mu_{20}\mu_{02} - \mu_{11}^2 \\
I_2 &= (\mu_{30}\mu_{03} - \mu_{21}\mu_{12})^2 - 4(\mu_{30}\mu_{12} - \mu_{21}^2)(\mu_{21}\mu_{03} - \mu_{12}^2) \\
I_3 &= \mu_{20}(\mu_{21}\mu_{03} - \mu_{12}^2) - \mu_{11}(\mu_{30}\mu_{03} - \mu_{21}\mu_{12}) + \mu_{02}(\mu_{30}\mu_{12} - \mu_{21}^2) \\
I_4 &= \mu_{30}^2\mu_{02}^3 - 6\mu_{30}\mu_{21}\mu_{11}\mu_{02}^2 + 6\mu_{30}\mu_{12}\mu_{02}(2\mu_{11}^2 - \mu_{20}\mu_{02}) \\
&\quad +\mu_{30}\mu_{03}(6\mu_{20}\mu_{11}\mu_{02} - 8\mu_{11}^3) + 9\mu_{21}^2\mu_{20}\mu_{02}^2 - 18\mu_{21}\mu_{12}\mu_{20}\mu_{11}\mu_{02} \\
&\quad +6\mu_{21}\mu_{03}\mu_{20}(2\mu_{11}^2 - \mu_{20}\mu_{02}) + 9\mu_{12}^2\mu_{20}^2\mu_{02} - 6\mu_{12}\mu_{03}\mu_{11}\mu_{20}^2 + \mu_{03}^2\mu_{20}^3
\end{aligned}
$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see `set_system`).

Parameters

- ▷ **`Regions`** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **`I1`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order.
- ▷ **`I2`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order.
- ▷ **`I3`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order.
- ▷ **`I4`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 3rd order.

Complexity

If Z is the area of the region the mean runtime complexity is $O(sqrt(Z))$.

Result

The operator `moments_region_central` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system ('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Alternatives

`moments_region_2nd`

See also

`elliptic_axis`

Module

Foundation

`moments_region_central_invar` (Regions : : : PSI1, PSI2, PSI3, PSI4)

Geometric moments of regions.

The operator `moments_region_central_invar` calculates the moments (`PSI1`, `PSI2`, `PSI3`, `PSI4`) that are invariant under translation and general linear transformations.

Calculation: Then the moments ψ_i are defined by:

$$\psi_1 = \frac{I_1}{\mu^4}$$
$$\psi_2 = \frac{I_2}{\mu^1 0}$$
$$\psi_3 = \frac{I_3}{\mu^7}$$
$$\psi_4 = \frac{I_4}{\mu^1 1}$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see `set_system`).

Parameters

- ▷ **`Regions`** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **`PSI1`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order.
- ▷ **`PSI2`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order.
- ▷ **`PSI3`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order.
- ▷ **`PSI4`** (output_control) ..... real(-array) ⇝ *real*
  Moment of 2nd order.

Complexity

If Z is the area of the region the mean runtime complexity is $O(sqrt(Z))$.

Result

The operator `moments_region_central_invar` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system ('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Alternatives

`moments_region_2nd`

See also

`elliptic_axis`

Module

Foundation

`orientation_region` (Regions : : : Phi)

Orientation of a region.

The operator orientation_region calculates the orientation of the region. The operator is based on elliptic_axis. In addition the point on the contour with maximal distance to the center of gravity is calculated. If, in the rotated coordinate system, the column coordinate of this point is less than the column coordinate of the center of gravity, the value of π is added to the angle.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their orientation.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see set_system ('no_object_result',<Result>)).

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Phi** (output_control) ..... angle.rad(-array) ⇝ *real*
Orientation of region (arc measure).
Assertion: - pi <= Phi && Phi < pi

Complexity

If F is the area of a region the mean runtime complexity is $O(sqrt(F))$.

Result

The operator orientation_region returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Possible Successors

disp_arrow

Alternatives

elliptic_axis, smallest_rectangle2

See also

moments_region_2nd, line_orientation

Module

Foundation

rectangularity (Regions : : : Rectangularity)

Shape factor for the rectangularity of a region.

The operator rectangularity calculates the rectangularity of the input regions.

To determine the rectangularity, first a rectangle is computed that has the same first and second order moments as the input region. The computation of the rectangularity measure is finally based on the area of the difference between the computed rectangle and the input region normalized with respect to the area of the rectangle.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their rectangularity.

For rectangles rectangularity returns the value 1. The more the input region deviates from a perfect rectangle, the less the returned value for Rectangularity will be.

In case of an empty region the operator rectangularity returns the value 0 (if no other behavior was set (see set_system)). If more than one region is passed the numerical values of the rectangularity are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

Attention

For input regions which orientation cannot be computed by using second order moments (as it is the case for square regions, for example), the returned Rectangularity is underestimated by up to 10% depending on the orientation of the input region.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Rectangularity** (output_control) ..... real(-array) ⇝ *real*
Rectangularity of the input region(s).
Assertion: 0 <= Rectangularity && Rectangularity <= 1.0

Result

The operator rectangularity returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

circularity, compactness, convexity, eccentricity

See also

contlength, area_center, select_shape

References

P. L. Rosin: "Measuring rectangularity"; Machine Vision and Applications; vol. 11; pp. 191-196; Springer-Verlag, 1999.

Module

Foundation

region_features (Regions : : Features : Value)

Calculate shape features of regions.

The operator region_features calculates for each input region from Regions the indicated features (Features).

For an illustration of these features, please refer to the documentation of this chapter (Regions / Features).

Possible values for Features:

'area': Area of the object

'row': Row index of the center

'column': Column index of the center

'row1': Row index of upper left corner

'column1': Column index of upper left corner

'row2': Row index of lower right corner

'column2': Column index of lower right corner
'width': Width of the region (parallel to the coordinate axes)
'height': Height of the region (parallel to the coordinate axes)
'ratio': Ratio of the height and the width of the region (parallel to the coordinate axes)
'circularity': Circularity (see `circularity`)
'compactness': Compactness (see `compactness`)
'contlength': Total length of contour (see operator `contlength`)
'convexity': Convexity (see `convexity`)
'rectangularity': Rectangularity (see `rectangularity`)
'ra': Main radius of the equivalent ellipse (see `elliptic_axis`)
'rb': Secondary radius of the equivalent ellipse (see `elliptic_axis`)
'phi': Orientation of the equivalent ellipse (see `elliptic_axis`)
'anisometry': Anisometry (see `eccentricity`)
'bulkiness': Bulkiness (see operator `eccentricity`)
'struct_factor': Structure Factor (see operator `eccentricity`)
'outer_radius': Radius of smallest surrounding circle (see `smallest_circle`)
'inner_radius': Radius of largest inner circle (see `inner_circle`)
'inner_width': Width of the largest axis-parallel rectangle that fits into the region (see `inner_rectangle1`)
'inner_height': Height of the largest axis-parallel rectangle that fits into the region (see `inner_rectangle1`)
'dist_mean': Mean distance from the region border to the center (see operator `roundness`)
'dist_deviation': Deviation of the distance from the region border to the center (see operator `roundness`)
'roundness': Roundness (see operator `roundness`)
'num_sides': Number of polygon sides (see operator `roundness`)
'connect_num': Number of connection components (see operator `connect_and_holes`)
'holes_num': Number of holes (see operator `connect_and_holes`)
'area_holes': Area of the holes of the object (see operator `area_holes`)
'max_diameter': Maximum diameter of the region (see operator `diameter_region`)
'orientation': Orientation of the region (see operator `orientation_region`)
'euler_number': Euler number (see operator `euler_number`)
'rect2_phi': Orientation of the smallest surrounding rectangle (see operator `smallest_rectangle2`)
'rect2_len1': Half the length of the smallest surrounding rectangle (see operator `smallest_rectangle2`)
'rect2_len2': Half the width of the smallest surrounding rectangle (see operator `smallest_rectangle2`)
'moments_m11': Geometric moments of the region (see operator `moments_region_2nd`)
'moments_m20': Geometric moments of the region (see operator `moments_region_2nd`)
'moments_m02': Geometric moments of the region (see operator `moments_region_2nd`)
'moments_ia': Geometric moments of the region (see operator `moments_region_2nd`)
'moments_ib': Geometric moments of the region (see operator `moments_region_2nd`)
'moments_m11_invar': Geometric moments of the region (see operator `moments_region_2nd_invar`)
'moments_m20_invar': Geometric moments of the region (see operator `moments_region_2nd_invar`)
'moments_m02_invar': Geometric moments of the region (see operator `moments_region_2nd_invar`)
'moments_phi1': Geometric moments of the region (see operator `moments_region_2nd_rel_invar`)
'moments_phi2': Geometric moments of the region (see operator `moments_region_2nd_rel_invar`)
'moments_m21': Geometric moments of the region (see operator `moments_region_3rd`)
'moments_m12': Geometric moments of the region (see operator `moments_region_3rd`)
'moments_m03': Geometric moments of the region (see operator `moments_region_3rd`)

'moments_m30': Geometric moments of the region (see operator moments_region_3rd)

'moments_m21_invar': Geometric moments of the region (see operator moments_region_3rd_invar)

'moments_m12_invar': Geometric moments of the region (see operator moments_region_3rd_invar)

'moments_m03_invar': Geometric moments of the region (see operator moments_region_3rd_invar)

'moments_m30_invar': Geometric moments of the region (see operator moments_region_3rd_invar)

'moments_i1': Geometric moments of the region (see operator moments_region_central)

'moments_i2': Geometric moments of the region (see operator moments_region_central)

'moments_i3': Geometric moments of the region (see operator moments_region_central)

'moments_i4': Geometric moments of the region (see operator moments_region_central)

'moments_psi1': Geometric moments of the region (see operator moments_region_central_invar)

'moments_psi2': Geometric moments of the region (see operator moments_region_central_invar)

'moments_psi3': Geometric moments of the region (see operator moments_region_central_invar)

'moments_psi4': Geometric moments of the region (see operator moments_region_central_invar)

Several features are processed in the sequence in which they are entered.

Parameters

▷ **Regions** (input_object) ..... region-array ⇝ *object*
Regions to be examined.

▷ **Features** (input_control) ..... string(-array) ⇝ *string*
Shape features to be calculated.
Default: 'area'
List of values: Features ∈ {'area', 'row', 'column', 'width', 'height', 'ratio', 'row1', 'column1', 'row2', 'column2', 'circularity', 'compactness', 'contlength', 'convexity', 'rectangularity', 'ra', 'rb', 'phi', 'anisometry', 'bulkiness', 'struct_factor', 'outer_radius', 'inner_radius', 'inner_width', 'inner_height', 'max_diameter', 'dist_mean', 'dist_deviation', 'roundness', 'num_sides', 'orientation', 'connect_num', 'holes_num', 'area_holes', 'euler_number', 'rect2_phi', 'rect2_len1', 'rect2_len2', 'moments_m11', 'moments_m20', 'moments_m02', 'moments_ia', 'moments_ib', 'moments_m11_invar', 'moments_m20_invar', 'moments_m02_invar', 'moments_phi1', 'moments_phi2', 'moments_m21', 'moments_m12', 'moments_m03', 'moments_m30', 'moments_m21_invar', 'moments_m12_invar', 'moments_m03_invar', 'moments_m30_invar', 'moments_i1', 'moments_i2', 'moments_i3', 'moments_i4', 'moments_psi1', 'moments_psi2', 'moments_psi3', 'moments_psi4'}

▷ **Value** (output_control) ..... real(-array) ⇝ *real*
The calculated features.

Example

```
read_image(Image,'monkey')
threshold(Image,S1,160,255)
connection(S1,S2)
region_features(S2,['area','anisometry'],Value)
```

Result

The operator region_features returns the value 2 (H_MSG_TRUE) if the input is not empty. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, runlength_features

Possible Successors

select_shape, select_gray, shape_trans, reduce_domain, count_obj

See also

area_center, circularity, compactness, contlength, convexity, rectangularity, elliptic_axis, eccentricity, inner_circle, smallest_circle, smallest_rectangle1, smallest_rectangle2, inner_rectangle1, roundness, connect_and_holes, area_holes, diameter_region, orientation_region, moments_region_2nd, moments_region_2nd_invar, moments_region_2nd_rel_invar, moments_region_3rd, moments_region_3rd_invar, moments_region_central, moments_region_central_invar, select_obj, select_shape

Module

Foundation

roundness (Regions : : : Distance, Sigma, Roundness, Sides)

Shape factors from contour.

The operator roundness examines the distance between the contour and the center of the area. In particular the mean distance (Distance), the deviation from the mean distance (Sigma) and two shape features derived therefrom are determined. Roundness is the relation between mean value and standard deviation, and Sides indicates the number of polygon pieces if a regular polygon is concerned.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in their mean distance, distance deviation, roundness and number of polygon pieces.

The contour for calculating the features is determined depending on the global neighborhood (see set_system).

Calculation:

If p is the center of the area, p_i the pixels and F the area of the contour.

$$\texttt{Distance} = \frac{1}{F} \sum ||p - p_i||$$

$$\texttt{Sigma}^2 = \frac{1}{F} \sum (||p - p_i|| - \texttt{Distance})^2$$

$$\texttt{Roundness} = 1 - \frac{\texttt{Sigma}}{\texttt{Distance}}$$

$$\texttt{Sides} = 1.4111 \left(\frac{\texttt{Distance}}{\texttt{Sigma}} \right)^{0.4724}$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see set_system).

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Distance** (output_control) ..... real(-array) ⇝ *real*
Mean distance from the center.
Assertion: Distance >= 0.0

▷ **Sigma** (output_control) ..... real(-array) ⇝ *real*
Standard deviation of Distance.
Assertion: Sigma >= 0.0

▷ **Roundness** (output_control) . . . . . real(-array) ⇝ *real*
Shape factor for roundness.
Assertion: Roundness <= 1.0

▷ **Sides** (output_control) . . . . . real(-array) ⇝ *real*
Number of polygon sides.
Assertion: Sides >= 0

Complexity

If F is the area of a region the mean runtime complexity is $O(sqrt(F))$.

Result

The operator `roundness` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system ('no_object_result',<Result>)`, the behavior in case of empty region is set via `set_system ('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Alternatives

`compactness`

See also

`contlength`

References

R. Haralick, L. Shapiro “Computer and Robot Vision” Addison-Wesley, 1992, pp. 61

Module

Foundation

runlength_distribution (Region : : : Foreground, Background)

Distribution of runs needed for runlength encoding of a region.

The operator `runlength_distribution` calculates the distribution of the runs of a region of the fore- and background. The frequency of the occurrence of a certain length is calculated. Runs of infinite length are not counted. Therefore the background are the holes of the region. As many values are passed as set by the maximum length of fore- or background, respectively. The length of both tuples usually differs. The first entry of the tuples is always 0 (no runs of the length 0). If there are no blanks the empty tuple is passed at `Background`. Analogously the empty tuple is passed in case of an empty region at `Foreground`.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
Region to be examined.

▷ **Foreground** (output_control) . . . . . integer-array ⇝ *integer*
Length distribution of the region (foreground).

▷ **Background** (output_control) . . . . . integer-array ⇝ *integer*
Length distribution of the background.

Complexity

If n is the number of runs of the region the runtime complexity is $O(n)$.

Result

The operator `runlength_distribution` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system ('no_object_result',<Result>)`. If more than one region is passed an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`threshold`, `select_obj`

Alternatives

`runlength_features`

See also

`runlength_features`

Module

Foundation

runlength_features (Regions : : : NumRuns, KFactor, LFactor, MeanLength, Bytes)

Characteristic values for runlength coding of regions.

The operator `runlength_features` calculates for every input region from `Regions` the number of runs necessary for storing this region with the aid of runlength coding. Furthermore the so-called "K-factor" is determined, which indicates by how much the number of runs differs from the ideal of the square in which this value is 1.0.

The K-factor (`KFactor`) is calculated according to the formula:

$$\texttt{KFactor} = \frac{\texttt{NumRuns}}{\sqrt{Area}}$$

wherein $Area$ indicates the area of the region. It should be noted that the K-factor can be smaller than 1.0 (in case of long horizontal regions).

The L-factor (`LFactor`) indicates the mean number of runs for each line index occurring in the region.

`MeanLength` indicates the mean length of the runs. The parameter `Bytes` indicates how many bytes are necessary for coding the region with runlengths.

Attention

All features calculated by the operator `runlength_features` are not rotation invariant because the runlength coding depends on the direction. The operator `runlength_features` does not serve for calculating shape features but for controlling and analyzing the efficiency of the runlength coding.

Parameters

▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be examined.

▷ **NumRuns** (output_control) . . . . . integer(-array) ⇝ *integer*
Number of runs.
Assertion: 0 <= NumRuns

▷ **KFactor** (output_control) . . . . . real(-array) ⇝ *real*
Storing factor in relation to a square.
Assertion: 0 <= KFactor

▷ **LFactor** (output_control) . . . . . real(-array) ⇝ *real*
Mean number of runs per line.
Assertion: 0 <= LFactor

▷ **MeanLength** (output_control) . . . . . real(-array) ⇝ *real*
Mean length of runs.
Assertion: 0 <= MeanLength

▷ **Bytes** (output_control) ..... integer(-array) ⇝ *integer*
Number of bytes necessary for coding the region.
Assertion: 0 <= Bytes

Complexity

The mean runtime complexity is $O(1)$.

Result

The operator `runlength_features` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

See also

`runlength_distribution`

Module

Foundation

select_region_point (Regions : DestRegions : Row, Column :)

Choose all regions containing a given pixel.

The operator `select_region_point` selects all regions from `Regions` containing the test pixel (`Row`,`Column`), i.e.:

$$|\texttt{Regions}[n] \cap \{(\texttt{Row}, \texttt{Column})\}| = 1$$

Attention

If the regions overlap more than one region might contain the pixel. In this case all these regions are returned. If no region contains the indicated pixel the empty tuple (= no region) is returned.

Parameters

▷ **Regions** (input_object) ..... region-array ⇝ *object*
Regions to be examined.

▷ **DestRegions** (output_object) ..... region-array ⇝ *object*
All regions containing the test pixel.

▷ **Row** (input_control) ..... point.y ⇝ *integer*
Line index of the test pixel.
Default: 100

▷ **Column** (input_control) ..... point.x ⇝ *integer*
Column index of the test pixel.
Default: 100

Example

```
read_image(Image,'fabrik')
regiongrowing(Image,Seg,3,3,5,0)
dev_set_color('red')
dev_set_draw('margin')
Button := 1
while (Button == 1)
  get_mbutton(WindowHandle,Row,Column,Button)
```

```
  select_region_point(Seg,Single,Row,Column)
  dev_display(Image)
  dev_display(Single)
endwhile
```

Complexity

If F is the area of the region and N is the number of regions, the mean runtime complexity is $O(ln(sqrt(F)) * N)$.

Result

The operator `select_region_point` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. The behavior in case of empty input (no input regions available) is set via the operator `set_system ('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Alternatives

`test_region_point`

See also

`get_mbutton`, `get_mposition`

Module

Foundation

select_region_spatial (Regions1, Regions2 : : Direction : RegionIndex1, RegionIndex2)

Pose relation of regions.

The operator `select_region_spatial` chooses the regions from `Regions2` which are sufficient for the neighboring relation `Direction`. The regions to be examined have to be passed in `Regions1` or `Regions2`, respectively. `Regions1` can have three different states:

- `Regions1` is empty:
  In this case all regions in `Regions2` are permutatively checked for neighborhood.
- `Regions1` consists of one region:
  The regions of `Regions1` are compared to all regions in `Regions2`.
- `Regions1` consists of the same number of regions as `Regions2`:
  The regions at the n-th position in `Regions1` and `Regions2` are each checked for a neighboring relation.

Possible values for `Direction` are:

'left': `Regions2` is left of `Regions1`

'right': `Regions2` is right of `Regions1`

'above': `Regions2` is above `Regions1`

'below': `Regions2` is below `Regions1`

The operator `select_region_spatial` calculates the centers of the regions to be compared and decides according to the angle between the center straight lines and the x axis whether the direction relation is fulfilled. The relation is fulfilled within the area of -45 degree to +45 degree around the coordinate axes. Thus, the direction relation can be understood in such a way that the center of the second region must be located left (or right, above, below) of the center of the first region. The indices of the regions fulfilling the direction relation are located at the

n-th position in RegionIndex1 and RegionIndex2, i.e., the region with the index RegionIndex2[n] has the indicated relation with the region with the index RegionIndex1[n]. Access to regions via the index can be obtained via the operator copy_obj.

Parameters

▷ **Regions1** (input_object) ..... region(-array) ⇝ *object*
Starting regions

▷ **Regions2** (input_object) ..... region(-array) ⇝ *object*
Comparative regions

▷ **Direction** (input_control) ..... string ⇝ *string*
Desired neighboring relation.
Default: 'left'
List of values: Direction ∈ {'left', 'right', 'above', 'below'}

▷ **RegionIndex1** (output_control) ..... integer-array ⇝ *integer*
Indices in the input tuples (Regions1 or Regions2), respectively.

▷ **RegionIndex2** (output_control) ..... integer-array ⇝ *integer*
Indices in the input tuples (Regions1 or Regions2), respectively.

Result

The operator select_region_spatial returns the value 2 (H_MSG_TRUE) if Regions2 is not empty. The behavior in case of empty parameter Regions2 (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

area_center, intersection

See also

spatial_relation, find_neighbors, copy_obj, obj_to_integer

Module

Foundation

select_shape (Regions : SelectedRegions : Features, Operation, Min, Max :)

Choose regions with the aid of shape features.

The operator select_shape chooses regions according to shape. For each input region from Regions the indicated features (Features) are calculated. If each (Operation = *'and'*) or at least one (Operation = *'or'*) of the calculated features is within the default limits (Min,Max) the region is adapted into the output (duplicated). The parameters Min and Max can be set to *'min'* or *'max'* in order to leave bottom and top limits, respectively, open.

Condition: $Min_i \leq Feature_i(Object) \leq Max_i$

For an illustration of these features, please refer to the documentation of this chapter (Regions / Features).

Possible values for Features:

'area': Area of the object

'row': Row index of the center

'column': Column index of the center

'width': Width of the region (parallel to the coordinate axes; see height_width_ratio)
'height': Height of the region (parallel to the coordinate axes; see height_width_ratio)
'ratio': Ratio of the height and width of the region (parallel to the coordinate axes; see height_width_ratio)
'row1': Row index of upper left corner
'column1': Column index of upper left corner
'row2': Row index of lower right corner
'column2': Column index of lower right corner
'circularity': Circularity (see circularity)
'compactness': Compactness (see compactness)
'contlength': Total length of contour (see operator contlength)
'convexity': Convexity (see convexity)
'rectangularity': Rectangularity (see rectangularity)
'ra': Main radius of the equivalent ellipse (see elliptic_axis)
'rb': Secondary radius of the equivalent ellipse (see elliptic_axis)
'phi': Orientation of the equivalent ellipse (see elliptic_axis)
'anisometry': Anisometry (see eccentricity)
'bulkiness': Bulkiness (see operator eccentricity)
'struct_factor': Structure Factor (see operator eccentricity)
'outer_radius': Radius of smallest surrounding circle (see smallest_circle)
'inner_radius': Radius of largest inner circle (see inner_circle)
'inner_width': Width of the largest axis-parallel rectangle that fits into the region (see inner_rectangle1)
'inner_height': Height of the largest axis-parallel rectangle that fits into the region (see inner_rectangle1)
'dist_mean': Mean distance from the region border to the center (see operator roundness)
'dist_deviation': Deviation of the distance from the region border from the center (see operator roundness)
'roundness': Roundness (see operator roundness)
'num_sides': Number of polygon sides (see operator roundness)
'connect_num': Number of connection components (see operator connect_and_holes)
'holes_num': Number of holes (see operator connect_and_holes)
'area_holes': Area of the holes of the object (see operator area_holes)
'max_diameter': Maximum diameter of the region (see operator diameter_region)
'orientation': Orientation of the region (see operator orientation_region)
'euler_number': Euler number (see operator euler_number)
'rect2_phi': Orientation of the smallest surrounding rectangle (see operator smallest_rectangle2)
'rect2_len1': Half the length of the smallest surrounding rectangle (see operator smallest_rectangle2)
'rect2_len2': Half the width of the smallest surrounding rectangle (see operator smallest_rectangle2)
'moments_m11': Geometric moments of the region (see operator moments_region_2nd)
'moments_m20': Geometric moments of the region (see operator moments_region_2nd)
'moments_m02': Geometric moments of the region (see operator moments_region_2nd)
'moments_ia': Geometric moments of the region (see operator moments_region_2nd)
'moments_ib': Geometric moments of the region (see operator moments_region_2nd)
'moments_m11_invar': Geometric moments of the region (see operator moments_region_2nd_invar)
'moments_m20_invar': Geometric moments of the region (see operator moments_region_2nd_invar)
'moments_m02_invar': Geometric moments of the region (see operator moments_region_2nd_invar)
'moments_phi1': Geometric moments of the region (see operator moments_region_2nd_rel_invar)
'moments_phi2': Geometric moments of the region (see operator moments_region_2nd_rel_invar)

'moments_m21': Geometric moments of the region (see operator `moments_region_3rd`)
'moments_m12': Geometric moments of the region (see operator `moments_region_3rd`)
'moments_m03': Geometric moments of the region (see operator `moments_region_3rd`)
'moments_m30': Geometric moments of the region (see operator `moments_region_3rd`)
'moments_m21_invar': Geometric moments of the region (see operator `moments_region_3rd_invar`)
'moments_m12_invar': Geometric moments of the region (see operator `moments_region_3rd_invar`)
'moments_m03_invar': Geometric moments of the region (see operator `moments_region_3rd_invar`)
'moments_m30_invar': Geometric moments of the region (see operator `moments_region_3rd_invar`)
'moments_i1': Geometric moments of the region (see operator `moments_region_central`)
'moments_i2': Geometric moments of the region (see operator `moments_region_central`)
'moments_i3': Geometric moments of the region (see operator `moments_region_central`)
'moments_i4': Geometric moments of the region (see operator `moments_region_central`)
'moments_psi1': Geometric moments of the region (see operator `moments_region_central_invar`)
'moments_psi2': Geometric moments of the region (see operator `moments_region_central_invar`)
'moments_psi3': Geometric moments of the region (see operator `moments_region_central_invar`)
'moments_psi4': Geometric moments of the region (see operator `moments_region_central_invar`)

If only one feature (`Features`) is used the value of `Operation` is meaningless. Several features are processed in the sequence in which they are entered.

Parameters

▷ **`Regions`** (input_object) ..... region-array ⇝ *object*
Regions to be examined.

▷ **`SelectedRegions`** (output_object) ..... region-array ⇝ *object*
Regions fulfilling the condition.

▷ **`Features`** (input_control) ..... string(-array) ⇝ *string*
Shape features to be checked.
Default: 'area'
List of values: `Features` ∈ {'area', 'row', 'column', 'width', 'height', 'ratio', 'row1', 'column1', 'row2', 'column2', 'circularity', 'compactness', 'contlength', 'convexity', 'rectangularity', 'ra', 'rb', 'phi', 'anisometry', 'bulkiness', 'struct_factor', 'outer_radius', 'inner_radius', 'inner_width', 'inner_height', 'max_diameter', 'dist_mean', 'dist_deviation', 'roundness', 'num_sides', 'orientation', 'connect_num', 'holes_num', 'area_holes', 'euler_number', 'rect2_phi', 'rect2_len1', 'rect2_len2', 'moments_m11', 'moments_m20', 'moments_m02', 'moments_ia', 'moments_ib', 'moments_m11_invar', 'moments_m20_invar', 'moments_m02_invar', 'moments_phi1', 'moments_phi2', 'moments_m21', 'moments_m12', 'moments_m03', 'moments_m30', 'moments_m21_invar', 'moments_m12_invar', 'moments_m03_invar', 'moments_m30_invar', 'moments_i1', 'moments_i2', 'moments_i3', 'moments_i4', 'moments_psi1', 'moments_psi2', 'moments_psi3', 'moments_psi4'}

▷ **`Operation`** (input_control) ..... string ⇝ *string*
Linkage type of the individual features.
Default: 'and'
List of values: `Operation` ∈ {'and', 'or'}

▷ **`Min`** (input_control) ..... real(-array) ⇝ *real* / integer / string
Lower limits of the features or 'min'.
Default: 150.0
Value range: 0.0 ≤ `Min` ≤ 99999.0
Minimum increment: 0.001
Recommended increment: 1.0

▷ **`Max`** (input_control) ..... real(-array) ⇝ *real* / integer / string
Upper limits of the features or 'max'.
Default: 99999.0
Value range: 0.0 ≤ `Max` ≤ 99999.0
Minimum increment: 0.001
Recommended increment: 1.0
Restriction: Max >= Min

Example

```
* Where are the eyes of the Mandrill?
read_image(Image,'monkey')
threshold(Image,Region,128,255)
connection(Region,ConnectedRegions)
select_shape(ConnectedRegions,Eyes,['area','max_diameter'],\
             'and',[500,30.0],[1000,50.0])
dev_display(Eyes)
```

Result

The operator select_shape returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input objects available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, runlength_features

Possible Successors

select_gray, shape_trans, reduce_domain, count_obj

Alternatives

select_shape_std

See also

area_center, circularity, compactness, contlength, convexity, rectangularity, elliptic_axis, eccentricity, inner_circle, smallest_circle, smallest_rectangle1, smallest_rectangle2, inner_rectangle1, roundness, connect_and_holes, area_holes, diameter_region, orientation_region, moments_region_2nd, moments_region_2nd_invar, moments_region_2nd_rel_invar, moments_region_3rd, moments_region_3rd_invar, moments_region_central, moments_region_central_invar, select_obj

Module

Foundation

select_shape_proto (Regions, Pattern : SelectedRegions : Feature, Min, Max :)

Choose regions having a certain relation to each other.

The operator select_shape_proto selects regions based on certain relations between the regions. Every region from Regions is compared to the union of regions from Pattern. The limits (Min and Max) are specified absolutely or in percent (0..100), depending on the feature. Possible values for Feature are:

'distance_dilate' The minimum distance in the maximum norm from the edge of Pattern to the edge of every region from Regions is determined (see distance_rr_min_dil).

'distance_contour' The minimum Euclidean distance from the edge of Pattern to the edge of every region from Regions is determined. (see distance_rr_min).

'distance_center' The Euclidean distance from the center of Pattern to the center of every region from Regions is determined.

'covers' It is examined how well the region `Pattern` fits into the regions from `Regions`. If there is no shift so that `Pattern` is a subset of `Regions` the overlap is 0. If `Pattern` corresponds to the region after a corresponding shift the overlap is 100. Otherwise the area of the opening of `Regions` with `Pattern` is put into relation with the area of `Regions` (in percent).

'fits' It is examined whether `Pattern` can be shifted in such a way that it fits in `Regions`. If this is possible the corresponding region is copied from `Regions`. The parameters `Min` and `Max` are ignored.

'overlaps_abs' The area of the intersection of `Pattern` and every region in `Regions` is computed.

'overlaps_rel' The area of the intersection of `Pattern` and every region in `Regions` is computed. The relative overlap is the ratio of the area of the intersection, and the area of the respective region in `Regions` (in percent).

Parameters

▷ **`Regions`** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be examined.

▷ **`Pattern`** (input_object) . . . . . region(-array) ⇝ *object*
Region compared to `Regions`.

▷ **`SelectedRegions`** (output_object) . . . . . region(-array) ⇝ *object*
Regions fulfilling the condition.

▷ **`Feature`** (input_control) . . . . . string(-array) ⇝ *string*
Shape features to be checked.
Default: 'covers'
List of values: `Feature` ∈ {'distance_center', 'distance_dilate', 'distance_contour', 'covers', 'fits', 'overlaps_abs', 'overlaps_rel'}

▷ **`Min`** (input_control) . . . . . number ⇝ *real* / integer
Lower border of feature.
Default: 50.0
Suggested values: `Min` ∈ {0.0, 1.0, 5.0, 10.0, 20.0, 30.0, 50.0, 60.0, 70.0, 80.0, 90.0, 95.0, 99.0, 100.0, 200.0, 400.0}
Value range: 0.0 ≤ Min
Minimum increment: 0.001
Recommended increment: 5.0

▷ **`Max`** (input_control) . . . . . number ⇝ *real* / integer
Upper border of the feature.
Default: 100.0
Suggested values: `Max` ∈ {0.0, 10.0, 20.0, 30.0, 50.0, 60.0, 70.0, 80.0, 90.0, 95.0, 99.0, 100.0, 200.0, 300.0, 400.0}
Value range: 0.0 ≤ Max
Minimum increment: 0.001
Recommended increment: 5.0

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  if (argc < 2)
  {
    cout << "Usage: " << argv[0] << " <radius of circle>" << endl;
    exit (1);
  }
```

```
  double   rad = atof (argv[1]);
  HImage   img ("monkey");
  HWindow  w;

  img.Display (w);

  HRegion       circ = HRegion::GenCircle (100, 100, rad);
  HRegionArray  reg  = img.Regiongrowing (3, 3, 5, 0);
  HRegionArray  seg  = reg.SelectShapeProto (circ, "fits", 0, 0);

  w.SetColor  ("red");
  seg.Display (w);
  w.Click ();
  return(0);
}
```

Result

The operator `select_shape_proto` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`connection`, `draw_region`, `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`

Possible Successors

`select_gray`, `shape_trans`, `reduce_domain`, `count_obj`

Alternatives

`select_shape`

See also

`opening`, `erosion1`, `distance_rr_min_dil`, `distance_rr_min`

Module

Foundation

select_shape_std (Regions : SelectedRegions : Shape, Percent :)

Select regions of a given shape.

The operator `select_shape_std` compares the shape of the given regions with default shapes. If the region has a similar shape it is adopted into the output. Possible values for `Shape` are:

'max_area' The largest region is selected.

'rectangle1' The surrounding rectangle parallel to the coordinate axes is determined via the operator `smallest_rectangle1`. If the area difference in percent is larger than `Percent` the region is adopted.

'rectangle2' The smallest surrounding rectangle with any orientation is determined via the operator `smallest_rectangle2`. If the area difference in percent is larger than `Percent` the region is adopted. Note that as a more robust alternative the operator `select_shape` with `Feature` set to *'rectangularity'* can be used instead.

Parameters

▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
Input regions to be selected.

▷ **SelectedRegions** (output_object) . . . . . region(-array) ⇝ *object*
Regions with desired shape.

▷ **Shape** (input_control) . . . . . string ⇝ *string*
Shape features to be checked.
Default: 'max_area'
List of values: `Shape` ∈ {'max_area', 'rectangle1', 'rectangle2'}

▷ **Percent** (input_control) . . . . . real ⇝ *real*
Similarity measure.
Default: 70.0
Suggested values: `Percent` ∈ {10.0, 30.0, 50.0, 60.0, 70.0, 80.0, 90.0, 95.0, 100.0}
Value range: 0.0 ≤ `Percent` ≤ 100.0 (lin)
Minimum increment: 0.1
Recommended increment: 10.0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `smallest_rectangle1`, `smallest_rectangle2`

Alternatives

`intersection`, `complement`, `area_center`, `select_shape`

See also

`smallest_rectangle1`, `smallest_rectangle2`, `rectangularity`

Module

Foundation

smallest_circle (Regions : : : Row, Column, Radius)

Smallest surrounding circle of a region.

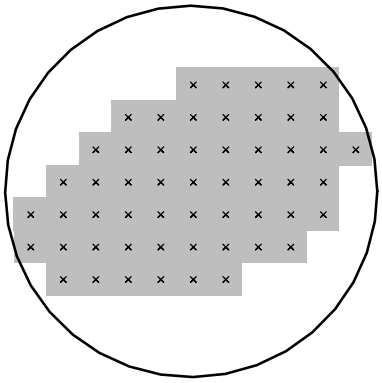
The operator `smallest_circle` determines the smallest surrounding circle of a region, i.e., the circle with the smallest area of all circles containing the region. For this circle the center (`Row`,`Column`) and the radius (`Radius`) are calculated. The operator is applied when, for example, the location and size of circular objects (e.g., coins) which, however, are not homogeneous inside or have broken edges due to bad segmentation, has to be determined. The output of the operator is selected in such a way that it can be used as input for the operators `disp_circle` and `gen_circle`.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions with varying outer and inner radii.

If several regions are passed in `Regions` corresponding tuples are returned as output parameter. In case of empty region all parameters have the value 0.0 if no other behavior was set (see `set_system`).

Attention

Internally, the calculation is based on the center coordinates of the region pixels. To take into account that pixels are not just infinitely small points but have a certain area, the calculated radius is enlarged by 0.5 before it is returned in `Radius`. This, in most cases, gives acceptable results. However, in the worst case (pixel diagonal) this enlargement is not sufficient. If one wants to ensure that the border of the input region completely lies within the circle, one had to enlarge the radius by $1/\sqrt{2}$ instead of 0.5. Consequently, the value returned in `Radius` must be corrected by $1/\sqrt{2} - 0.5$. However, this would also be only an upper bound, i.e., the circle with the corrected radius would be slightly too big in most cases.



The smallest surrounding circle of a region. Note that the calculation is based on the center coordinates of the region pixels and that 0.5 is added to the resulting radius.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Regions to be examined.

▷ **Row** (output_control) ..... circle.center.y(-array) ⇝ *real*
Line index of the center.

▷ **Column** (output_control) ..... circle.center.x(-array) ⇝ *real*
Column index of the center.

▷ **Radius** (output_control) ..... circle.radius(-array) ⇝ *real*
Radius of the surrounding circle.
Assertion: Radius >= 0

Example

```
read_image(Image,'fabrik')
regiongrowing(Image,Regions,5,5,6,100)
select_shape(Regions,SelectedRegions,'area','and',100,2000)
smallest_circle(SelectedRegions,Row,Column,Radius)
gen_circle(Circles,Row,Column,Radius)
dev_display (Circles)
```

Complexity

If F is the area of the region, then the mean runtime complexity is $O(sqrt(F)$.

Result

The operator `smallest_circle` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `runlength_features`

Possible Successors

`gen_circle`, `disp_circle`

Alternatives

`elliptic_axis`, `smallest_rectangle1`, `smallest_rectangle2`

See also

set_shape, select_shape, inner_circle

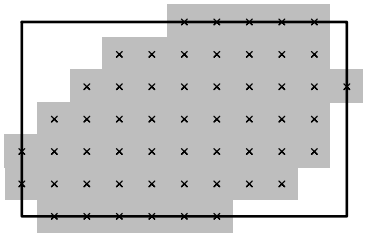
Module

Foundation

smallest_rectangle1 (Regions : : : Row1, Column1, Row2, Column2)

Surrounding rectangle parallel to the coordinate axes.

The operator smallest_rectangle1 calculates the surrounding rectangle of all input regions (parallel to the coordinate axes). The surrounding rectangle is described by the coordinates of the corner pixels (Row1,Column1,Row2,Column2). The calculation of the rectangle is based on the center coordinates of the region pixels.



The smallest axis aligned surrounding rectangle of a region. Note that the calculation is based on the center coordinates of the region pixels.

If more than one region is passed in Regions, the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input. In case of empty region all parameters have the value 0 if no other behavior was set (see set_system).

Attention

In case of empty region the result of Row1,Column1, Row2 and Column2 (all are 0) can lead to confusion.

Parameters

- ▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **Row1** (output_control) ..... rectangle.origin.y(-array) ⇝ *integer*
  Line index of upper left corner point.
- ▷ **Column1** (output_control) ..... rectangle.origin.x(-array) ⇝ *integer*
  Column index of upper left corner point.
- ▷ **Row2** (output_control) ..... rectangle.corner.y(-array) ⇝ *integer*
  Line index of lower right corner point.
- ▷ **Column2** (output_control) ..... rectangle.corner.x(-array) ⇝ *integer*
  Column index of lower right corner point.

Complexity

If F is the area of the region the mean runtime complexity is $O(sqrt(F))$.

Result

The operator smallest_rectangle1 returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection, runlength_features

Possible Successors

disp_rectangle1, gen_rectangle1

Alternatives

height_width_ratio, smallest_rectangle2, area_center

See also

select_shape

Module

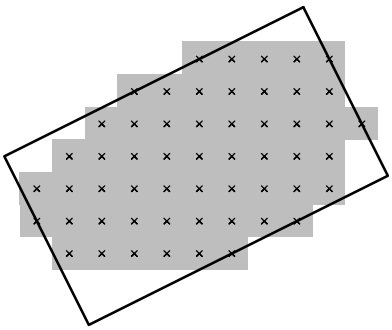
Foundation

smallest_rectangle2 (Regions : : : Row, Column, Phi, Length1, Length2)

Smallest surrounding rectangle with any orientation.

The operator smallest_rectangle2 determines the smallest surrounding rectangle of a region, i.e., the rectangle with the smallest area of all rectangles containing the region. For this rectangle the center, the inclination and the two radii are calculated. The calculation of the rectangle is based on the center coordinates of the region pixels.

In the documentation of this chapter (Regions / Features), you can find an image illustrating regions which vary in the length and phi of their smallest surrounding rectangle.



The smallest surrounding rectangle of a region. Note that the calculation is based on the center coordinates of the region pixels.

The operator is applied when, for example, the location of a scenery of several regions (e.g., printed text on a rectangular paper or in rectangular print (justified lines)) must be found. The parameters of smallest_rectangle2 are chosen in such a way that they can be used directly as input for the operators disp_rectangle2 and gen_rectangle2.

If more than one region is passed in Regions the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input. In case of empty region all parameters have the value 0.0 if no other behavior was set (see set_system).

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Regions to be examined.

▷ **Row** (output_control) ..... rectangle2.center.y(-array) ⇝ *real*
Line index of the center.

▷ **Column** (output_control) ..... rectangle2.center.x(-array) ⇝ *real*
Column index of the center.

▷ **Phi** (output_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real*
Orientation of the surrounding rectangle (arc measure)
Assertion: - pi / 2 < Phi && Phi <= pi / 2

▷ **Length1** (output_control) . . . . . rectangle2.hwidth(-array) ⇝ *real*
First radius (half length) of the surrounding rectangle.
Assertion: Length1 >= 0.0

▷ **Length2** (output_control) . . . . . rectangle2.hheight(-array) ⇝ *real*
Second radius (half width) of the surrounding rectangle.
Assertion: Length2 >= 0.0 && Length2 <= Length1

Example

```
read_image(Image,'fabrik')
regiongrowing(Image,Regions,5,5,6,100)
smallest_rectangle2(Regions,Row,Column,Phi,Length1,Length2)
gen_rectangle2(Rectangle,Row,Column,Phi,Length1,Length2)
dev_set_draw ('margin')
dev_display(Rectangle)
```

Complexity

If F is the area of the region and N is the number of supporting points of the convex hull, the runtime complexity is $O(sqrt(F) + N^2)$.

Result

The operator `smallest_rectangle2` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `runlength_features`

Possible Successors

`disp_rectangle2`, `gen_rectangle2`

Alternatives

`elliptic_axis`, `smallest_rectangle1`

See also

`smallest_circle`, `set_shape`

Module

Foundation

spatial_relation (Regions1, Regions2 : : Percent : RegionIndex1, RegionIndex2, Relation1, Relation2)

Pose relation of regions with regard to the coordinate axes.

The operator `spatial_relation` selects regions located by `Percent` percent “left”, “right”, “above” or “below” other regions. `Regions1` and `Regions2` contain the regions to be compared. `Regions1` can have three states:

- `Regions1` contains an empty object tuple, i.e., `count_obj` returns *0*:
  In this case all regions in `Regions2` are permutatively checked for neighborhood.

- Regions1 consists of one region:
  The regions of Regions1 are compared to all regions in Regions2.
- Regions1 consists of the same number of regions as Regions2:
  Regions1 and Regions2 are checked for a neighboring relation.

The percentage Percent is interpreted in such a way that the area of the second region has to be located really left/right or above/below the region margins of the first region by at least Percent percent. The indices of the regions that fulfill at least one of these conditions are then located at the n-th position in the output parameters RegionIndex1 and RegionIndex2. Additionally the output parameters Relation1 and Relation2 contain at the n-th position the type of relation of the region pair (RegionIndex1[n], RegionIndex2[n]), i.e., region with index RegionIndex2[n] has the relation Relation1[n] and Relation2[n] with region with index RegionIndex1[n].

Possible values for Relation1 and Relation2 are:

- Relation1: *'above'*, *'below'* or *''*
- Relation2: *'left'*, *'right'* or *''*

In RegionIndex1 and RegionIndex2 the indices of the regions in the tuples of the input regions (Regions1 or Regions2), respectively, are entered as image identifiers. Access to chosen regions via the index can be obtained by the operator copy_obj.

Parameters

▷ **Regions1** (input_object) . . . . . region(-array) ⇝ *object*
Starting regions.

▷ **Regions2** (input_object) . . . . . region(-array) ⇝ *object*
Comparative regions.

▷ **Percent** (input_control) . . . . . integer ⇝ *integer*
Percentage of the area of the comparative region which must be located left/right or above/below the region margins of the starting region.
Default: 50
Suggested values: Percent ∈ {0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
Value range: 0 ≤ Percent ≤ 100 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **RegionIndex1** (output_control) . . . . . integer-array ⇝ *integer*
Indices of the regions in the tuple of the input regions which fulfill the pose relation.

▷ **RegionIndex2** (output_control) . . . . . integer-array ⇝ *integer*
Indices of the regions in the tuple of the input regions which fulfill the pose relation.

▷ **Relation1** (output_control) . . . . . string-array ⇝ *string*
Vertical pose relation in which RegionIndex2[n] stands with RegionIndex1[n].

▷ **Relation2** (output_control) . . . . . string-array ⇝ *string*
Horizontal pose relation in which RegionIndex2[n] stands with RegionIndex1[n].

Result

The operator spatial_relation returns the value 2 (H_MSG_TRUE) if Regions2 is not empty and Percent is correctly chosen. The behavior in case of empty parameter Regions2 (no input regions available) is set via the operator set_system('no_object_result',<Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

area_center, intersection

See also

select_region_spatial, find_neighbors, copy_obj, obj_to_integer

Module

Foundation

23.4 Geometric Transformations

affine_trans_region (Region : RegionAffineTrans : HomMat2D, Interpolate :)

Apply an arbitrary affine 2D transformation to regions.

affine_trans_region applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the regions given in Region and returns the transformed regions in RegionAffineTrans. The affine transformation is described by the homogeneous transformation matrix given in HomMat2D, which can be created using the operators hom_mat2d_identity, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_translate, etc., or be the result of operators like vector_angle_to_rigid.

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to x and the *column* coordinate corresponds to y of the coordinate system in which the transformation matrix was defined. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

The parameter Interpolate determines whether the transformation is to be done by using interpolation internally. The interpolation modes *'nearest_neighbor'* and *'constant'*, which are described in detail for affine_trans_image, can be used. An interpolation can lead to smoother region boundaries, especially if regions are enlarged. However, the runtime increases drastically.

Attention

affine_trans_region in general is not reversible (clipping and discretization during rotation and scaling).

affine_trans_region does not use the HALCON standard coordinate system (with the origin in the center of the upper left pixel), but instead uses the same coordinate system as in affine_trans_pixel, i.e., the origin lies in the upper left corner of the upper left pixel. Therefore, applying affine_trans_region corresponds to a chain of transformations (see affine_trans_pixel), which is applied to each point of the region (input and output pixels as homogeneous vectors). As an effect, you might get unexpected results when creating affine transformations based on coordinates that are derived from the region, e.g., by operators like area_center. For example, if you use this operator to calculate the center of gravity of a rotationally symmetric region and then rotate the region around this point using hom_mat2d_rotate, the resulting region will not lie on the original one. In such a case, you can compensate this effect by applying the following translations to HomMat2D before using it in affine_trans_region:

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
affine_trans_region(Region, RegionAffineTrans, HomMat2DAdapted,
'nearest_neighbor')
```

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Region(s) to be rotated and scaled.

▷ **RegionAffineTrans** (output_object) . . . . . region(-array) ⇝ *object*
Transformed output region(s).
Number of elements: RegionAffineTrans == Region

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Interpolate** (input_control) . . . . . string ⇝ *string*
Should the transformation be done using interpolation?
Default: 'nearest_neighbor'
List of values: `Interpolate` ∈ {'constant', 'nearest_neighbor'}

Result

If the matrix `HomMat2D` represents an affine transformation (i.e., not a projective transformation), `affine_trans_region` returns 2 (`H_MSG_TRUE`). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result',<Result>)`, the behavior in case of an empty input region via `set_system('empty_region_result',<Result>)`, and the behavior in case of an empty result region via `set_system('store_empty_region',<'true'/'false'>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`hom_mat2d_identity`, `hom_mat2d_scale`, `hom_mat2d_translate`, `hom_mat2d_invert`, `hom_mat2d_rotate`, `hom_mat2d_reflect`

Possible Successors

`select_shape`

Alternatives

`move_region`, `mirror_region`, `zoom_region`

See also

`affine_trans_image`

Module

Foundation

mirror_region (Region : RegionMirror : Mode, WidthHeight :)

Reflect a region about an axis.

`mirror_region` reflects a region about one of three possible axes. If `Mode` is set to *'row'*, it is reflected about the horizontal axis, if `Mode` is set to *'column'*, about the vertical axis, and if `Mode` is set to *'diagonal'*, about the main diagonal $x = y$.

For `Mode` = *'row'* or *'column'* the parameter `WidthHeight` specifies two times the coordinate of the axis of symmetry. Hence, if `Region` has been extracted from an image and should be mirrored in a way such as if it had been extracted from a mirrored version of this image, `WidthHeight` corresponds to one of the dimensions of this image (according to `Mode`). If `Mode` = *'diagonal'*, the parameter `WidthHeight` is not used.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Region(s) to be reflected.

▷ **RegionMirror** (output_object) ..... region(-array) ⇝ *object*
Reflected region(s).
Number of elements: RegionMirror == Region

▷ **Mode** (input_control) ..... string ⇝ *string*
Axis of symmetry.
Default: 'row'
List of values: Mode ∈ {'column', 'row', 'diagonal'}

▷ **WidthHeight** (input_control) ..... integer ⇝ *integer*
Twice the coordinate of the axis of symmetry.
Default: 512
Suggested values: WidthHeight ∈ {128, 256, 512, 525, 768, 1024}
Value range: 1 ≤ WidthHeight ≤ 1024 (lin)
Minimum increment: 1
Recommended increment: 1
Restriction: WidthHeight > 0

Example

```
read_image(&Image,"monkey");
threshold(Image,&Seg,128.0,255.0);
mirror_region(Seg,&Mirror,"row",512);
disp_region(Mirror,WindowHandle);
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

Alternatives

hom_mat2d_reflect, affine_trans_region

See also

zoom_region

Module

Foundation

move_region (Region : RegionMoved : Row, Column :)

Translate a region.

move_region translates the input regions by the vector given by (Row, Column). If necessary, the resulting regions are clipped with the current image format.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be moved.

▷ **RegionMoved** (output_object) ..... region(-array) ⇝ *object*
Translated region(s).
Number of elements: RegionMoved == Region

▷ **Row** (input_control) . . . . . point.y ⇝ *integer*
Row coordinate of the vector by which the region is to be moved.
Default: 30
Suggested values: Row ∈ {-128, -64, -32, -16, -10, -8, -4, -2, -1, 0, 1, 2, 4, 5, 8, 10, 16, 32, 64, 128}
Value range: -512 ≤ Row ≤ 512 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Column** (input_control) . . . . . point.x ⇝ *integer*
Row coordinate of the vector by which the region is to be moved.
Default: 30
Suggested values: Column ∈ {-128, -64, -32, -16, -10, -8, -4, -2, -1, 0, 1, 2, 4, 5, 8, 10, 16, 32, 64, 128}
Value range: -512 ≤ Column ≤ 512 (lin)
Minimum increment: 1
Recommended increment: 10

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(F)$.

Result

move_region always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>), the behavior in case of an empty input region via set_system('empty_region_result',<Result>), and the behavior in case of an empty result region via set_system('store_empty_region',<'true'/'false'>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

See also

affine_trans_image, mirror_region, zoom_region

Module

Foundation

polar_trans_region (Region : PolarTransRegion : Row, Column, AngleStart, AngleEnd, RadiusStart, RadiusEnd, Width, Height, Interpolation :)

Transform a region within an annular arc to polar coordinates.

polar_trans_region transforms a Region within the annular arc specified by the center point (Row, Column), the radii RadiusStart and RadiusEnd and the angles AngleStart and AngleEnd to its polar coordinate version in a virtual image of the dimensions Width × Height.

The polar transformation is a change of the coordinate system. Instead of a row and a column coordinate, each point's position is expressed by its radius r (i.e. the distance to the center point Row, Column) and the angle phi between the column axis (through the center point) and the line from the center point towards the point. Note that this transformation is not affine.

The coordinate $(0, 0)$ in the output region always corresponds to the point in the input region that is specified by RadiusStart and AngleStart. Analogously, the coordinate $(\text{Height} - 1, \text{Width} - 1)$ corresponds to the point in the input region that is specified by RadiusEnd and AngleEnd. In the usual mode (AngleStart < AngleEnd and RadiusStart < RadiusEnd), the polar transformation is performed in the mathematically positive orientation (counterclockwise). Furthermore, points with smaller radii lie in the upper part of the

output region. By suitably exchanging the values of these parameters (e.g., AngleStart > AngleEnd or RadiusStart > RadiusEnd), any desired orientation of the output region can be achieved.

The angles can be chosen from all real numbers. Center point and radii can be real as well. However, if they are both integers and the difference of RadiusEnd and RadiusStart equals Height − 1, calculation will be sped up through an optimized routine.

The radii and angles are inclusive, which means that the first row of the virtual target image contains the circle with radius RadiusStart and the last row contains the circle with radius RadiusEnd. For complete circles, where the difference between AngleStart and AngleEnd equals 2π (360 degrees), this also means that the first column of the target image will be the same as the last.

To avoid this, do not make this difference 2π, but $2\pi(1 - \frac{1}{\text{Width}})$ degrees instead.

The parameter Interpolation is used to select the interpolation method *'bilinear'* or *'nearest_neighbor'*. Setting Interpolation to *'bilinear'* leads to smoother region boundaries, especially if regions are enlarged. However, the runtime increases significantly.

If more than one region is passed in Region, their polar transformations are computed individually and stored as a tuple in PolarTransRegion. Please note that the indices of an input region and its transformation only correspond if the system variable *'store_empty_regions'* is set to *'true'* (see set_system). Otherwise empty output regions are discarded and the length of the input tuple Region is most likely not equal to the length of the output tuple PolarTransRegion.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Attention

If Width or Height are chosen greater than the dimensions of the current image, the system variable *'clip_region'* should be set to *'false'* (see set_system). Otherwise, an output region that does not lie within the dimensions of the current image can produce an error message.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Input region.

▷ **PolarTransRegion** (output_object) . . . . . region(-array) ⇝ *object*
Output region.

▷ **Row** (input_control) . . . . . number ⇝ *real* / integer
Row coordinate of the center of the arc.
Default: 256
Suggested values: Row ∈ {0, 16, 32, 64, 128, 240, 256, 480, 512}

▷ **Column** (input_control) . . . . . number ⇝ *real* / integer
Column coordinate of the center of the arc.
Default: 256
Suggested values: Column ∈ {0, 16, 32, 64, 128, 256, 320, 512, 640}

▷ **AngleStart** (input_control) . . . . . angle.rad ⇝ *real*
Angle of the ray to be mapped to column coordinate *0* of PolarTransRegion.
Default: 0.0
Suggested values: AngleStart ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853, 12.566370616}
Value range: -6.2831853 ≤ AngleStart ≤ 6.2831853

▷ **AngleEnd** (input_control) . . . . . angle.rad ⇝ *real*
Angle of the ray to be mapped to column coordinate Width − 1 of PolarTransRegion.
Default: 6.2831853
Suggested values: AngleEnd ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853, 12.566370616}
Value range: -6.2831853 ≤ AngleEnd ≤ 6.2831853

▷ **RadiusStart** (input_control) . . . . . number ⇝ *real* / integer
Radius of the circle to be mapped to row coordinate *0* of PolarTransRegion.
Default: 0
Suggested values: RadiusStart ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ RadiusStart ≤ 32767

▷ **RadiusEnd** (input_control) ..... number ⇝ *real / integer*
Radius of the circle to be mapped to row coordinate Height − 1 of PolarTransRegion.
Default: 100
Suggested values: RadiusEnd ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ RadiusEnd ≤ 32767

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the virtual output image.
Default: 512
Suggested values: Width ∈ {256, 320, 512, 640, 800, 1024}
Value range: 2 ≤ Width ≤ 32767

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the virtual output image.
Default: 512
Suggested values: Height ∈ {240, 256, 480, 512, 600, 1024}
Value range: 2 ≤ Height ≤ 32767

▷ **Interpolation** (input_control) ..... string ⇝ *string*
Interpolation method for the transformation.
Default: 'nearest_neighbor'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

See also

polar_trans_image_ext, polar_trans_image_inv, polar_trans_region_inv, polar_trans_contour_xld, polar_trans_contour_xld_inv

Module

Foundation

polar_trans_region_inv (PolarRegion : XYTransRegion : Row, Column, AngleStart, AngleEnd, RadiusStart, RadiusEnd, WidthIn, HeightIn, Width, Height, Interpolation :)

Transform a region in polar coordinates back to Cartesian coordinates.

polar_trans_region_inv transforms the polar coordinate representation of a region, stored in PolarRegion, back onto an annular arc in Cartesian coordinates, described by the radii RadiusStart and RadiusEnd and the angles AngleStart and AngleEnd with the center point located at (Row, Column). All of these values can be chosen as real numbers. In addition, the dimensions of the virtual image containing the region PolarRegion must be given in WidthIn and HeightIn. WidthIn − 1 is the column coordinate corresponding to AngleEnd and HeightIn − 1 is the row coordinate corresponding to RadiusEnd. AngleStart and RadiusStart correspond to column and row coordinate *0*. Furthermore, the dimensions Width and Height of the virtual output image containing the transformed region XYTransRegion are required.

The angles and radii are inclusive, which means that the row coordinate *0* in PolarRegion will be mapped onto a a circle with a distance of RadiusStart pixels from the specified center and the row with the coordinate HeightIn − 1 will be mapped onto a circle of radius RadiusEnd. This applies to AngleStart, AngleEnd, and WidthIn in an analogous way. If the width of the input region PolarRegion corresponds to an angle interval greater than $2pi$, the region is cropped such that length of this interval is $2pi$.

The parameter Interpolation is used to select the interpolation method *'bilinear'* or *'nearest_neighbor'*. Setting Interpolation to *'bilinear'* leads to smoother region boundaries, especially if regions are enlarged. However, the runtime increases significantly.

polar_trans_region_inv is the inverse function of polar_trans_region.

The call sequence:

```
polar_trans_region(Region, PolarRegion, Row, Column, rad(360)
polar_trans_region_inv(PolarRegion, XYTransRegion, Row,
Column, rad(360), 0, 0, Radius, Width, Height, Width, Height,
'nearest_neighbor')
```

returns the region Region, restricted to the circle around (Row, Column) with radius Radius, as its output region XYTransRegion.

If more than one region is passed in PolarRegion, their Cartesian transformations are computed individually and stored as a tuple in XYTransRegion. Please note that the indices of an input region and its transformation only correspond if the system variable *'store_empty_regions'* is set to *'false'* (see set_system). Otherwise empty output regions are discarded and the length of the input tuple PolarRegion is most likely not equal to the length of the output tuple XYTransRegion.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Attention

If Width or Height are chosen greater than the dimensions of the current image, the system variable *'clip_region'* should be set to *'false'* (see set_system). Otherwise, an output region that does not lie within the dimensions of the current image can produce an error message.

Parameters

- ▷ **PolarRegion** (input_object) ..... region(-array) ⇝ *object*
  Input region.
- ▷ **XYTransRegion** (output_object) ..... region(-array) ⇝ *object*
  Output region.
- ▷ **Row** (input_control) ..... number ⇝ *real* / integer
  Row coordinate of the center of the arc.
  Default: 256
  Suggested values: Row ∈ {0, 16, 32, 64, 128, 240, 256, 480, 512}
  Value range: -131068 ≤ Row ≤ 131068
- ▷ **Column** (input_control) ..... number ⇝ *real* / integer
  Column coordinate of the center of the arc.
  Default: 256
  Suggested values: Column ∈ {0, 16, 32, 64, 128, 256, 320, 512, 640}
  Value range: -131068 ≤ Column ≤ 131068
- ▷ **AngleStart** (input_control) ..... angle.rad ⇝ *real*
  Angle of the ray to map the column coordinate *0* of PolarRegion to.
  Default: 0.0
  Suggested values: AngleStart ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853}
  Value range: -6.2831853 ≤ AngleStart ≤ 6.2831853
- ▷ **AngleEnd** (input_control) ..... angle.rad ⇝ *real*
  Angle of the ray to map the column coordinate WidthIn − 1 of PolarRegion to.
  Default: 6.2831853
  Suggested values: AngleEnd ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853}
  Value range: -6.2831853 ≤ AngleEnd ≤ 6.2831853
- ▷ **RadiusStart** (input_control) ..... number ⇝ *real* / integer
  Radius of the circle to map the row coordinate *0* of PolarRegion to.
  Default: 0
  Suggested values: RadiusStart ∈ {0, 16, 32, 64, 100, 128, 256, 512}
  Value range: 0 ≤ RadiusStart ≤ 32767
- ▷ **RadiusEnd** (input_control) ..... number ⇝ *real* / integer
  Radius of the circle to map the row coordinate HeightIn − 1 of PolarRegion to.
  Default: 100
  Suggested values: RadiusEnd ∈ {0, 16, 32, 64, 100, 128, 256, 512}
  Value range: 0 ≤ RadiusEnd ≤ 32767

▷ **WidthIn** (input_control) ..... extent.x ⇝ *integer*
Width of the virtual input image.
Default: 512
Suggested values: WidthIn ∈ {256, 320, 512, 640, 800, 1024}
Value range: 2 ≤ WidthIn ≤ 32767

▷ **HeightIn** (input_control) ..... extent.y ⇝ *integer*
Height of the virtual input image.
Default: 512
Suggested values: HeightIn ∈ {240, 256, 480, 512, 600, 1024}
Value range: 2 ≤ HeightIn ≤ 32767

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the virtual output image.
Default: 512
Suggested values: Width ∈ {256, 320, 512, 640, 800, 1024}
Value range: 1 ≤ Width ≤ 32767

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the virtual output image.
Default: 512
Suggested values: Height ∈ {240, 256, 480, 512, 600, 1024}
Value range: 1 ≤ Height ≤ 32767

▷ **Interpolation** (input_control) ..... string ⇝ *string*
Interpolation method for the transformation.
Default: 'nearest_neighbor'
List of values: Interpolation ∈ {'nearest_neighbor', 'bilinear'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

See also

polar_trans_image_ext, polar_trans_image_inv, polar_trans_region, polar_trans_contour_xld, polar_trans_contour_xld_inv

Module

Foundation

projective_trans_region (Regions : TransRegions : HomMat2D, Interpolation :)

Apply a projective transformation to a region.

projective_trans_region applies the projective transformation specified by the homogeneous matrix HomMat2D on the regions in Regions and returns the transformed regions in TransRegions.

For creation and interpretation details of this matrix see also projective_trans_image.

If *'clip_region'* is set to its default value *'true'* by set_system('clip_region', 'true') or if the transformation is degenerated and thus produces infinite regions, the output region is clipped by the rectangle with upper left corner (0, 0) and lower right corner (*'width'*, *'height'*), where *'width'* and *'height'* are system variables (see also get_system). If *'clip_region'* is *'false'*, the output region is not clipped except by the maximum supported coordinate size. See the "Installation Guide" for further information about coordinate range limitations. This may result in extremely memory and time intensive computations, so use with care.

Attention

The used coordinate system is the same as in affine_trans_pixel. This means that in fact not HomMat2D is applied but a modified version. Therefore, applying projective_trans_region corresponds to the following chain of transformations, which is applied to each point (Row_i, Col_i) of the region (input and output pixels as homogeneous vectors):

$$\begin{pmatrix} RowTrans_i \\ ColTrans_i \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & -0.5 \\ 0 & 1 & -0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D} \cdot \begin{bmatrix} 1 & 0 & +0.5 \\ 0 & 1 & +0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} Row_i \\ Col_i \\ 1 \end{pmatrix}$$

As an effect, you might get unexpected results when creating projective transformations based on coordinates that are derived from the region, e.g., by operators like `area_center`. For example, if you use this operator to calculate the center of gravity of a rotationally symmetric region and then rotate the region around this point using `hom_mat2d_rotate`, the resulting region will not lie on the original one. In such a case, you can compensate this effect by applying the following translations to `HomMat2D` before using it in `projective_trans_region`:

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
projective_trans_region(Region, TransRegion, HomMat2DAdapted,
'bilinear')
```

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Regions** (input_object) ..... region(-array) ⇝ *object*
Input regions.

▷ **TransRegions** (output_object) ..... region(-array) ⇝ *object*
Output regions.

▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
Homogeneous projective transformation matrix.

▷ **Interpolation** (input_control) ..... string ⇝ *string*
Interpolation method for the transformation.
Default: 'bilinear'
List of values: `Interpolation` ∈ {'nearest_neighbor', 'bilinear'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`vector_to_proj_hom_mat2d`, `hom_vector_to_proj_hom_mat2d`, `proj_match_points_ransac`, `proj_match_points_ransac_guided`, `hom_mat3d_project`

See also

`projective_trans_image`, `projective_trans_image_size`, `projective_trans_contour_xld`, `projective_trans_point_2d`, `projective_trans_pixel`

Module

Foundation

transpose_region (Region : Transposed : Row, Column :)

Reflect a region about a point.

`transpose_region` reflects a region about a point. The fixed point is given by `Column` and `Row`. The image P' of a point P is determined by the following requirement:

If $P = S$, then $P' = S$, i.e., the point S is the fixed point of the mapping. If $P \neq S$, S is the center point of a line segment connecting P and P'. Therefore, the following equations result:

$$\begin{aligned} \texttt{Column} &= \frac{x + x'}{2} \\ \texttt{Row} &= \frac{y + y'}{2}. \end{aligned}$$

If `Row` and `Column` are set to the origin, the in morphology often used transposition results. Hence `transpose_region` is often used to reflect (transpose) a structuring element.

Parameters

▷ **`Region`** (input_object) . . . . . region(-array) ⇝ *object*
Region to be reflected.

▷ **`Transposed`** (output_object) . . . . . region(-array) ⇝ *object*
Transposed region.

▷ **`Row`** (input_control) . . . . . point.y ⇝ *integer*
Row coordinate of the reference point.
Default: 0
Suggested values: `Row` ∈ {0, 64, 128, 256, 512}
Value range: 0 ≤ `Row` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **`Column`** (input_control) . . . . . point.x ⇝ *integer*
Column coordinate of the reference point.
Default: 0
Suggested values: `Column` ∈ {0, 64, 128, 256, 512}
Value range: 0 ≤ `Column` ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let F be the area of the input region. Then the runtime complexity for one region is

$$O(\sqrt{F}) \ .$$

Result

`transpose_region` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result',<RegionResult>)`
- empty region: `set_system('empty_region_result',<RegionResult>)`

Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

See also

`dilation1`, `opening`, `closing`

Module

Foundation

zoom_region (Region : RegionZoom : ScaleWidth, ScaleHeight :)

Zoom a region.

zoom_region enlarges or reduces the regions given in Region in the x- and y-direction by the given scale factors ScaleWidth and ScaleHeight.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be zoomed.

▷ **RegionZoom** (output_object) ..... region(-array) ⇝ *object*
Zoomed region(s).
Number of elements: RegionZoom == Region

▷ **ScaleWidth** (input_control) ..... extent.x ⇝ *real*
Scale factor in x-direction.
Default: 2.0
Suggested values: ScaleWidth ∈ {0.25, 0.5, 1.0, 2.0, 3.0}
Value range: 0.0 ≤ ScaleWidth ≤ 100.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.5

▷ **ScaleHeight** (input_control) ..... extent.y ⇝ *real*
Scale factor in y-direction.
Default: 2.0
Suggested values: ScaleHeight ∈ {0.25, 0.5, 1.0, 2.0, 3.0}
Value range: 0.0 ≤ ScaleHeight ≤ 100.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.5

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

See also

zoom_image_size, zoom_image_factor

Module

Foundation

23.5 Sets

complement (Region : RegionComplement : :)

Return the complement of a region.

complement determines the complement of the input region(s).

If the system flag *'clip_region'* is *'true'*, which is the default, the difference of the largest image processed so far (see reset_obj_db) and the input region is returned.

If the system flag *'clip_region'* is *'false'* (see set_system), the resulting region would be infinitely large. To avoid this, the complement is done only virtually by setting the complement flag of Region to TRUE. For succeeding operations the de Morgan laws are applied while calculating results. Using complement with *'clip_region'*

set to *'false'* makes sense only to avoid fringe effects, e.g., if the area of interest is bigger or smaller than the image. For the latter case, the clipping would be set explicitly. If there is no reason to use the operator with *'clip_region'='false'* but you need the flag for other operations of your program, it is recommended to temporarily set the system flag to*'true'* and change it back to *'false'* after applying `complement`. Otherwise, negative regions may result from succeeding operations.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Input region(s).

▷ **RegionComplement** (output_object) . . . . . region(-array) ⇝ *object*
Complemented regions.
Number of elements: RegionComplement == Region

Result

`complement` always returns the value 2 (`H_MSG_TRUE`). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result',<Result>)` and the behavior in case of an empty input region via `set_system('empty_region_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `connection`, `regiongrowing`, `pouring`, `class_ndim_norm`

Possible Successors

`select_shape`

See also

`difference`, `union1`, `union2`, `intersection`, `reset_obj_db`, `set_system`

Module

Foundation

difference (Region, Sub : RegionDifference : :)

Calculate the difference of two regions.

`difference` calculates the set-theoretic difference of two regions:

(Regions in `Region`) − (Regions in `Sub`)

The resulting region is defined as the input region (`Region`) with all points from `Sub` removed. Note that, internally, all regions of `Sub` are united to a single region before the differences between the individual regions of `Region` and the united region are calculated.

Attention

Empty regions are valid for both parameters. On output, empty regions may result. The value of the system flag *'store_empty_region'* determines the behavior in this case.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be processed.

▷ **Sub** (input_object) . . . . . region(-array) ⇝ *object*
The union of these regions is subtracted from Region.

▷ **RegionDifference** (output_object) . . . . . region(-array) ⇝ *object*
Resulting region.

Example

```
* provides the region X without the points in Y
difference(X,Y,RegionDifference)
```

Complexity

Let N be the number of regions, F_1 be their average area, and F_2 be the total area of all regions in `Sub`. Then the runtime complexity is $O(F_1 * \log(F_1) + N * (\sqrt{F_1} + \sqrt{F_2}))$.

Result

`difference` always returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result',<Result>)` and the behavior in case of an empty input region via `set_system('empty_region_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`threshold`, `connection`, `regiongrowing`, `pouring`, `class_ndim_norm`

Possible Successors

`select_shape`, `disp_region`

See also

`intersection`, `union1`, `union2`, `complement`, `symm_difference`

Module

Foundation

intersection (Region1, Region2 : RegionIntersection : :)

Calculate the intersection of two regions.

`intersection` calculates the intersection of the regions in `Region1` with the regions in `Region2`. Each region in `Region1` is intersected with all regions in `Region2`. That is, internally all regions of `Region2` are united to a single region before the individual regions of `Region1` are intersected with the united region. The order of regions in `RegionIntersection` is identical to the order of regions in `Region1`.

Attention

Empty input regions are permitted. Because empty result regions are possible, the system flag *'store_empty_region'* should be set appropriately.

Parameters

▷ **Region1** (input_object) ..... region(-array) ⇝ *object*
Regions to be intersected with all regions in Region2.

▷ **Region2** (input_object) ..... region(-array) ⇝ *object*
Regions with which Region1 is intersected.

▷ **RegionIntersection** (output_object) ..... region(-array) ⇝ *object*
Result of the intersection.
Number of elements: RegionIntersection <= Region1

Complexity

Let N be the number of regions in `Region1`, F_1 be their average area, and F_2 be the total area of all regions in `Region2`. Then the runtime complexity is $O(F_1 \log(F_1) + N * (\sqrt{F_1} + \sqrt{F_2}))$.

Result

`intersection` always returns `2` (`H_MSG_TRUE`). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result',<Result>)` and the behavior in case of an empty input region via `set_system('empty_region_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

See also

union1, union2, complement

Module

Foundation

symm_difference (Region1, Region2 : RegionDifference : :)

Calculate the symmetric difference of two regions.

symm_difference calculates the symmetric difference of two regions. Note that, internally, all regions of Region2 are united to a single region before the symmetric differences between the individual regions of Region1 and the united region are calculated. Two possible definitions of the symmetric difference can be seen in the example below. A third definition is to regard the exclusive or of the two regions.

Attention

Empty regions are valid for both parameters. On output, empty regions may result. The value of the system flag *'store_empty_region'* determines the behavior in this case.

Parameters

▷ **Region1** (input_object) ..... region(-array) ⇝ *object*
Input region 1.

▷ **Region2** (input_object) ..... region(-array) ⇝ *object*
Input region 2.

▷ **RegionDifference** (output_object) ..... region(-array) ⇝ *object*
Resulting region.

Example

```
* Simulate the symmetric difference of Region1 and Region2 with
* difference and union:
difference(Region1, Region2, Diff1)
difference(Region2, Region1, Diff2)
union2(Diff1, Diff2, Difference)

* Simulate the symmetric difference of Region1 and Region2 with
* union, intersection, and difference:
union2(Region1, Region2, Union)
intersection(Region1, Region2, Intersection)
difference(Union, Intersection, Difference)
```

Result

symm_difference always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

select_shape, disp_region

See also

intersection, union1, union2, complement, difference

Module

Foundation

union1 (Region : RegionUnion : :)

Return the union of all input regions.

union1 computes the union of all input regions and returns the result in RegionUnion.

Parameters

▷ **Region** (input_object) ..... region-array ⇝ *object*
Regions of which the union is to be computed.

▷ **RegionUnion** (output_object) ..... region ⇝ *object*
Union of all input regions.
Number of elements: RegionUnion <= Region

Example

```
* Union of segmentation results:
threshold(Image,Region1,128,255)
dyn_threshold(Image,Mean,Region2,5,'light')
concat_obj(Region1,Region2,Regions)
union1(Regions,RegionUnion)
```

Complexity

Let F be the sum of all areas of the input regions. Then the runtime complexity is $O(log(sqrt(F)) * sqrt(F))$.

Result

union1 always returns 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

Alternatives

union2

See also

intersection, complement

Module

Foundation

union2 (Region1, Region2 : RegionUnion : :)

Return the union of two regions.

union2 computes the union of the region in Region1 with all regions in Region2. Internally, all regions of Region2 are united to a single region before the individual regions of Region1 are united with the already united region. This means that union2 is not commutative!

Parameters

- ▷ **Region1** (input_object) ..... region(-array) ⇝ *object*
  Region for which the union with all regions in Region2 is to be computed.
- ▷ **Region2** (input_object) ..... region(-array) ⇝ *object*
  Regions which should be added to Region1.
- ▷ **RegionUnion** (output_object) ..... region(-array) ⇝ *object*
  Resulting regions.
  Number of elements: RegionUnion == Region1

Complexity

Let F be the sum of all areas of the input regions. Then the runtime complexity is $O(log(sqrt(F)) * sqrt(F))$.

Result

union2 always returns 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

Alternatives

union1

See also

intersection, complement

Module

Foundation

23.6 Tests

test_equal_region (Regions1, Regions2 : : : IsEqual)

Test whether the regions of two objects are identical.

The operator test_equal_region compares the regions of the two input parameters. The n-th element in Regions1 is compared to the n-th object in Regions2 (for all n). If all regions are equal and the number of regions is identical the operator IsEqual is set to TRUE, otherwise FALSE.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Parameters

▷ **Regions1** (input_object) . . . . . region(-array) ⇝ *object*
Test regions.

▷ **Regions2** (input_object) . . . . . region(-array) ⇝ *object*
Comparative regions.
Number of elements: Regions1 == Regions2

▷ **IsEqual** (output_control) . . . . . integer ⇝ *integer*
Boolean result value.

Complexity

If F is the area of a region the runtime complexity is $O(1)$ or $O(sqrt(F))$ if the result is TRUE, $O(sqrt(F))$ if the result is FALSE.

Result

The operator `test_equal_region` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. The behavior in case of empty input (no input objects available) is set via the operator `set_system(::'no_object_result',<Result>:)`. If the number of objects differs an exception is raised. Else `test_equal_region` returns the value `2` (`H_MSG_TRUE`)

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`intersection`, `complement`, `area_center`

See also

`test_equal_obj`

Module

Foundation

test_region_point (Regions : : Row, Column : IsInside)

Test if the region contains a given point.

If an array of regions and one test point is provided, `test_region_point` tests if at least one input region of `Regions` contains the single test point given in (`Row`,`Column`). If this is the case, `IsInside` is set to TRUE, else to FALSE. Alternatively, if a single region and several test points are provided, `test_region_point` tests if at least one of the test points (given in `Row`,`Column`) is contained in the input region given in `Regions`. If this is the case, `IsInside` is set to TRUE, otherwise it is set to FALSE. If (`Row`,`Column`) are real-valued, rounding is performed internally since regions are pixel precise.

To test, which points exactly are contained in a region, use the operator `test_region_points`.

Attention

The test pixel is not contained in an empty region (no pixel of the region corresponds to the pixel). If all regions are empty `IsInside` is set to FALSE.

In case of an empty input object (empty tuple) `Regions` and `set_system('no_object_result','true')`, an empty tuple is returned in `IsInside`. If `set_system('no_object_result','false')` was set, an exception is raised.

Parameters

▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Row** (input_control) . . . . . point.y(-array) ⇝ *integer* / real
Row index of the test pixel(s).
Default: 100

▷ **Column** (input_control) . . . . . point.x(-array) ⇝ *integer* / real
Column index of the test pixel(s).
Number of elements: Row == Column
Default: 100

▷ **IsInside** (output_control) . . . . . integer ⇝ *integer*
Boolean result value.

Complexity

If F is the area of one region, N is the number of regions, and M is the number of test points, the runtime complexity is $O(ln(sqrt(F)) * N)$ in the case of one test point and several regions and $O(ln(sqrt(F)) * M)$ in the case of one region and several test points.

Result

The operator `test_region_point` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. The behavior in case of an empty object is set via the operator `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Alternatives

`union1`, `intersection`, `area_center`, `test_region_points`

See also

`select_region_point`, `test_region_points`

Module

Foundation

test_region_points (Regions : : Row, Column : IsInside)

Test if points are contained in regions.

`test_region_points` tests if points (`Row`,`Column`) are contained in the regions `Regions`.

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple or object array of length *1* or *N*. Parameters with length *1* will be repeated internally such that the number of computed tests is always *N*.

In all cases, for each performed test, `IsInside` is set to TRUE (*1*) if the point is contained in the region and to FALSE (*0*) otherwise.

If (`Row`,`Column`) are real-valued, rounding is performed internally since regions are pixel precise.

Attention

Test pixels are not contained in an empty region.

Parameters

▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
Region(s) to be examined.

▷ **Row** (input_control) . . . . . point.y(-array) ⇝ *integer* / real
Row index of the test pixel(s).
Default: 100

▷ **Column** (input_control) . . . . . point.x(-array) ⇝ *integer* / real
Column index of the test pixel(s).
Number of elements: Row == Column
Default: 100

▷ **IsInside** (output_control) . . . . . integer(-array) ⇝ *integer*
Boolean result value.

Result

The operator test_region_points returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of an empty object is set via the operator set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

intersection, area_center, test_region_point

See also

select_region_point, test_region_point

Module

Foundation

test_subset_region (Region1, Region2 : : : IsSubset)

Test whether a region is contained in another region.

test_subset_region tests whether Region1 is a subset of Region2 and returns the result in IsSubset. If more than one region should be tested, Region1 and Region2 must have the same number of elements. In this case, a tuple that contains as many elements as Region1 and Region2 is returned in IsSubset.

For a short description of the iconic objects that are available in HALCON see the introduction of chapter Object.

Parameters

▷ **Region1** (input_object) . . . . . region(-array) ⇝ *object*
Test region.

▷ **Region2** (input_object) . . . . . region(-array) ⇝ *object*
Region for comparison.
Number of elements: Region1 == Region2

▷ **IsSubset** (output_control) . . . . . integer(-array) ⇝ *integer*
Is Region1 contained in Region2?

Result

test_subset_region returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input objects available) is set via the operator set_system(:: 'no_object_result',<Result>:). If the number of objects differs an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Alternatives

difference, area_center

See also

test_equal_region, compare_obj

Module

Foundation

23.7 Transformations

background_seg (Foreground : BackgroundRegions : :)

Determine the connected components of the background of given regions.

background_seg determines connected components of the background of the foreground regions given in Foreground. This operator is normally used after an edge operator in order to determine the regions enclosed by the extracted edges. The connected components are determined using 4-neighborhood.

Parameters

▷ **Foreground** (input_object) ..... region(-array) ⇝ *object*
Input regions.

▷ **BackgroundRegions** (output_object) ..... region-array ⇝ *object*
Connected components of the background.

Example

```
* Simulation of background_seg:
background_seg(Foreground,BackgroundRegions)
  complement(Foreground,Background)
  get_system('neighborhood',Save)
  set_system('neighborhood',4)
  connection(Background,BackgroundRegions)
  set_system('neighborhood',Save)

* Segmentation with edge filter:
read_image(Image,'fabrik')
sobel_dir(Image,Sobel,Dir,'sum_sqrt',3)
threshold(Sobel,Edges,20,255)
skeleton(Edges,Margins)
background_seg(Margins,Regions)
```

Complexity

Let F be the area of the background, H and W be the height and width of the image, and N be the number of resulting regions. Then the runtime complexity is $O(H + sqrt(F) * sqrt(N))$.

Result

background_seg always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, connection, regiongrowing, pouring, class_ndim_norm

Possible Successors

select_shape

Alternatives

complement, connection

See also

threshold, hysteresis_threshold, skeleton, expand_region, set_system, sobel_amp, edges_image, roberts, bandpass_image

Module

Foundation

clip_region (Region : RegionClipped : Row1, Column1, Row2, Column2 :)

Clip a region to a rectangle.

clip_region clips the input regions to the rectangle given by the four control parameters. clip_region is more efficient than calling intersection with a rectangle generated by gen_rectangle1.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Region to be clipped.

▷ **RegionClipped** (output_object) . . . . . region(-array) ⇝ *object*
Clipped regions.

▷ **Row1** (input_control) . . . . . rectangle.origin.y ⇝ *integer*
Row coordinate of the upper left corner of the rectangle.
Default: 0
Suggested values: Row1 ∈ {0, 128, 200, 256}
(lin)

▷ **Column1** (input_control) . . . . . rectangle.origin.x ⇝ *integer*
Column coordinate of the upper left corner of the rectangle.
Default: 0
Suggested values: Column1 ∈ {0, 128, 200, 256}
(lin)

▷ **Row2** (input_control) . . . . . rectangle.corner.y ⇝ *integer*
Row coordinate of the lower right corner of the rectangle.
Default: 256
Suggested values: Row2 ∈ {128, 200, 256, 512}
Value range: 0 ≤ Row2 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Column2** (input_control) . . . . . rectangle.corner.x ⇝ *integer*
Column coordinate of the lower right corner of the rectangle.
Default: 256
Suggested values: Column2 ∈ {128, 200, 256, 512}
Value range: 0 ≤ Column2 ≤ 511 (lin)
Minimum increment: 1
Recommended increment: 10

Result

clip_region returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

Alternatives

intersection, gen_rectangle1, clip_region_rel

Module

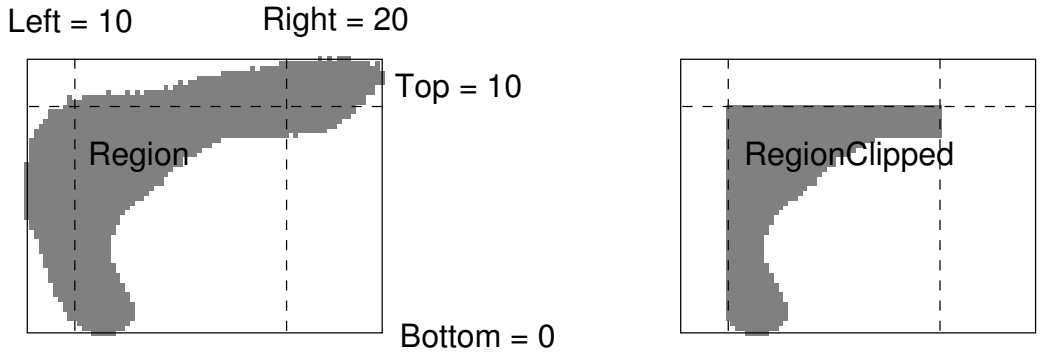
Foundation

clip_region_rel (Region : RegionClipped : Top, Bottom, Left, Right :)

Clip a region relative to its smallest surrounding rectangle.

clip_region_rel reduces the Region by eliminating parts close to the smallest surrounding rectangle of the Region. Specifically this means that the Region is clipped to a rectangle that is smaller than or equal to the smallest surrounding rectangle. The resulting clipped region is returned in RegionClipped.

The rectangle to which the Region is clipped is determined by reducing the smallest surrounding axis-parallel rectangle at the top, bottom, left, and right side by the values given in Top, Bottom, Left, and Right, respectively.



Region with smallest surrounding rectangle (1) and clipped region (2).

These four parameters must contain numbers larger or equal to zero. If all parameters are set to zero, the region remains unchanged.

Parameters

▷ **Region** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be clipped.

▷ **RegionClipped** (output_object) . . . . . region(-array) ⇝ *object*
Clipped regions.

▷ **Top** (input_control) . . . . . integer ⇝ *integer*
Number of rows clipped at the top.
Default: 1
Suggested values: Top ∈ {0, 1, 2, 3, 4, 5, 7, 10, 20, 30, 50}
Value range: 0 ≤ Top (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Bottom** (input_control) . . . . . integer ⇝ *integer*
Number of rows clipped at the bottom.
Default: 1
Suggested values: Bottom ∈ {0, 1, 2, 3, 4, 5, 7, 10, 20, 30, 50}
Value range: 0 ≤ Bottom (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Left** (input_control) . . . . . integer ⇝ *integer*
Number of columns clipped at the left.
Default: 1
Suggested values: Left ∈ {0, 1, 2, 3, 4, 5, 7, 10, 20, 30, 50}
Value range: 0 ≤ Left (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Right** (input_control) . . . . . integer ⇝ *integer*
Number of columns clipped at the right.
Default: 1
Suggested values: Right ∈ {0, 1, 2, 3, 4, 5, 7, 10, 20, 30, 50}
Value range: 0 ≤ Right (lin)
Minimum increment: 1
Recommended increment: 1

Result

clip_region_rel returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

Alternatives

smallest_rectangle1, intersection, gen_rectangle1, clip_region

Module

Foundation

closest_point_transform (Region : Distances,
ClosestPoints : Metric, Foreground, ClosestPointMode, Width,
Height :)

Compute the closest-point transformation of a region.

closest_point_transform computes for every pixel of the input region Region (or its complement, respectively) the distance to the closest pixel outside the region (i.e., to the pixel on the outside border of the region) and returns this information in Distances. In addition to the distance, the corresponding closest pixel is returned in ClosestPoints.

The parameter Foreground determines whether the distances are calculated for all points within the region (Foreground = *'true'*) or for all points outside the region (Foreground = *'false'*). The distance is computed for every pixel of the output images Distances and ClosestPoints. The size of the images is specified by Width and Height. The input region is always clipped to the extent of the output image. If it is important that the distances within the entire region should be computed, the region should be moved (see move_region) so that it has only positive coordinates and the width and height of the output image should be large enough to contain the region. The extent of the input region can be obtained with smallest_rectangle1.

The parameter Metric determines which metric is used for the calculation of the distances. If Metric = *'city-block'*, the distance is calculated from the shortest path from the point to the border of the region, where only horizontal and vertical "movements" are allowed. They are weighted with a weight of 1. If Metric = *'chess-board'*, the distance is calculated from the shortest path to the border, where horizontal, vertical, and diagonal "movements" are allowed. They are weighted with a weight of 1. If Metric = *'octagonal'*, a combination of these approaches is used, which leads to diagonal paths receiving a higher weight. If Metric = *'chamfer-3-4'*, horizontal and vertical movements are weighted with a weight of 3, while diagonal movements are weighted with a weight of 4. To normalize the distances, the resulting distance image is divided by 3. Since this normalization step takes some time, and one usually is interested in the relative distances of the points, the normalization can be suppressed with Metric = *'chamfer-3-4-unnormalized'*. Finally, if Metric = *'euclidean'*, the computed distance is approximately Euclidean.

The parameter ClosestPointMode determines how the closest points are stored. For ClosestPointMode = *'absolute'*, absolute coordinates are stored in ClosestPoints. For ClosestPointMode = *'relative'*, the offset to the coordinate of the respective pixel is stored in ClosestPoints.

Attention

It should be noted that the closest points are usually not unique, i.e., for each pixel in the image Distances, there usually exist several points on the outer border of the region that have the respective distance to that pixel. For example, all points on the skeleton of the region in the chosen metric have the same distance to at least two distinct points on the outer border of the region. closest_point_transform returns one of these points that is determined by the implementation of the algorithm. In particular, invariances with respect to rotation or mirroring of the region should *not* be expected.

Furthermore, it should be noted that for Foreground = *'true'*, point coordinates that lie outside the image defined by Width and Height are returned if the input region Region touches the border of this image, since in this case the outside border of the region lies one pixel outside of the image. If the returned coordinates should be used for a direct access to an image, a suitable border treatment must be implemented.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Region for which the distance to the border is computed.
- ▷ **Distances** (output_object) ..... image ⇝ *object :* int4
  Image containing the distance information.
- ▷ **ClosestPoints** (output_object) ..... image ⇝ *object :* vector_field
  Image containing the coordinates of the closest points.
- ▷ **Metric** (input_control) ..... string ⇝ *string*
  Type of metric to be used for the closest-point transformation.
  Default: 'city-block'
  List of values: Metric ∈ {'city-block', 'chessboard', 'octagonal', 'chamfer-3-4', 'chamfer-3-4-unnormalized', 'euclidean'}
- ▷ **Foreground** (input_control) ..... string ⇝ *string*
  Compute the distance for pixels inside (*'true'*) or outside (*'false'*) the input region.
  Default: 'true'
  List of values: Foreground ∈ {'true', 'false'}
- ▷ **ClosestPointMode** (input_control) ..... string ⇝ *string*
  Mode in which the coordinates of the closest points are returned.
  Default: 'absolute'
  List of values: ClosestPointMode ∈ {'absolute', 'relative'}
- ▷ **Width** (input_control) ..... extent.x ⇝ *integer*
  Width of the output images.
  Default: 640
  Suggested values: Width ∈ {160, 192, 320, 384, 640, 768}
  Value range: 1 ≤ Width
- ▷ **Height** (input_control) ..... extent.y ⇝ *integer*
  Height of the output images.
  Default: 480
  Suggested values: Height ∈ {120, 144, 240, 288, 480, 576}
  Value range: 1 ≤ Height

Complexity

The runtime complexity is $O(\text{Width} * \text{Height})$.

Result

closest_point_transform returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, dyn_threshold, regiongrowing

Possible Successors

threshold, vector_field_to_real

Alternatives

distance_transform

See also

skeleton

References

Y. Ge, C.R. Maurer, Jr., J.M. Fitzpatrick: "Surface-based 3-D image registration using the Iterative Closest Point algorithm with a closest point transform"; in: "Medical Imaging 1996: Image Processing", M.H. Loew, K.M. Hanson, Editors, Proc. SPIE 2710, pages 358–367, 1996.

P. Soille: "Morphological Image Analysis, Principles and Applications"; Springer Verlag Berlin Heidelberg New York, 1999.

G. Borgefors: "Distance Transformations in Arbitrary Dimensions"; Computer Vision, Graphics, and Image Processing, Vol. 27, pages 321–345, 1984.

P.E. Danielsson: "Euclidean Distance Mapping"; Computer Graphics and Image Processing, Vol. 14, pages 227–248, 1980.

Module

Foundation

connection (Region : ConnectedRegions : :)

Compute connected components of a region.

connection determines the connected components of the input regions given in Region. The neighborhood used for this can be set via set_system('neighborhood',<4/8>). The default is 8-neighborhood, which is useful for determining the connected components of the foreground. The maximum number of connected components that is returned by connection can be set via set_system('max_connection',<Num>). The default value of *0* causes all connected components to be returned. The inverse operator of connection is union1.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Input region.
- ▷ **ConnectedRegions** (output_object) ..... region-array ⇝ *object*
  Connected components.

Example

```
read_image(Image,'clip')
dev_set_colored(12)
threshold(Image,Dark,0,150)
count_obj(Dark,NumThresholded)
dev_display (Dark)
connection(Dark,ConnectedRegions)
count_obj(ConnectedRegions,NumConnected)
dev_display (ConnectedRegions)
```

Complexity

Let F be the area of the input region and N be the number of generated connected components. Then the runtime complexity is $O(sqrt(F) * sqrt(N))$.

Result

connection always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given)

can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

auto_threshold, threshold, dyn_threshold, erosion1

Possible Successors

select_shape, select_gray, shape_trans, set_colored, dilation1, count_obj, reduce_domain, add_channels

Alternatives

background_seg

See also

set_system, union1

Module

Foundation

distance_transform (Region : DistanceImage : Metric, Foreground, Width, Height :)

Compute the distance transformation of a region.

distance_transform computes for every point of the input region Region (or its complement, respectively) the distance of the point to the border of the region. The parameter Foreground determines whether the distances are calculated for all points within the region (Foreground = *'true'*) or for all points outside the region (Foreground = *'false'*). The distance is computed for every point of the output image DistanceImage, which has the specified dimensions Width and Height. The input region is always clipped to the extent of the output image. If it is important that the distances within the entire region should be computed, the region should be moved (see move_region) so that it has only positive coordinates and the width and height of the output image should be large enough to contain the region. The extent of the input region can be obtained with smallest_rectangle1.

The parameter Metric determines which metric is used for the calculation of the distances. If Metric = *'city-block'*, the distance is calculated from the shortest path from the point to the border of the region, where only horizontal and vertical "movements" are allowed. They are weighted with a weight of 1. If Metric = *'chessboard'*, the distance is calculated from the shortest path to the border, where horizontal, vertical, and diagonal "movements" are allowed. They are weighted with a weight of 1. If Metric = *'octagonal'*, a combination of these approaches is used, which leads to diagonal paths receiving a higher weight. If Metric = *'chamfer-3-4'*, horizontal and vertical movements are weighted with a weight of 3, while diagonal movements are weighted with a weight of 4. To normalize the distances, the resulting distance image is divided by 3. Since this normalization step takes some time, and one usually is interested in the relative distances of the points, the normalization can be suppressed with Metric = *'chamfer-3-4-unnormalized'*. Finally, if Metric = *'euclidean'*, the computed distance is approximately Euclidean.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Region for which the distance to the border is computed.

▷ **DistanceImage** (output_object) ..... image ⇝ *object :* int4
Image containing the distance information.

▷ **Metric** (input_control) ..... string ⇝ *string*
Type of metric to be used for the distance transformation.
Default: 'city-block'
List of values: Metric ∈ {'city-block', 'chessboard', 'octagonal', 'chamfer-3-4', 'chamfer-3-4-unnormalized', 'euclidean'}

▷ **Foreground** (input_control) ..... string ⇝ *string*
Compute the distance for pixels inside (*'true'*) or outside (*'false'*) the input region.
Default: 'true'
List of values: Foreground ∈ {'true', 'false'}

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the output image.
Default: 640
Suggested values: Width ∈ {160, 192, 320, 384, 640, 768}
Value range: 1 ≤ Width

▷ **Height** (input_control) ..... extent.y ⇝ *integer*
Height of the output image.
Default: 480
Suggested values: Height ∈ {120, 144, 240, 288, 480, 576}
Value range: 1 ≤ Height

Example

```
* Step towards extracting the medial axis of a shape:
gen_rectangle1 (Rectangle1, 0, 0, 200, 400)
gen_rectangle1 (Rectangle2, 200, 0, 400, 200)
union2 (Rectangle1, Rectangle2, Shape)
distance_transform (Shape, DistanceImage, 'chessboard', 'true', 640, 480)
```

Complexity

The runtime complexity is $O(\texttt{Width} * \texttt{Height})$.

Result

distance_transform returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, dyn_threshold, regiongrowing

Possible Successors

threshold

Alternatives

closest_point_transform

See also

skeleton

References

P. Soille: "Morphological Image Analysis, Principles and Applications"; Springer Verlag Berlin Heidelberg New York, 1999.

G. Borgefors: "Distance Transformations in Arbitrary Dimensions"; Computer Vision, Graphics, and Image Processing, Vol. 27, pages 321–345, 1984.

P.E. Danielsson: "Euclidean Distance Mapping"; Computer Graphics and Image Processing, Vol. 14, pages 227–248, 1980.

Module

Foundation

eliminate_runs (Region : RegionClipped : ElimShorter, ElimLonger :)

Eliminate runs of a given length.

eliminate_runs eliminates all runs of the run length encoding of the input regions which are shorter than ElimShorter or longer as ElimLonger.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Region to be clipped.

▷ **RegionClipped** (output_object) ..... region(-array) ⇝ *object*
Clipped regions.

▷ **ElimShorter** (input_control) ..... integer ⇝ *integer*
All runs which are shorter are eliminated.
Default: 3
Suggested values: ElimShorter ∈ {2, 3, 4, 5, 6, 8, 10, 12, 15}
Value range: 1 ≤ ElimShorter ≤ 500 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **ElimLonger** (input_control) ..... integer ⇝ *integer*
All runs which are longer are eliminated.
Default: 1000
Suggested values: ElimLonger ∈ {50, 100, 200, 500, 1000, 2000}
Value range: 1 ≤ ElimLonger ≤ 10000 (lin)
Minimum increment: 1
Recommended increment: 10

Result

eliminate_runs returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

erosion1, dilation1, disp_region

Alternatives

shape_trans

Module

Foundation

expand_region (Regions, ForbiddenArea : RegionExpanded : Iterations, Mode :)

Fill gaps between regions or split overlapping regions.

expand_region closes gaps between the input regions, which resulted from the suppression of small regions in a segmentation operator, for example, (mode *'image'*), or to separate overlapping regions (mode *'region'*). Both uses result from the expansion of regions. The operator works by adding or removing a one pixel wide "strip" to a region.

The expansion takes place only in regions that are designated as not "forbidden" (parameter ForbiddenArea). The number of iterations is determined by the parameter Iterations. By passing *'maximal'*, expand_region iterates until convergence, i.e., until no more changes occur. By passing 0 for this parameter, all non-overlapping regions are returned. The two modes of operation (*'image'* and *'region'*) are different in the following ways:

'image' The input regions are expanded iteratively until they touch another region or the image border. In this case, the image border is defined to be the rectangle ranging from (0,0) to (row_max,col_max). Here, (row_max,col_max) corresponds to the lower right corner of the smallest surrounding rectangle of all input regions (i.e., of all regions that are passed in `Regions` and `ForbiddenArea`). Because `expand_region` processes all regions simultaneously, gaps between regions are distributed evenly to all regions. Overlapping regions are split by distributing the area of overlap evenly to both regions.

'region' No expansion of the input regions is performed. Instead, only overlapping regions are split by distributing the area of overlap evenly to the respective regions. Because the intersection with the original region is computed after the shrinking operation gaps in the output regions may result, i.e., the segmentation is not complete. This can be prevented by calling `expand_region` a second time with the complement of the original regions as "forbidden area."

Parameters

▷ **`Regions`** (input_object) ..... region(-array) ⇝ *object*
Regions for which the gaps are to be closed, or which are to be separated.

▷ **`ForbiddenArea`** (input_object) ..... region ⇝ *object*
Regions in which no expansion takes place.

▷ **`RegionExpanded`** (output_object) ..... region(-array) ⇝ *object*
Expanded or separated regions.

▷ **`Iterations`** (input_control) ..... integer ⇝ *integer* / string
Number of iterations.
Default: 'maximal'
Suggested values: `Iterations` ∈ {'maximal', 0, 1, 2, 3, 5, 7, 10, 15, 20, 30, 50, 70, 100, 200}
Value range: 0 ≤ `Iterations` ≤ 1000 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **`Mode`** (input_control) ..... string ⇝ *string*
Expansion mode.
Default: 'image'
List of values: `Mode` ∈ {'image', 'region'}

Example

```
read_image(Image,'clip')
threshold(Image,Dark,0,150)
connection(Dark,ConnectedRegions)
gen_circle (Circle, 400, 400, 200.5)
expand_region (ConnectedRegions, Circle, RegionExpanded, 'maximal', 'image')
dev_display (RegionExpanded)
```

Result

`expand_region` always returns the value 2 (`H_MSG_TRUE`). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result',<Result>)`, the behavior in case of an empty input region via `set_system('empty_region_result',<Result>)`, and the behavior in case of an empty result region via `set_system('store_empty_region',<'true'/'false'>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`pouring`, `threshold`, `dyn_threshold`, `regiongrowing`

Alternatives

`dilation1`

See also

expand_gray, interjacent, skeleton

Module

Foundation

fill_up (Region : RegionFillUp : :)

Fill up holes in regions.

fill_up fills up holes in regions. The number of regions remains unchanged. The neighborhood type is set via set_system('neighborhood',<4/8>) (default: 8-neighborhood).

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Input regions containing holes.
- ▷ **RegionFillUp** (output_object) ..... region(-array) ⇝ *object*
  Regions without holes.

Result

fill_up returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

Alternatives

fill_up_shape

See also

boundary

Module

Foundation

fill_up_shape (Region : RegionFillUp : Feature, Min, Max :)

Fill up holes in regions having given shape features.

fill_up_shape fills up those holes in the input region Region having given shape features. The parameter Feature determines the shape feature to be used, while Min and Max determine the range the shape feature has to lie in in order for the hole to be filled up.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Input region(s).
- ▷ **RegionFillUp** (output_object) ..... region(-array) ⇝ *object*
  Output region(s) with filled holes.

▷ **Feature** (input_control) ..... string ⇝ *string*
Shape feature used.
Default: 'area'
List of values: Feature ∈ {'area', 'compactness', 'convexity', 'anisometry', 'phi', 'ra', 'rb', 'inner_circle', 'outer_circle'}

▷ **Min** (input_control) ..... number ⇝ *real* / integer
Minimum value for Feature.
Default: 1.0
Suggested values: Min ∈ {0.0, 1.0, 10.0, 50.0, 100.0, 500.0, 1000.0, 10000.0}
Value range: 0.0 ≤ Min

▷ **Max** (input_control) ..... number ⇝ *real* / integer
Maximum value for Feature.
Default: 100.0
Suggested values: Max ∈ {10.0, 50.0, 100.0, 500.0, 1000.0, 10000.0, 100000.0}
Value range: 0.0 ≤ Max

Example

```
read_image(&Image,"monkey");
threshold(Image,&Seg,120.0,255.0);
fill_up_shape(Seg,&Filled,"area",0.0,200.0);
```

Result

fill_up_shape returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection, regiongrowing, pouring

Possible Successors

select_shape, disp_region

Alternatives

fill_up

See also

select_shape, connection, area_center

Module

Foundation

junctions_skeleton (Region : EndPoints, JuncPoints : :)

Find junctions and end points in a skeleton.

junctions_skeleton detects junctions and end points in a skeleton (see skeleton). The junctions in the input region Region are output as a region in JuncPoints, while the end points are output as a region in EndPoints.

To obtain reasonable results with junctions_skeleton the input region Region must not contain lines which are more than one pixel wide. Regions obtained by skeleton meet this condition, while regions obtained by morph_skeleton do not meet this condition in general.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Input skeletons.

▷ **EndPoints** (output_object) ..... region(-array) ⇝ *object*
Extracted end points.
Number of elements: EndPoints == Region

▷ **JuncPoints** (output_object) ..... region(-array) ⇝ *object*
Extracted junctions.
Number of elements: JuncPoints == Region

Example

```
* non-connected branches of a skeleton
skeleton(Region,Skeleton)
junctions_skeleton(Skeleton,EPoints,JPoints)
difference(Skeleton,JPoints,Rows)
set_system('neighborhood',4)
connection(Rows,Parts)
```

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(F)$.

Result

`junctions_skeleton` always returns the value `2` (`H_MSG_TRUE`). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result',<Result>)`, the behavior in case of an empty input region via `set_system('empty_region_result',<Result>)`, and the behavior in case of an empty result region via `set_system('store_empty_region',<'true'/'false'>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`skeleton`

Possible Successors

`area_center`, `connection`, `get_region_points`, `difference`

See also

`pruning`, `split_skeleton_region`

Module

Foundation

merge_regions_line_scan (CurrRegions, PrevRegions : CurrMergedRegions, PrevMergedRegions : ImageHeight, MergeBorder, MaxImagesRegion :)

Merge regions from line scan images.

The operator `merge_regions_line_scan` connects adjacent regions, which were segmented from adjacent images with the height `ImageHeight`. This operator was especially designed to process regions that were extracted from images grabbed by a line scan camera. `CurrRegions` contains the regions from the current image and `PrevRegions` the regions from the previous one.

With the help of the parameter `MergeBorder` two cases can be distinguished: If the top (first) line of the current image touches the bottom (last) line of the previous image, `MergeBorder` must be set to *'top'*, otherwise set `MergeBorder` to *'bottom'*.

If the operator merge_regions_line_scan is used recursively, the parameter MaxImagesRegion determines the maximum number of images which are covered by a merged region. All older region parts are removed.

The operator merge_regions_line_scan returns two region arrays. PrevMergedRegions contains all those regions from the previous input regions PrevRegions, which could not be merged with a current region. CurrMergedRegions collects all current regions together with the merged parts from the previous images. Merged regions will exceed the original image, because the previous regions are moved upward (MergeBorder='top') or downward (MergeBorder='bottom') according to the image height. For this the system parameter *'clip_region'* (see also set_system) will internally be set to *'false'*.

Parameters

- ▷ **CurrRegions** (input_object) ..... region(-array) ⇝ *object*
  Current input regions.
- ▷ **PrevRegions** (input_object) ..... region(-array) ⇝ *object*
  Merged regions from the previous iteration.
- ▷ **CurrMergedRegions** (output_object) ..... region(-array) ⇝ *object*
  Current regions, merged with old ones where applicable.
- ▷ **PrevMergedRegions** (output_object) ..... region(-array) ⇝ *object*
  Regions from the previous iteration which could not be merged with the current ones.
- ▷ **ImageHeight** (input_control) ..... integer ⇝ *integer*
  Height of the line scan images.
  Default: 512
  Suggested values: ImageHeight ∈ {240, 480, 512, 1024}
- ▷ **MergeBorder** (input_control) ..... string ⇝ *string*
  Image line of the current image, which touches the previous image.
  Default: 'top'
  List of values: MergeBorder ∈ {'top', 'bottom'}
- ▷ **MaxImagesRegion** (input_control) ..... integer ⇝ *integer*
  Maximum number of images for a single region.
  Default: 3
  Suggested values: MaxImagesRegion ∈ {1, 2, 3, 4, 5}

Result

The operator merge_regions_line_scan returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

partition_dynamic (Region : Partitioned : Distance, Percent :)

Partition a region horizontally at positions of small vertical extent.

partition_dynamic partitions the input Region horizontally into regions that have an approximate width of Distance. The input region is split at positions where it has a relatively small vertical extent.

The positions where the input region is split are determined by the following approach: First, initial split positions are determined such that they are equally distributed over the horizontal extent of the input region, i.e., such that all the resulting parts would have the same width. For this, the number n of resulting parts is determined by dividing the width of the input region by Distance and rounding the result to the closest integer value. The distance between the initial split positions is now calculated by dividing the width of the input region by n. Note that the distance between these initial split positions is typically not identical to Distance. Then, the final split positions are determined in the neighborhood of the initial split positions such that the input region is split at positions where

it has the least vertical extent within this neighborhood. The maximum deviation of the final split position from the initial split position is Distance*Percent*0.01.

The resulting regions are returned in Partitioned. Note that the input region is only partitioned if its width is larger than 1.5 times Distance.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Region to be partitioned.
- ▷ **Partitioned** (output_object) ..... region-array ⇝ *object*
  Partitioned region.
- ▷ **Distance** (input_control) ..... real ⇝ *real*
  Approximate width of the resulting region parts.
- ▷ **Percent** (input_control) ..... real ⇝ *real*
  Maximum percental shift of the split position.
  Default: 20
  Suggested values: Percent ∈ {0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 70.0, 90.0, 100.0}
  Value range: 0 ≤ Percent ≤ 100

Result

partition_dynamic returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>), the behavior in case of an empty input region via set_system('empty_region_result',<Result>), and the behavior in case of an empty result region via set_system('store_empty_region',<'true'/'false'>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection

Alternatives

partition_rectangle

See also

intersection, smallest_rectangle1, shape_trans, clip_region

Module

Foundation

partition_rectangle (Region : Partitioned : Width, Height :)

Partition a region into rectangles of approximately equal size.

partition_rectangle partitions the input region into rectangles having an extent of Width times Height. The rectangles are returned in Partitioned. The region is always split into rectangles of approximately equal size. If this is not possible with the requested rectangle size, then Width and Height are adapted so that the size of the resulting rectangles is approximately equal. If the region is smaller than the given size, its output remains unchanged. A partition is only done if the size of the region is at least 1.5 times the size of the rectangle given by the parameters.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Region to be partitioned.
- ▷ **Partitioned** (output_object) ..... region(-array) ⇝ *object*
  Partitioned region.

▷ **Width** (input_control) ..... extent.x ⇝ *real*
Width of the individual rectangles.

▷ **Height** (input_control) ..... extent.y ⇝ *real*
Height of the individual rectangles.

Result

`partition_rectangle` returns 2 (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result',<Result>)`, the behavior in case of an empty input region via `set_system('empty_region_result',<Result>)`, and the behavior in case of an empty result region via `set_system('store_empty_region',<'true'/'false'>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`threshold`, `connection`

Alternatives

`partition_dynamic`

See also

`intersection`, `smallest_rectangle1`, `shape_trans`, `clip_region`

Module

Foundation

rank_region (Region : RegionCount : Width, Height, Number :)

Rank operator for regions.

`rank_region` calculates the binary rank operator. A filter mask of size `Height` x `Width`) is used. In the process, for each point in the region the number of points of `Region` lying within the filter mask are counted. If this number is greater or equal to `Number`, the current point is added to the output region. If

$$\texttt{Number} = \frac{\texttt{Height} * \texttt{Width}}{2},$$

is chosen, the median operator is obtained.

Attention

For `Height` and `Width` only odd values > 3 are valid. If invalid parameters are chosen they are converted automatically (without raising an exception) to the next larger odd values.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Region(s) to be transformed.

▷ **RegionCount** (output_object) ..... region(-array) ⇝ *object*
Resulting region(s).

▷ **Width** (input_control) ..... extent.x ⇝ *integer*
Width of the filter mask.
Default: 15
Suggested values: `Width` ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21}
Value range: $3 \leq$ `Width` ≤ 511 (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: Width >= 3 && odd(Width)

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the filter mask.
Default: 15
Suggested values: Height ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21}
Value range: 3 ≤ Height ≤ 511 (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: Height >= 3 && odd(Height)

▷ **Number** (input_control) . . . . . integer ⇝ *integer*
Minimum number of points lying within the filter mask.
Default: 70
Suggested values: Number ∈ {5, 10, 20, 40, 60, 80, 90, 120, 150, 200}
Value range: 1 ≤ Number ≤ 1000 (lin)
Minimum increment: 1
Recommended increment: 10
Restriction: Number > 0

Example

```
read_image(Image,'monkey')
mean_image(Image,Mean,5,5)
dyn_threshold(Image,Mean,Points,25,'light')
rank_region(Points,Textur,15,15,30)
gen_circle(Mask,10,10,3)
opening(Textur,Mask,Seg)
```

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(F * 8)$.

Result

rank_region returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, connection, regiongrowing, pouring, class_ndim_norm

Possible Successors

select_shape, disp_region

Alternatives

closing_rectangle1, expand_region

See also

rank_image, mean_image

Module

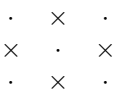
Foundation

remove_noise_region (InputRegion : OutputRegion : Type :)

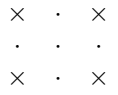
Remove noise from a region.

remove_noise_region removes noise from a region. Depending on Type one of the following structuring elements will be used:

'n_4' A structuring element consisting of the four 4-neighbors of a point is being used:



'n_8' A structuring element consisting of the four 8-neighbors of a point is being used:



'n_48' A structuring element consisting of the four 4-neighbors and the four 8-neighbors of a point is being used:

A dilation with this structuring element is performed and the intersection of the result and the input region is calculated. Thus, all pixels having no according neighbors are removed.

Parameters

▷ **InputRegion** (input_object) . . . . . region(-array) ⇝ *object*
Regions to be modified.

▷ **OutputRegion** (output_object) . . . . . region(-array) ⇝ *object*
Less noisy regions.

▷ **Type** (input_control) . . . . . string ⇝ *string*
Mode of noise removal.
Default: 'n_4'
List of values: `Type` ∈ {'n_4', 'n_8', 'n_48'}

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(\sqrt{F} * 4)$.

Result

`remove_noise_region` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`connection`, `regiongrowing`, `pouring`, `class_ndim_norm`

Possible Successors

`select_shape`

See also

`dilation1`, `intersection`, `gen_region_points`

Module

Foundation

shape_trans (Region : RegionTrans : Type :)

Transform the shape of a region.

`shape_trans` transforms the shape of the input regions depending on the parameter `Type`:

'convex' Convex hull.

'ellipse' Ellipse with the same moments and area as the input region.

'outer_circle' Smallest enclosing circle.

'inner_circle' Largest circle fitting into the region.

'rectangle1' Smallest enclosing rectangle parallel to the coordinate axes.

'rectangle2' Smallest enclosing rectangle.

'inner_rectangle1' Largest axis-parallel rectangle fitting into the region.

'inner_center' The point on the skeleton of the input region having the smallest distance to the center of gravity of the input region.

Attention

If `Type` = *'outer_circle'* is selected it might happen that the resulting circular region does not completely cover the input region. This is because internally the operators `smallest_circle` and `gen_circle` are used to compute the outer circle. As described in the documentation of `smallest_circle`, the calculated radius can be too small by up to $1/\sqrt{2} - 0.5$ pixels. Additionally, the circle that is generated by `gen_circle` is translated by up to 0.5 pixels in both directions, i.e., by up to $1/\sqrt{2}$ pixels. Consequently, when adding up both effects, the original region might protrude beyond the returned circular region by at most 1 pixel.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be transformed.

▷ **RegionTrans** (output_object) ..... region(-array) ⇝ *object*
Transformed regions.

▷ **Type** (input_control) ..... string ⇝ *string*
Type of transformation.
Default: 'convex'
List of values: `Type` ∈ {'convex', 'ellipse', 'outer_circle', 'inner_circle', 'rectangle1', 'rectangle2', 'inner_rectangle1', 'inner_center'}

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(F)$.

Result

`shape_trans` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`connection`, `regiongrowing`

Possible Successors

`disp_region`, `regiongrowing_mean`, `area_center`

See also

`convexity`, `elliptic_axis`, `area_center`, `smallest_rectangle1`, `smallest_rectangle2`, `inner_rectangle1`, `set_shape`, `select_shape`, `inner_circle`

Module

Foundation

skeleton (Region : Skeleton : :)

Compute the skeleton of a region.

skeleton computes the skeleton, i.e., the medial axis of the input regions. The skeleton is constructed in a way that each point on it can be seen as the center point of a circle with the largest radius possible while still being completely contained in the region.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Region to be thinned.

▷ **Skeleton** (output_object) ..... region(-array) ⇝ *object*
Resulting skeleton.
Number of elements: Skeleton == Region

Complexity

Let F be the area of the enclosing rectangle of the input region. Then the runtime complexity is $O(F)$ (per region).

Result

skeleton returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>) and the behavior in case of an empty input region via set_system('empty_region_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

sobel_amp, edges_image, bandpass_image, threshold, hysteresis_threshold

Possible Successors

junctions_skeleton, pruning

Alternatives

morph_skeleton, thinning

See also

gray_skeleton, sobel_amp, edges_image, roberts, bandpass_image, threshold

References

Eckardt, U. "Verdünnung mit Perfekten Punkten", Proceedings 10. DAGM-Symposium, IFB 180, Zurich, 1988

Module

Foundation

sort_region (Regions : SortedRegions : SortMode, Order, RowOrCol :)

Sorting of regions with respect to their relative position.

The operator sort_region sorts the regions with respect to their relative position. All sorting methods with the exception of *'character'* use one point of the region. With the help of the parameter RowOrCol = *'row'* these points will be sorted according to their row and then according to their column. By using *'column'*, the column value will be used first. The following values are available for the parameter SortMode:

'character' The regions are treated like characters, which can be read horizontally or vertically. They are sorted according to the reading direction given by RowOrCol with the following specifications:

- *'row'*: Regarded as rows, the reading direction is firstly from left to right and then from top to bottom.
- *'column'*: Regarded as columns, the reading direction is firstly from top to bottom and then from left to right.

For SortMode *'character'*, a second numeric value can be passed to SortMode in addition to *'character'*. This value specifies the maximum percentage of overlap. This overlap depends on the parameter RowOrCol and is determined as follows:

- RowOrCol = *'row'*: overlap of row coordinates between two regions
- RowOrCol = *'column'*: overlap of column coordinates between two regions,

whereas the percentage of overlap is calculated using the smaller one of the overlapping regions (with respect to the specified axis). Regions that do not overlap in their coordinates or have an overlap of less than the set percentage are seen as being in different rows (or columns, respectively) and sorted accordingly. In case the overlap is larger, the regions are seen as being in the same row (or column) instead and therefore sorted within the row (or column). For example when this additional parameter is set to 0, all regions that overlap in their row (or column) coordinates are sorted within the same row (or column). The default value of this parameter is 15 which means that adjacent rows (or columns) can have an overlap of maximum 15% in order to be considered as different rows (or columns). Note that for Order = *'false'*, the characters are sorted in a reversed order to the one described above.

'first_point' The point with the lowest column value in the first row of the region.

'last_point' The point with the highest column value in the last row of the region.

'upper_left' Upper left corner of the surrounding rectangle.

'upper_right' Upper right corner of the surrounding rectangle.

'lower_left' Lower left corner of the surrounding rectangle.

'lower_right' Lower right corner of the surrounding rectangle.

The parameter Order determines whether the sorting order is increasing or decreasing: using *'true'* the order will be increasing, using *'false'* the order will be decreasing.

Parameters

▷ **Regions** (input_object) ..... region-array ⇝ *object*
Regions to be sorted.

▷ **SortedRegions** (output_object) ..... region-array ⇝ *object*
Sorted regions.

▷ **SortMode** (input_control) ..... tuple(-array) ⇝ *string* / integer / real
Kind of sorting.
Default: 'first_point'
List of values: SortMode ∈ {'character', 'first_point', 'last_point', 'upper_left', 'lower_left', 'upper_right', 'lower_right'}

▷ **Order** (input_control) ..... string ⇝ *string*
Increasing or decreasing sorting order.
Default: 'true'
List of values: Order ∈ {'true', 'false'}

▷ **RowOrCol** (input_control) ..... string ⇝ *string*
Sorting first with respect to row, then to column.
Default: 'row'
List of values: RowOrCol ∈ {'row', 'column'}

Result

If the parameters are correct, the operator sort_region returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

do_ocr_multi_class_mlp, do_ocr_single_class_mlp

Module

Foundation

split_skeleton_lines (SkeletonRegion : : MaxDistance : BeginRow, BeginCol, EndRow, EndCol)

Split lines represented by one pixel wide, non-branching lines.

split_skeleton_lines splits lines represented by one pixel wide, non-branching regions into shorter lines based on their curvature. A line is split if the maximum distance of a point on the line to the line segment connecting its end points is larger than MaxDistance (split & merge algorithm). The start and end points of the approximating line segments are returned in BeginRow, BeginCol, EndRow, and EndCol.

Attention

The input regions must represent non-branching lines, that is single branches of the skeleton.

Parameters

▷ **SkeletonRegion** (input_object) ..... region-array ⇝ *object*
Input lines (represented by 1 pixel wide, non-branching regions).

▷ **MaxDistance** (input_control) ..... integer ⇝ *integer*
Maximum distance of the line points to the line segment connecting both end points.
Default: 3
Suggested values: MaxDistance ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Value range: 1 ≤ MaxDistance ≤ 500 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **BeginRow** (output_control) ..... line.begin.y-array ⇝ *integer*
Row coordinates of the start points of the output lines.

▷ **BeginCol** (output_control) ..... line.begin.x-array ⇝ *integer*
Column coordinates of the start points of the output lines.

▷ **EndRow** (output_control) ..... line.end.y-array ⇝ *integer*
Row coordinates of the end points of the output lines.

▷ **EndCol** (output_control) ..... line.end.x-array ⇝ *integer*
Column coordinates of the end points of the output lines.

Example

```
read_image(Image,'fabrik')
edges_image (Image, ImaAmp, ImaDir, 'lanser2', 0.5, 'nms', 8, 16)
threshold (ImaAmp, RawEdges, 8, 255)
skeleton (RawEdges, Skeleton)
junctions_skeleton (Skeleton, EndPoints, JuncPoints)
difference (Skeleton, JuncPoints, SkelWithoutJunc)
connection (SkelWithoutJunc, SingleBranches)
select_shape (SingleBranches, SelectedBranches, 'area', 'and', 16, 99999)
split_skeleton_lines (SelectedBranches, 3, BeginRow, BeginCol, EndRow, \
                      EndCol)
```

Result

split_skeleton_lines always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>), the behavior in case of an empty input region via set_system('empty_region_result',<Result>), and the behavior in case of an empty result region via set_system('store_empty_region',<'true'/'false'>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

connection, select_shape, skeleton, junctions_skeleton, difference

Possible Successors

select_lines, partition_lines, disp_line

See also

split_skeleton_region, detect_edge_segments

Module

Foundation

split_skeleton_region (
SkeletonRegion : RegionLines : MaxDistance :)

Split lines represented by one pixel wide, non-branching regions.

split_skeleton_region splits lines represented by one pixel wide, non-branching regions into shorter lines based on their curvature. A line is split if the maximum distance of a point on the line to the line segment connecting its end points is larger than MaxDistance (split & merge algorithm). However, not the approximating lines are returned, but rather the original lines split into several output regions.

Attention

The input regions must represent non-branching lines, that is single branches of the skeleton.

Parameters

▷ **SkeletonRegion** (input_object) ..... region(-array) ⇝ *object*
Input lines (represented by 1 pixel wide, non-branching regions).

▷ **RegionLines** (output_object) ..... region-array ⇝ *object*
Split lines.

▷ **MaxDistance** (input_control) ..... integer ⇝ *integer*
Maximum distance of the line points to the line segment connecting both end points.
Default: 3
Suggested values: MaxDistance ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Value range: 1 ≤ MaxDistance ≤ 500 (lin)
Minimum increment: 1
Recommended increment: 1

Example

```
read_image(Image,'fabrik')
edges_image (Image, ImaAmp, ImaDir, 'lanser2', 0.5, 'nms', 8, 16)
threshold (ImaAmp, RawEdges, 8, 255)
skeleton (RawEdges, Skeleton)
junctions_skeleton (Skeleton, EndPoints, JuncPoints)
difference (Skeleton, JuncPoints, SkelWithoutJunc)
connection (SkelWithoutJunc, SingleBranches)
select_shape (SingleBranches, SelectedBranches, 'area', 'and', 16, 99999)
split_skeleton_region (SelectedBranches, Lines, 3)
```

Result

split_skeleton_region always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>), the behavior in case of an empty input region via set_system('empty_region_result',<Result>), and the behavior in case of an empty result region via set_system('store_empty_region',<'true'/'false'>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.

Possible Predecessors

connection, select_shape, skeleton, junctions_skeleton, difference

Possible Successors

count_obj, select_shape, select_obj, area_center, elliptic_axis, smallest_rectangle2, get_region_polygon, get_region_contour

See also

split_skeleton_lines, get_region_polygon, gen_polygons_xld

Module

Foundation

Chapter 24

Segmentation

24.1 Classification

add_samples_image_class_gmm (Image, ClassRegions : : GMMHandle, Randomize :)

Add training samples from an image to the training data of a Gaussian Mixture Model.

add_samples_image_class_gmm adds training samples from the Image to the Gaussian Mixture Model (GMM) given by GMMHandle. add_samples_image_class_gmm is used to store the training samples before a classifier to be used for the pixel classification of multichannel images with classify_image_class_gmm is trained. add_samples_image_class_gmm works analogously to add_sample_class_gmm. The Image must have a number of channels equal to NumDim, as specified with create_class_gmm. The training regions for the NumClasses pixel classes are passed in ClassRegions. Hence, ClassRegions must be a tuple containing NumClasses regions. The order of the regions in ClassRegions determines the class of the pixels. If there are no samples for a particular class in Image an empty region must be passed at the position of the class in ClassRegions. With this mechanism it is possible to use multiple images to add training samples for all relevant classes to the GMM by calling add_samples_image_class_gmm multiple times with the different images and suitably chosen regions. The regions in ClassRegions should contain representative training samples for the respective classes. Hence, they need not cover the entire image. The regions in ClassRegions should not overlap each other, because this would lead to the fact that in the training data the samples from the overlapping areas would be assigned to multiple classes, which may lead to a lower classification performance. Image data of integer type can be particularly badly suited for modeling with a GMM. Randomize can be used to overcome this problem, as explained in add_sample_class_gmm.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / cyclic / direction / int1 / int2 / uint2 / int4 / real
Training image.

▷ **ClassRegions** (input_object) ..... region-array ⇝ *object*
Regions of the classes to be trained.

▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
GMM handle.

▷ **Randomize** (input_control) ..... real ⇝ *real*
Standard deviation of the Gaussian noise added to the training data.
Default: 0.0
Suggested values: Randomize ∈ {0.0, 1.5, 2.0}
Restriction: Randomize >= 0.0

Result

If the parameters are valid, the operator add_samples_image_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- GMMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_gmm

Possible Successors

train_class_gmm, write_samples_class_gmm

Alternatives

read_samples_class_gmm

See also

classify_image_class_gmm, add_sample_class_gmm, clear_samples_class_gmm, get_sample_num_class_gmm, get_sample_class_gmm

Module

Foundation

add_samples_image_class_knn (Image, ClassRegions : : KNNHandle :)

Add training samples from an image to the training data of a k-Nearest-Neighbor classifier.

add_samples_image_class_knn adds training samples from the Image to the k-Nearest-Neighbor (k-NN) given by KNNHandle. add_samples_image_class_knn is used to store the training samples before a classifier is used for the pixel classification of multichannel images with classify_image_class_knn. add_samples_image_class_knn works analogously to add_sample_class_knn. The Image must have a number of channels equal to NumDim, as specified with create_class_knn. ClassRegions must be a tuple containing of at least 2 regions. The order of the regions in ClassRegions determines the class of the pixels. If there are no samples for a particular class in Image an empty region must be passed at the position of the class in ClassRegions. With this mechanism it is possible to use multiple images to add training samples for all relevant classes to the k-NN classifier by calling add_samples_image_class_knn multiple times with different images and suitably chosen regions. The regions in ClassRegions should contain representative training samples for the respective classes. Hence, they do not need to cover the entire image. The regions in ClassRegions should not overlap each other, as these samples from overlapping areas would be assigned to multiple classes in the training data, which may lead to a lower classification performance.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / cyclic / direction / int1 / int2 / uint2 / int4 / real
Training image.

▷ **ClassRegions** (input_object) ..... region-array ⇝ *object*
Regions of the classes to be trained.

▷ **KNNHandle** (input_control) ..... class_knn ⇝ *handle*
Handle of the k-NN classifier.

Result

If the parameters are valid, the operator add_samples_image_class_knn returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- KNNHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_knn

Possible Successors

train_class_knn

Alternatives

add_sample_class_knn

See also

classify_image_class_knn, add_sample_class_knn, add_samples_image_class_svm

Module

Foundation

add_samples_image_class_mlp (Image, ClassRegions : : MLPHandle :)

Add training samples from an image to the training data of a multilayer perceptron.

add_samples_image_class_mlp adds training samples from the image Image to the multilayer perceptron (MLP) given by MLPHandle. add_samples_image_class_mlp is used to store the training samples before a classifier to be used for the pixel classification of multichannel images with classify_image_class_mlp is trained. add_samples_image_class_mlp works analogously to add_sample_class_mlp. Because here the MLP is always used for classification, OutputFunction = *'softmax'* must be specified when the MLP is created with create_class_mlp. The image Image must have a number of channels equal to NumInput, as specified with create_class_mlp. The training regions for the NumOutput pixel classes are passed in ClassRegions. Hence, ClassRegions must be a tuple containing NumOutput regions. The order of the regions in ClassRegions determines the class of the pixels. If there are no samples for a particular class in Image an empty region must be passed at the position of the class in ClassRegions. With this mechanism it is possible to use multiple images to add training samples for all relevant classes to the MLP by calling add_samples_image_class_mlp multiple times with the different images and suitably chosen regions. The regions in ClassRegions should contain representative training samples for the respective classes. Hence, they need not cover the entire image. The regions in ClassRegions should not overlap each other, because this would lead to the fact that in the training data the samples from the overlapping areas would be assigned to multiple classes, which may lead to slower convergence of the training and a lower classification performance.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / cyclic / direction / int1 / int2 / uint2 / int4 / real
Training image.

▷ **ClassRegions** (input_object) ..... region-array ⇝ *object*
Regions of the classes to be trained.

▷ **MLPHandle** (input_control) ..... class_mlp ⇝ *handle*
MLP handle.

Result

If the parameters are valid, the operator add_samples_image_class_mlp returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MLPHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_class_mlp

Possible Successors

train_class_mlp, write_samples_class_mlp

Alternatives

read_samples_class_mlp

See also

classify_image_class_mlp, add_sample_class_mlp, clear_samples_class_mlp, get_sample_num_class_mlp, get_sample_class_mlp, add_samples_image_class_svm

Module

Foundation

add_samples_image_class_svm (Image, ClassRegions : : SVMHandle :)

Add training samples from an image to the training data of a support vector machine.

add_samples_image_class_svm adds training samples from the image Image to the support vector machine (SVM) given by SVMHandle. add_samples_image_class_svm is used to store the training samples before training a classifier for the pixel classification of multichannel images with classify_image_class_svm. add_samples_image_class_svm works analogously to add_sample_class_svm.

The image Image must have a number of channels equal to NumFeatures, as specified with create_class_svm. The training regions for the NumClasses pixel classes are passed in ClassRegions. Hence, ClassRegions must be a tuple containing NumClasses regions. The order of the regions in ClassRegions determines the class of the pixels. If there are no samples for a particular class in Image, an empty region must be passed at the position of the class in ClassRegions. With this mechanism it is possible to use multiple images to add training samples for all relevant classes to the SVM by calling add_samples_image_class_svm multiple times with the different images and suitably chosen regions.

The regions in ClassRegions should contain representative training samples for the respective classes. Hence, they need not cover the entire image. The regions in ClassRegions should not overlap each other, because this would lead to the fact that in the training data the samples from the overlapping areas would be assigned to multiple classes, which may lead to slower convergence of the training and a lower classification performance.

A further application of this operator is the automatic novelty detection, where, e.g., anomalies in color or texture can be detected. For this mode a training set that defines a sample region (e.g., skin regions for skin detection or samples of the correct texture) is passed to the SVMHandle, which is created in the Mode *'novelty-detection'*. After training, regions that differ from the trained sample regions are detected (e.g., the rejection class for skin or errors in texture).

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / cyclic / direction / int1 / int2 / uint2 / int4 / real
Training image.

▷ **ClassRegions** (input_object) ..... region-array ⇝ *object*
Regions of the classes to be trained.

▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
SVM handle.

Result

If the parameters are valid `add_samples_image_class_svm` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SVMHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_class_svm`

Possible Successors

`train_class_svm`, `write_samples_class_svm`

Alternatives

`read_samples_class_svm`

See also

`classify_image_class_svm`, `add_sample_class_svm`, `clear_samples_class_svm`, `get_sample_num_class_svm`, `get_sample_class_svm`, `add_samples_image_class_mlp`

Module

Foundation

class_2dim_sup (ImageCol, ImageRow, FeatureSpace : RegionClass2Dim : :)

Segment an image using two-dimensional pixel classification.

`class_2dim_sup` classifies the points in two-channel images using a two-dimensional feature space. For each point, two gray values (one from each image) are used as features. The feature space is represented by the input region. The classification is done as follows:

A point from the input region of an image is accepted if the point (g_r, g_c), which is determined by the respective gray values, is contained in the region `FeatureSpace`. g_r is here a gray value from the image `ImageRow`, while g_c is the corresponding gray value from `ImageCol`.

Let P be a point with the coordinates $P = (R, C)$, g_r be the gray value at position (R, C) in the image `ImageRow`, and g_c be the gray value at position (R, C) in the image `ImageCol`. Then the point P is aggregated into the output region if

$$(g_r, g_c) \in \texttt{FeatureSpace}$$

g_r is interpreted as row coordinate and g_c as column coordinate.

For the generation of `FeatureSpace`, see `histo_2dim`. The feature space can be modified by applying region transformation operators, such as `rank_region`, `dilation1`, `shape_trans`, `elliptic_axis`, etc., before calling `class_2dim_sup`.

The parameters `ImageCol` and `ImageRow` must contain an equal number of images with the same size. The image points are taken from the intersection of the domains of both images (see `reduce_domain`).

Parameters

- ▷ **ImageCol** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1
  Input image (first channel).
- ▷ **ImageRow** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1
  Input image (second channel).
- ▷ **FeatureSpace** (input_object) . . . . . region(-array) ⇝ *object*
  Region defining the feature space.
- ▷ **RegionClass2Dim** (output_object) . . . . . region(-array) ⇝ *object*
  Classified regions.

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  if (argc != 2)
  {
    cout << "Usage : " << argv[0] << " 'image' " << endl;
    return (-1);
  }

  HRegion   feats, cd2reg;
  HImage    image (argv[1]),
            text1, text2,
            mean1, mean2,
            histo;

  HWindow   win;
  Hlong      nc;

  if ((nc = image.CountChannels ()) != 3)
  {
    cout << argv[1] << " is not a rgb-image " << endl;
    return (-2);
  }

  image.Display (win);

  win.SetColor ("green");
  cout << "Draw the region of interrest " << endl;

  HRegion  region = win.DrawRegion ();

  text1 = image.TextureLaws ("el", 2, 5);
  mean1 = text1.MeanImage (21, 21);
  text2 = mean1.TextureLaws ("es", 2, 5);
  mean2 = text2.MeanImage (21, 21);

  histo = region.Histo2dim (mean1, mean2);
  feats = histo.Threshold (1.0, 1000000.0);

  win.SetDraw ("fill");
  win.SetColor ("red");
```

```
  feats.Display (win);

  cout << "Charakteristics area in red" << endl;

  cd2reg = mean1.Class2dimSup (mean2, feats);

  win.SetColor ("blue");
  cd2reg.Display (win);

  cout << "Result of classification in blue " << endl;
  win.Click ();
  return (0);
}
```

Complexity

Let A be the area of the input region. Then the runtime complexity is $O(256^2 + A)$.

Result

`class_2dim_sup` returns `2` (`H_MSG_TRUE`). If all parameters are correct, the behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`histo_2dim`, `threshold`, `draw_region`, `dilation1`, `opening`, `shape_trans`

Possible Successors

`connection`, `select_shape`, `select_gray`

Alternatives

`class_ndim_norm`, `threshold`

See also

`histo_2dim`

Module

Foundation

class_2dim_unsup (Image1, Image2 : Classes : Threshold, NumClasses :)

Segment two images by clustering.

`class_2dim_unsup` performs a classification with two single-channel images. First, a two-dimensional histogram of the two images is computed (`histo_2dim`). In this histogram, the first maximum is extracted; it serves as the first cluster center. The histogram is computed with the intersection of the domains of both images (see `reduce_domain`). After this, all pixels in the images that are at most `Threshold` pixels from the cluster center in the maximum norm, are determined. These pixels form one output region. Next, the pixels thus classified are deleted from the histogram so that they are not taken into account for the next class. In this modified histogram, again the maximum is extracted; it again serves as a cluster center. The above steps are repeated `NumClasses` times; thus, `NumClasses` output regions result. Only pixels defined in both images are returned.

Attention

Both input images must have the same size.

Parameters

▷ **Image1** (input_object) ..... singlechannelimage ⇝ *object :* byte
First input image.

▷ **Image2** (input_object) ..... singlechannelimage ⇝ *object :* byte
Second input image.

▷ **Classes** (output_object) ..... region-array ⇝ *object*
Classification result.

▷ **Threshold** (input_control) ..... integer ⇝ *integer*
Threshold (maximum distance to the cluster's center).
Default: 15
Suggested values: `Threshold` ∈ {0, 2, 5, 8, 12, 17, 20, 30, 50, 70}

▷ **NumClasses** (input_control) ..... integer ⇝ *integer*
Number of classes (cluster centers).
Default: 5
Suggested values: `NumClasses` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20, 30, 40, 50}

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  if (argc != 2)
  {
    cout << "Usage : " << argv[0] << " 'image' " << endl;
    return (-1);
  }

  HImage   colimg (argv[1]),
           green, blue;

  HWindow  w;
  Hlong     nc;

  if ((nc = colimg.CountChannels ()) != 3)
  {
    cout << argv[1] << " is not a rgb-image " << endl;
    return (-2);
  }

  colimg.Display (w);

  HImage        red = colimg.Decompose3 (&green, &blue);
  HRegionArray  seg = red.Class2dimUnsup (green, 15, 5);

  w.SetDraw ("margin");
  w.SetColored (12);
  seg.Display (w);
  w.Click ();

  return (0);
}
```

Result

`class_2dim_unsup` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect to

the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

decompose2, decompose3, median_image, anisotropic_diffusion, reduce_domain

Possible Successors

select_shape, select_gray, connection

Alternatives

threshold, histo_2dim, class_2dim_sup, class_ndim_norm

Module

Foundation

class_ndim_norm (MultiChannelImage : Regions : Metric, SingleMultiple, Radius, Center :)

Classify pixels using hyper-spheres or hyper-cubes.

class_ndim_norm classifies the pixels of the multi-channel image given in MultiChannelImage. The result is returned in Regions as one region per classification object. The metric used (*'euclid'* or *'maximum'*) is determined by Metric. This parameter must be set to the same value used in learn_ndim_norm. The parameter SingleMultiple determines whether one region (*'single'*) or multiples regions (*'multiple'*) are generated for each cluster. Radius determines the radii or half edge length of the clusters, respectively. Center determines their centers.

Parameters

▷ **MultiChannelImage** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Multi channel input image.

▷ **Regions** (output_object) ..... region-array ⇝ *object*
Classification result.

▷ **Metric** (input_control) ..... string ⇝ *string*
Metric to be used.
Default: 'euclid'
List of values: Metric ∈ {'euclid', 'maximum'}

▷ **SingleMultiple** (input_control) ..... string ⇝ *string*
Return one region or one region for each cluster.
Default: 'single'
List of values: SingleMultiple ∈ {'single', 'multiple'}

▷ **Radius** (input_control) ..... number(-array) ⇝ *real* / integer
Cluster radii or half edge lengths (returned by learn_ndim_norm).

▷ **Center** (input_control) ..... number(-array) ⇝ *real* / integer
Coordinates of the cluster centers (returned by learn_ndim_norm).

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;
```

```
int main ()
{
  HImage   image ("meer"),
           t1, t2, t3,
           m1, m2, m3, m;

  HWindow  w;

  w.SetColor ("green");
  image.Display (w);

  cout << "Draw your region of interest " << endl;

  HRegion testreg = w.DrawRegion ();

  t1 = image.TextureLaws ("el", 2, 5);     m1 = t1.MeanImage (21, 21);
  t2 = image.TextureLaws ("es", 2, 5);     m2 = t2.MeanImage (21, 21);
  t3 = image.TextureLaws ("le", 2, 5);     m3 = t3.MeanImage (21, 21);

  m  = m1.Compose3 (m2, m3);

  Tuple Metric = "euclid";
  Tuple Radius = 20.0;
  Tuple MinNum = 5;
  Tuple NbrCha = 3;

  HRegion empty;
  Tuple cen, t;

  Radius = testreg.LearnNdimNorm (empty, m, Metric, Radius,
                                  MinNum, NbrCha, &cen, &t);
  Tuple RegMod = "multiple";

  HRegionArray reg = m.ClassNdimNorm (Metric, RegMod, Radius, cen, NbrCha);

  w.SetColored (12);
  reg.Display (w);
  cout << "Result of classification" << endl;
  return (0);
}
```

Complexity

Let N be the number of clusters and A be the area of the input region. Then the runtime complexity is $O(N, A)$.

Result

`class_ndim_norm` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`learn_ndim_norm`, `compose2`, `compose3`, `compose4`, `compose5`, `compose6`, `compose7`

Possible Successors

`connection`, `select_shape`, `reduce_domain`, `select_gray`

Alternatives

class_2dim_sup, class_2dim_unsup

Module

Foundation

classify_image_class_gmm (Image : ClassRegions : GMMHandle, RejectionThreshold :)

Classify an image with a Gaussian Mixture Model.

classify_image_class_gmm performs a pixel classification with the Gaussian Mixture Model (GMM) GMMHandle on the multichannel image Image. Before calling classify_image_class_gmm the GMM must be trained with train_class_gmm. Image must have NumDim channels, as specified with create_class_gmm. On output, ClassRegions contains NumClasses regions as the result of the classification. Note that the order of the regions that are returned in ClassRegions corresponds to the order of the classes as defined by the training regions in add_samples_image_class_gmm. The parameter RejectionThreshold can be used to reject pixels that have an uncertain classification. RejectionThreshold represents a threshold on the K-sigma probability measure returned by the classification (see classify_class_gmm and evaluate_class_gmm). All pixels having a probability below RejectionThreshold are not assigned to any class.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / cyclic / direction / int1 / int2 / uint2 / int4 / real
Input image.

▷ **ClassRegions** (output_object) ..... region-array ⇝ *object*
Segmented classes.

▷ **GMMHandle** (input_control) ..... class_gmm ⇝ *handle*
GMM handle.

▷ **RejectionThreshold** (input_control) ..... real ⇝ *real*
Threshold for the rejection of the classification.
Default: 0.5
Suggested values: RejectionThreshold ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: RejectionThreshold >= 0.0 && RejectionThreshold <= 1.0

Example

```
read_image (Image, 'ic')
gen_rectangle1 (Board, 80, 320, 110, 350)
gen_rectangle1 (Capacitor, 359, 263, 371, 302)
gen_rectangle1 (Resistor, 200, 252, 290, 256)
gen_rectangle1 (IC, 180, 135, 216, 165)
concat_obj (Board, Capacitor, Classes)
concat_obj (Classes, Resistor, Classes)
concat_obj (Classes, IC, Classes)
create_class_gmm (3, 4, [1,30], 'full', 'none',0, 42, GMMHandle)
add_samples_image_class_gmm (Image, Classes, GMMHandle, 1.5)
get_sample_num_class_gmm (GMMHandle, NumSamples)
train_class_gmm (GMMHandle, 150, 1e-4, 'training', 1e-4, Centers, Iter)
classify_image_class_gmm (Image, ClassRegions, GMMHandle, 0.0001)
```

Result

If the parameters are valid, the operator classify_image_class_gmm returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

train_class_gmm, read_class_gmm

Alternatives

classify_image_class_knn, classify_image_class_mlp, classify_image_class_svm, classify_image_class_lut, class_ndim_norm, class_2dim_sup

See also

add_samples_image_class_gmm, create_class_gmm

Module

Foundation

classify_image_class_knn (Image : ClassRegions, DistanceImage : KNNHandle, RejectionThreshold :)

Classify an image with a k-Nearest-Neighbor classifier.

classify_image_class_knn performs a pixel classification with a k-Nearest-Neighbor classifier (k-NN) KNNHandle on the multichannel image Image. Before calling classify_image_class_knn the k-NN classifier must be trained with train_class_knn. Image must have NumDim channels, as specified with create_class_knn. On output, ClassRegions contains NumClasses regions as the result of the classification. Note that the order of the regions that are returned in ClassRegions corresponds to the order of the classes as defined by the training regions in add_samples_image_class_knn. The parameter RejectionThreshold can be used to reject pixels that have an uncertain classification. RejectionThreshold represents a threshold on the distance to the nearest neighbor returned by the classification. All pixels having a probability below RejectionThreshold are not assigned to any class. DistanceImage contains the distance of each pixel to its nearest neighbor.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / cyclic / direction / int1 / int2 / uint2 / int4 / real
Input image.

▷ **ClassRegions** (output_object) ..... region-array ⇝ *object :* real
Segmented classes.

▷ **DistanceImage** (output_object) ..... image ⇝ *object*
Distance of the pixel's nearest neighbor.

▷ **KNNHandle** (input_control) ..... class_knn ⇝ *handle*
Handle of the k-NN classifier.

▷ **RejectionThreshold** (input_control) ..... real ⇝ *real*
Threshold for the rejection of the classification.
Default: 0.5
Suggested values: RejectionThreshold ∈ {0.0, 0.1, 0.2, 0.3, 5.0, 10.0, 255.0}
Restriction: RejectionThreshold >= 0.0

Example

```
read_image (Image, 'ic')
gen_rectangle1 (Board, 80, 320, 110, 350)
gen_rectangle1 (Capacitor, 359, 263, 371, 302)
gen_rectangle1 (Resistor, 200, 252, 290, 256)
gen_rectangle1 (IC, 180, 135, 216, 165)
concat_obj (Board, Capacitor, Classes)
concat_obj (Classes, Resistor, Classes)
concat_obj (Classes, IC, Classes)
create_class_knn (3, KNNHandle)
add_samples_image_class_knn (Image, Classes, KNNHandle)
```

```
get_sample_num_class_knn (KNNHandle, NumSamples)
train_class_knn (KNNHandle, [], [])
classify_image_class_knn (Image, ClassRegions, DistanceImage, \
                          KNNHandle, 0.5)
dev_display (ClassRegions)
```

Result

If the parameters are valid, the operator `classify_image_class_knn` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`train_class_knn`, `read_class_knn`

Alternatives

`classify_image_class_svm`, `classify_image_class_mlp`, `classify_image_class_gmm`, `classify_image_class_lut`, `class_ndim_norm`, `class_2dim_sup`

See also

`add_samples_image_class_knn`, `create_class_knn`

Module

Foundation

classify_image_class_lut (
Image : ClassRegions : ClassLUTHandle :)

Classify a byte image using a look-up table.

`classify_image_class_lut` performs a pixel classification on a multi-channel byte `Image` using a look-up table (LUT) `ClassLUTHandle`. The operator can replace `classify_image_class_gmm`, `classify_image_class_knn`, `classify_image_class_mlp` and `classify_image_class_svm`. The classification gets a major speed-up, because the estimation of the class in every image point is no longer necessary since every possible response of the GMM, KNN, MLP or SVM, respectively, is stored in the LUT. This LUT classifier must be created with the trained classifier in `create_class_lut_gmm`, `create_class_lut_knn`, `create_class_lut_mlp` or `create_class_lut_svm`, respectively, before `classify_image_class_lut` can be used. For the classification the parameters in `create_class_gmm`, `create_class_knn`, `create_class_mlp` and `create_class_svm`, respectively, are important: The byte `Image` must have the same number of channels as specified by `NumInput`, `NumFeatures` or `NumDim`, respectively. As result of the pixel classification `classify_image_class_lut` passes `NumOutput` or `NumClasses` regions in `ClassRegions`, respectively

Parameters

▷ **Image** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte
Input image.

▷ **ClassRegions** (output_object) . . . . . region-array ⇝ *object*
Segmented classes.

▷ **ClassLUTHandle** (input_control) . . . . . class_lut ⇝ *handle*
Handle of the LUT classifier.

Example

```
read_image (Image, 'patras')
gen_rectangle1 (Sea, 10, 10, 120, 270)
```

```
gen_rectangle2 (Deck, [170,400], [350,375], [-0.56192,-0.75139], \
                [64,104], [26,11])
union1 (Deck, Deck)
gen_rectangle1 (Walls, 355, 623, 420, 702)
gen_rectangle2 (Chimney, 286, 623, -0.56192, 64, 33)
concat_obj (Sea, Deck, Classes)
concat_obj (Classes, Walls, Classes)
concat_obj (Classes, Chimney, Classes)
*
* create MLP classifier and train it with sample classes
create_class_mlp (3, 3, 4, 'softmax', 'principal_components', 3, \
                  42, MLPHandle)
add_samples_image_class_mlp (Image, Classes, MLPHandle)
train_class_mlp (MLPHandle, 200, 1, 0.01, Error, ErrorLog)
*
* create the LUT classifier
create_class_lut_mlp (MLPHandle, [], [], ClassLUTHandle)
*
* classify the image with the LUT
classify_image_class_lut (Image, ClassRegions, ClassLUTHandle)
```

Result

If the parameters are valid, the operator `classify_image_class_lut` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

`create_class_lut_gmm`, `create_class_lut_knn`, `create_class_lut_mlp`, `create_class_lut_svm`

Alternatives

`classify_image_class_gmm`, `classify_image_class_knn`, `classify_image_class_mlp`, `classify_image_class_svm`

See also

`create_class_lut_gmm`, `create_class_lut_knn`, `create_class_lut_mlp`, `create_class_lut_svm`, `create_class_lut_gmm`

Module

Foundation

classify_image_class_mlp (Image : ClassRegions : MLPHandle, RejectionThreshold :)

Classify an image with a multilayer perceptron.

`classify_image_class_mlp` performs a pixel classification with the multilayer perceptron (MLP) `MLPHandle` on the multichannel image `Image`. Before calling `classify_image_class_mlp` the MLP must be trained with `train_class_mlp`. `Image` must have `NumInput` channels, as specified with `create_class_mlp`. On output, `ClassRegions` contains `NumOutput` regions as the result of the classification. Note that the order of the regions that are returned in `ClassRegions` corresponds to the order of the classes as defined by the training regions in `add_samples_image_class_mlp`. The parameter `RejectionThreshold` can be used to reject pixels that have an uncertain classification. `RejectionThreshold` represents a threshold on the probability measure returned by the classification (see `classify_class_mlp` and `evaluate_class_mlp`). All pixels having a probability below

RejectionThreshold are not assigned to any class. Because an MLP typically assigns pixels that lie outside the convex hull of the training data in the feature space to one of the classes with high probability (confidence), it is useful in many cases to explicitly train a rejection class, even if RejectionThreshold is used, by adding samples for the rejection class with add_samples_image_class_mlp and by re-training the MLP with train_class_mlp.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / cyclic / direction / int1 / int2 / uint2 / int4 / real
Input image.

▷ **ClassRegions** (output_object) ..... region-array ⇝ *object*
Segmented classes.

▷ **MLPHandle** (input_control) ..... class_mlp ⇝ *handle*
MLP handle.

▷ **RejectionThreshold** (input_control) ..... real ⇝ *real*
Threshold for the rejection of the classification.
Default: 0.5
Suggested values: RejectionThreshold ∈ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Restriction: RejectionThreshold >= 0.0 && RejectionThreshold <= 1.0

Example

```
read_image (Image, 'ic')
gen_rectangle1 (Board, 80, 320, 110, 350)
gen_rectangle1 (Capacitor, 359, 263, 371, 302)
gen_rectangle1 (Resistor, 200, 252, 290, 256)
gen_rectangle1 (IC, 180, 135, 216, 165)
concat_obj (Board, Capacitor, Classes)
concat_obj (Classes, Resistor, Classes)
concat_obj (Classes, IC, Classes)
create_class_mlp (3, 3, 4, 'softmax', 'principal_components', 3, 42, \
                  MLPHandle)
add_samples_image_class_mlp (Image, Classes, MLPHandle)
get_sample_num_class_mlp (MLPHandle, NumSamples)
train_class_mlp (MLPHandle, 200, 1, 0.01, Error, ErrorLog)
classify_image_class_mlp (Image, ClassRegions, MLPHandle, 0.5)
dev_display (ClassRegions)
```

Result

If the parameters are valid, the operator classify_image_class_mlp returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

train_class_mlp, read_class_mlp

Alternatives

classify_image_class_gmm, classify_image_class_knn, classify_image_class_svm, classify_image_class_lut, class_ndim_norm, class_2dim_sup

See also

add_samples_image_class_mlp, create_class_mlp

Module

Foundation

classify_image_class_svm (Image : ClassRegions : SVMHandle :)

Classify an image with a support vector machine.

classify_image_class_svm performs a pixel classification with the support vector machine (SVM) SVMHandle on the multichannel image Image. Before calling classify_image_class_svm the SVM must be trained with train_class_svm. Image must have NumFeatures channels, as specified with create_class_svm. On output, ClassRegions contains NumClasses regions as the result of the classification. Note that the order of the regions that are returned in ClassRegions corresponds to the order of the classes as defined by the training regions in add_samples_image_class_svm.

To prevent that the SVM assigns pixels that lie outside the convex hull of the training data in the feature space to one of the classes, it is useful in many cases to explicitly train a rejection class by adding samples for the rejection class with add_samples_image_class_svm and by re-training the SVM with train_class_svm.

An alternative for explicitly defining a rejection class is to use an SVM in the mode *'novelty-detection'*. Please refer to the description in create_class_svm and add_samples_image_class_svm.

Parameters

- ▷ **Image** (input_object) ..... (multichannel-)image ⇝ *object :* byte / cyclic / direction / int1 / int2 / uint2 / int4 / real
  Input image.
- ▷ **ClassRegions** (output_object) ..... region-array ⇝ *object*
  Segmented classes.
- ▷ **SVMHandle** (input_control) ..... class_svm ⇝ *handle*
  SVM handle.

Example

```
read_image (Image, 'ic')
gen_rectangle1 (Board, 20, 270, 160, 420)
gen_rectangle1 (Capacitor, 359, 263, 371, 302)
gen_rectangle1 (Resistor, 200, 252, 290, 256)
gen_rectangle1 (IC, 180, 135, 216, 165)
concat_obj (Board, Capacitor, Classes)
concat_obj (Classes, Resistor, Classes)
concat_obj (Classes, IC, Classes)
create_class_svm (3, 'rbf', 0.01, 0.01, 4, 'one-versus-all', \
                  'normalization', 3, SVMHandle)
add_samples_image_class_svm (Image, Classes, SVMHandle)
train_class_svm (SVMHandle, 0.001, 'default')
reduce_class_svm (SVMHandle, 'bottom_up', 2, 0.01, SVMHandleReduced)
classify_image_class_svm (Image, ClassRegions, SVMHandleReduced)
dev_display (ClassRegions)
```

Result

If the parameters are valid the operator classify_image_class_svm returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

train_class_svm, read_class_svm, reduce_class_svm

Alternatives

classify_image_class_gmm, classify_image_class_knn, classify_image_class_mlp, classify_image_class_lut, class_ndim_norm, class_2dim_sup

See also

add_samples_image_class_svm, create_class_svm

Module

Foundation

learn_ndim_norm (Foreground, Background, Image : : Metric, Distance, MinNumberPercent : Radius, Center, Quality)

Construct classes for class_ndim_norm.

learn_ndim_norm generates classification clusters from the region Foreground and the corresponding gray values in the multi-channel image Image, which can be used in class_ndim_norm. Background determines a class of pixels not to be found in class_ndim_norm. This parameter may be empty (empty object).

The parameter Distance determines the maximum distance Radius of the clusters. It describes the minimum distance between two cluster centers. If the parameter Distance is small the (small) hyper-cubes or hyper-spheres can approximate the feature space well. Simultaneously the runtime during classification increases.

The ratio of the number of pixels in a cluster to the total number of pixels (in percent) must be larger than the value of MinNumberPercent, otherwise the cluster is not returned. MinNumberPercent serves to eliminate outliers in the training set. If it is chosen too large many clusters are suppressed.

Two different clustering procedures can be selected: The minimum Euclidean distance algorithm (n-dimensional hyper-spheres) and the maximum algorithm (n-dimensional hyper-cubes) for describing the pixels of the image to classify in the n-dimensional histogram (parameter Metric). The Euclidean metric usually yields the better results, but takes longer to compute. The parameter Quality returns the quality of the clustering. It is a measure of overlap between the rejection class and the classificator classes. Values larger than 0 denote the corresponding ratio of overlap. If no rejection region is given, its value is set to 1. The regions in Background do not influence on the clustering. They are merely used to check the results that can be expected.

From a user's point of view the key difference between learn_ndim_norm and learn_ndim_box is that in the latter case the rejection class affects the classification process itself. Here, a hyper plane is generated that separates Foreground and Background classes, so that no points in feature space are classified incorrectly. As for learn_ndim_norm, however, an overlap between Foreground and Background class is allowed. This has its effect on the return value Quality. The larger the overlap, the smaller this value.

Parameters

▷ **Foreground** (input_object) . . . . . region(-array) ⇝ *object*
Foreground pixels to be trained.

▷ **Background** (input_object) . . . . . region(-array) ⇝ *object*
Background pixels to be trained (rejection class).

▷ **Image** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte
Multi-channel training image.

▷ **Metric** (input_control) . . . . . string ⇝ *string*
Metric to be used.
Default: 'euclid'
List of values: Metric ∈ {'euclid', 'maximum'}

▷ **Distance** (input_control) . . . . . number ⇝ *real* / integer
Maximum cluster radius.
Default: 10.0
Suggested values: Distance ∈ {1.0, 2.0, 3.0, 4.0, 6.0, 8.0, 10.0, 13.0, 17.0, 24.0, 30.0, 40.0}
Value range: 1.0 ≤ Distance (lin)
Minimum increment: 0.01
Recommended increment: 1.0

▷ **MinNumberPercent** (input_control) . . . . . number ⇝ *real* / integer
The ratio of the number of pixels in a cluster to the total number of pixels (in percent) must be larger than

MinNumberPercent (otherwise the cluster is not output).
Default: 0.01
Suggested values: MinNumberPercent ∈ {0.001, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0}
Value range: 0.0 ≤ MinNumberPercent ≤ 100.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.1

▷ **Radius** (output_control) ..... real-array ⇝ *real*
Cluster radii or half edge lengths.

▷ **Center** (output_control) ..... real-array ⇝ *real*
Coordinates of all cluster centers.

▷ **Quality** (output_control) ..... real ⇝ *real*
Overlap of the rejection class with the classified objects (1: no overlap).
Assertion: 0 <= Quality && Quality <= 1

Result

learn_ndim_norm returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images can be determined by setting the values of the flags *'no_object_result'* and *'empty_region_result'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

min_max_gray, sobel_amp, binomial_filter, gauss_filter, reduce_domain, diff_of_gauss

Possible Successors

class_ndim_norm, connection, dilation1, erosion1, opening, closing, rank_region, shape_trans, skeleton

Alternatives

learn_ndim_box, learn_class_box

See also

class_ndim_norm, histo_2dim

References

P. Haberäcker, "Digitale Bildverarbeitung"; Hanser-Studienbücher, München, Wien, 1987

Module

Foundation

24.2 Edges

detect_edge_segments (Image : : SobelSize, MinAmplitude, MaxDistance, MinLength : BeginRow, BeginCol, EndRow, EndCol)

Detect straight edge segments.

detect_edge_segments detects straight edge segments in the gray image Image. The extracted edge segments are returned as line segments with start point (BeginRow,BeginCol) and end point (EndRow,EndCol). Edge detection is based on the Sobel filter, using *'sum_abs'* as parameter and SobelSize as the filter mask size (see sobel_amp). Only pixels with a filter response larger than MinAmplitude are used as candidates for edge points. These thresholded edge points are thinned and split into straight segments. Due to technical reasons, edge points in which several edges meet are lost. Therefore, detect_edge_segments usually does not return closed object contours. The parameter MaxDistance controls the maximum allowed distance of an edge point to its approximating line. For efficiency reasons, the sum of the absolute values of the coordinate differences is used instead of the Euclidean distance. MinLength controls the minimum length of the line segments. Lines shorter than MinLength are not returned.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte
Input image.

▷ **SobelSize** (input_control) ..... integer ⇝ *integer*
Mask size of the Sobel operator.
Default: 5
List of values: SobelSize ∈ {3, 5, 7, 9, 11, 13}

▷ **MinAmplitude** (input_control) ..... integer ⇝ *integer*
Minimum edge strength.
Default: 32
Suggested values: MinAmplitude ∈ {10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 80, 90, 100, 110}
Value range: 1 ≤ MinAmplitude ≤ 255
Minimum increment: 1
Recommended increment: 1

▷ **MaxDistance** (input_control) ..... integer ⇝ *integer*
Maximum distance of the approximating line to its original edge.
Default: 3
Suggested values: MaxDistance ∈ {2, 3, 4, 5, 6, 7, 8}
Value range: 0 ≤ MaxDistance
Minimum increment: 1
Recommended increment: 1

▷ **MinLength** (input_control) ..... integer ⇝ *integer*
Minimum length of to resulting line segments.
Default: 10
Suggested values: MinLength ∈ {3, 5, 7, 9, 11, 13, 16, 20}
Value range: 1 ≤ MinLength
Minimum increment: 1
Recommended increment: 1

▷ **BeginRow** (output_control) ..... line.begin.y-array ⇝ *integer*
Row coordinate of the line segments' start points.

▷ **BeginCol** (output_control) ..... line.begin.x-array ⇝ *integer*
Column coordinate of the line segments' start points.

▷ **EndRow** (output_control) ..... line.end.y-array ⇝ *integer*
Row coordinate of the line segments' end points.

▷ **EndCol** (output_control) ..... line.end.x-array ⇝ *integer*
Column coordinate of the line segments' end points.

Example

```
Htuple  SobelSize,MinAmplitude,MaxDistance,MinLength;
Htuple  RowBegin,ColBegin,RowEnd,ColEnd;

create_tuple(&SobelSize,1);
set_i(SobelSize,5,0);
create_tuple(&MinAmplitude,1);
set_i(MinAmplitude,32,0);
create_tuple(&MaxDistance,1);
set_i(MaxDistance,3,0);
create_tuple(&MinLength,1);
set_i(MinLength,10,0);
T_detect_edge_segments(Image,SobelSize,MinAmplitude,MaxDistance,MinLength,
                       &RowBegin,&ColBegin,&RowEnd,&ColEnd);
```

Result

detect_edge_segments returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the

behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`sigma_image`, `median_image`

Possible Successors

`partition_lines`, `line_position`, `line_orientation`

Alternatives

`sobel_amp`, `threshold`, `skeleton`

Module

Foundation

`hysteresis_threshold` (`Image` : `RegionHysteresis` : `Low`, `High`, `MaxLength` :)

Perform a hysteresis threshold operation on an image.

`hysteresis_threshold` performs a hysteresis threshold operation (introduced by Canny) on an image. All points in the input image `Image` having a gray value larger than or equal to `High` are immediately accepted ("secure" points). Conversely, all points with gray values less than `Low` are immediately rejected. "Potential" points with gray values between both thresholds are accepted if they are connected to "secure" points by a path of "potential" points having a length of at most `MaxLength` points. This means that "secure" points influence their surroundings (hysteresis).

Attention

For images of type byte, uint2, or int4 the lower threshold must be `Low` > 0.

Parameters

▷ **`Image`** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4 / real
Input image.

▷ **`RegionHysteresis`** (output_object) ..... region(-array) ⇝ *object*
Segmented region.

▷ **`Low`** (input_control) ..... number ⇝ *integer* / real
Lower threshold for the gray values.
Default: 30
Suggested values: `Low` ∈ {5, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100}

▷ **`High`** (input_control) ..... number ⇝ *integer* / real
Upper threshold for the gray values.
Default: 60
Suggested values: `High` ∈ {5, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130}
Restriction: High >= Low

▷ **`MaxLength`** (input_control) ..... integer ⇝ *integer*
Maximum length of a path of "potential" points to reach a "secure" point.
Default: 10
Suggested values: `MaxLength` ∈ {1, 2, 3, 5, 7, 10, 12, 14, 17, 20, 25, 30, 35, 40, 50}
Value range: 1 ≤ MaxLength
Minimum increment: 1
Recommended increment: 5

Result

`hysteresis_threshold` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect

to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Alternatives

`dyn_threshold`, `threshold`, `class_2dim_sup`, `fast_threshold`

See also

`edges_image`, `sobel_dir`, `background_seg`

References

J. Canny, "Finding Edges and Lines in Images"; Report, AI-TR-720, M.I.T. Artificial Intelligence Lab., Cambridge, MA, 1983.

Module

Foundation

`nonmax_suppression_amp` (ImgAmp : ImageResult : Mode :)

Suppress non-maximum points on an edge.

`nonmax_suppression_amp` suppresses all points in the regions of the image `ImgAmp` whose gray values are not local (directed) maxima. In contrast to `nonmax_suppression_dir`, a direction image is not needed. Two modes of operation can be selected:

'hvnms' A point is labeled as a local maximum if its gray value is larger than or equal to the gray values within a search space of pm 5 pixels, either horizontally or vertically. Non-maximum points are removed from the region, gray values remain unchanged.

'loc_max' A point is labeled as a local maximum if its gray value is larger than or equal to the gray values of its eight neighbors.

Parameters

▷ **`ImgAmp`** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Amplitude (gradient magnitude) image.

▷ **`ImageResult`** (output_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Image with thinned edge regions.

▷ **`Mode`** (input_control) . . . . . string ⇝ *string*
Select horizontal/vertical or undirected NMS.
Default: 'hvnms'
List of values: `Mode` ∈ {'hvnms', 'loc_max'}

Result

`nonmax_suppression_amp` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

sobel_amp

Possible Successors

threshold, hysteresis_threshold

Alternatives

local_max, nonmax_suppression_dir

See also

skeleton

References

S.Lanser: "Detektion von Stufenkanten mittels rekursiver Filter nach Deriche"; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

J.Canny: "Finding Edges and Lines in Images"; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge, MA; 1983.

Module

Foundation

nonmax_suppression_dir (ImgAmp, ImgDir : ImageResult : Mode :)

Suppress non-maximum points on an edge using a direction image.

nonmax_suppression_dir suppresses all points in the regions of the image ImgAmp whose gray values are not local (directed) maxima. ImgDir is a direction image giving the direction perpendicular to the local maximum (Unit: 2 degrees, i.e., 50 degrees are coded as 25 in the image). Such images are returned, for example, by edges_image. Two modes of operation can be selected:

'nms' Each point in the image is tested whether its gray value is a local maximum perpendicular to its direction. In this mode only the two neighbors closest to the given direction are examined. If one of the two gray values is greater than the gray value of the point to be tested, it is suppressed (i.e., removed from the input region. The corresponding gray value remains unchanged).

'inms' Like *'nms'*. However, the two gray values for the test are obtained by interpolation from four adjacent points.

Parameters

▷ **ImgAmp** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Amplitude (gradient magnitude) image.

▷ **ImgDir** (input_object) ..... singlechannelimage(-array) ⇝ *object :* direction
Direction image.

▷ **ImageResult** (output_object) ..... image(-array) ⇝ *object :* byte / uint2 / real
Image with thinned edge regions.

▷ **Mode** (input_control) ..... string ⇝ *string*
Select non-maximum-suppression or interpolating NMS.
Default: 'nms'
List of values: Mode ∈ {'nms', 'inms'}

Result

nonmax_suppression_dir returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

edges_image, sobel_dir, frei_dir

Possible Successors

threshold, hysteresis_threshold

Alternatives

nonmax_suppression_amp

See also

skeleton

References

S.Lanser: "Detektion von Stufenkanten mittels rekursiver Filter nach Deriche"; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

J.Canny: "Finding Edges and Lines in Images"; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge; 1983.

Module

Foundation

24.3 Maximally Stable Extremal Regions

```
segment_image_mser ( Image : MSERDark, MSERLight : Polarity,
    MinArea, MaxArea, Delta, GenParamName, GenParamValue : )
```

Segment image using Maximally Stable Extremal Regions (MSER).

segment_image_mser segments an image into regions of homogenous gray values using the approach of Maximally Stable Extremal Regions (MSER). The segmentation process determines if a region is homogenous by observing the local region surrounding. Therefore, the operator is particularly suited to robustly segment objects in front of inhomogeneous background or in applications with changing illumination.

Parameters

Polarity The Polarity determines the type of the regions that are extracted.

Value	Meaning
'dark'	Only MSERs that are darker than their surroundings are extracted
'light'	Only MSERs that are lighter than their surroundings are extracted
'both' (default)	Both types of MSERs are extracted

MinArea, MaxArea The values MinArea and MaxArea restrict the size of the returned MSERs.

Note that very small values of MinArea, e.g., values smaller than 5, can increase the runtime significantly, especially for noisy images.

If MaxArea is set to an empty tuple (default), the MSERs are restricted to be true subsets of the connected components of the input domain.

Delta The value of Delta influences the selectivity of the algorithm. Larger values lead to fewer MSERs. Smaller values lead to more MSERs.

Please read the description of the segmentation process below to help understand the effect of this parameter.

The following generic parameters can be used to fine-tune the segmentation of MSERs. The generic parameters can be set with GenParamName and GenParamValue.

'max_variation': The maximum variation of a component's area within the range of ± Delta thresholds. Larger values lead to more MSERs. Smaller values lead to fewer MSERs.

Please read the description of the segmentation process below for a definition of 'variation' and to help understand the effect of this generic parameter.

Suggested values: *0.1*, *0.2*, *0.5*, *1.0*, *2.0*, *5.0*

Default: *0.2*

Restriction: real values lager than or equal to *0.0*.

'min_diversity': The minimum relative difference of the sizes of two overlapping MSERs. Smaller values lead to more overlapping MSERs. Larger values lead to fewer overlapping MSERs.

Please read the description of the segmentation process below for a definition of 'diversity' and to help understand the effect of this generic parameter.

Setting *'min_diversity'* very close to *0.0* may increase the runtime.

Suggested values: *0.1*, *0.5*, *0.8*, *1.0*, *2.0*, *5.0*

Default: *0.8*

Restriction: real values larger than or equal to *0.0*.

'may_touch_border': Controls if regions that touch the border of the input domain are returned (*'true'*) or rejected (*'false'*).

List of values: *'false'*, *'true'*

Default: *'false'* if a full domain is used, *'true'* if the input domain is reduced.

'min_gray', *'max_gray'*: The values *'min_gray'* and *'max_gray'* reduce the input domain dynamically by applying a `threshold` to the input image. All pixels outside the specified gray value range are ignored in the segmentation process. This may reduce the runtime considerably.

Please note, that if `Image` has a full domain and the domain is reduced by the settings of *'min_gray'* or *'max_gray'*, the default behavior of *'may_touch_border'* may lead to more result regions than without restricted gray value range.

Default: *'min_gray'*: 0, *'max_gray'*: *255* for byte images, *65535* for uint2 images

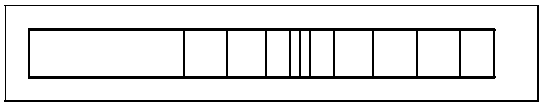
Restriction: integer values larger than or equal to *0*.

Segmentation Process

In a first step, the image is segmented with all threshold values t, from 0 to the maximal present gray value.

To illustrate this, the following example input image with twelve gray values (0...11) is used. On the right, the boundaries of the resulting threshold regions are shown.

Example input image with twelve gray values reaching from 0 (= black) to 11 (= white)



Boundaries of the threshold regions for all thresholds (t = 0...11)

The resulting threshold regions are split into their connected components (4-connected neighborhood) and the area increase of the individual components is monitored over the increasing thresholds. The area of each individual component increases monotonically with each (increasing) threshold. An MSER is a component whose area does not vary significantly within the range of $\pm$ `Delta` thresholds. To be accepted as an MSER, the variation of the component's area within the range of $\pm$ `Delta` thresholds must be a local minimum and it must be lower than *'max_variation'*. Furthermore, the diversity of overlapping MSERs must be greater than *'min_diversity'* (see below).

The variation of a component's area is defined by

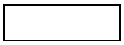
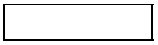
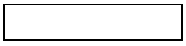
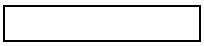
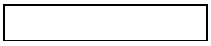
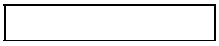
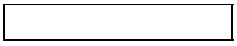
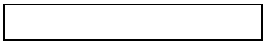
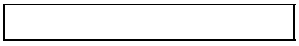
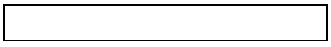
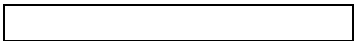
$$Variation_t = \frac{area_{t+\Delta} - area_{t-\Delta}}{area_t}$$

where

$$\begin{aligned} area_{t-\Delta} &= area_0, \qquad t - \Delta < 0 \\ area_{t+\Delta} &= area_{t_{max}}, \qquad t + \Delta > t_{max} \end{aligned}$$

Decreasing the value of *'max_variation'* will reduce the number of accepted regions.

In our example, each threshold region consists of only one component. Therefore, the terms 'threshold region' and 'component' are used synonymously hereinafter. The following table shows the area of the threshold regions and their variations for `Delta` set to 1 and for `Delta` set to 2.

Threshold value t	Threshold region	Area of the threshold region	Variation of the area (for `Delta` = 1)	Variation of the area (for `Delta` = 2)
0		320	**0.28**	**0.53**
1		410	0.41	0.54
2		490	0.27	0.49
3		540	0.13	0.31
4		560	**0.07**	**0.25**
5		580	0.12	0.29
6		630	0.21	0.38
7		710	0.24	0.44
8		800	0.23	**0.41**
9		890	**0.18**	1.67
10		960	1.36	1.46
11		2200	**0.56**	**0.60**

The area of the threshold regions increases with the threshold value t. If `Delta` is set to 1, the local minima of the variation are obtained for the threshold values t = 0, t = 4, t = 9, and t = 11. Therefore, the threshold regions for t = 0, t = 4, t = 9, and t = 11 are MSER candidates. If `Delta` is set to 2, the MSER candidates correspond to the threshold regions for t = 0, t = 4, t = 8, t = 11.

Per default, the connected components of the image domain will not be returned as MSER. This behavior can be altered by explicitly setting `MaxArea` to a value larger than the area of the input domain and *'may_touch_border'* to *'true'*. In our example, the default behavior eliminates the MSER candidates that correspond to the threshold regions for t = 11.

The following figure shows the resulting MSER candidates for `Delta` = 1 and for `Delta` = 2 (with *'max_variation'* set to 1.0), overlaid over the input image.

MSER candidates for Delta = 1	MSER candidates for Delta = 2
	
corresponds to t = 0	corresponds to t = 0
	
corresponds to t = 4	corresponds to t = 4
	
corresponds to t = 9	corresponds to t = 8

The MSER segmentation does not divide the image into disjunct regions, but rather determines all MSERs within the input image. Hence for one image, multiple mutual overlapping regions may be returned as MSER candidates. Since the MSERs are calculated by continuously increasing the threshold t, two regions that overlap each other always consist of a larger region that completely contains the smaller one. The 'diversity' measures, how much the areas of two overlapping MSERs differ. It is calculated as the relative difference of the areas of the two MSER candidates:

$$diversity_i = \frac{area_{i+1} - area_i}{area_i}$$

where $area_i$: is the area of the currently examined MSER candidate and $area_{i+1}$: is the area of the MSER candidate that contains the currently examined MSER candidate.

If the diversity of an MSER candidate is smaller than *'min_diversity'*, it is discarded. Hence the parameter *'min_diversity'* is used to control the amount of overlapping regions. Higher values of *'min_diversity'* reduce the number of overlapping regions and lower values increase it. Note that the largest MSER candidates, i.e., those candidates that are not contained in another MSER candidate are always kept.

	MSER candidate index i	Threshold value t	Area of the threshold region	Diversity of the MSER candidate
Delta=1	0	0	320	0.75
	1	4	560	0.59
	2	9	890	N/A
Delta=2	0	0	320	0.75
	1	4	560	0.43
	2	8	800	N/A

First, the MSER candidate with the smallest diversity is eliminated if its diversity is less than *'min_diversity'*. In our example, the value for *'min_diversity'* is set to 0.5. For delta = 2, the MSER candidate with the diversity of 0.43 is eliminated. Then, the diversity is recalculated for the remaining MSER candidates. In our example, this yields a diversity of 1.5 for the MSER candidate 0. Now, all remaining MSER candidates have a diversity greater than 0.5 and are therefore accepted as MSERs.

The segmentation process calculates MSERs for each connected component of the input domain independently.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image.

▷ **MSERDark** (output_object) . . . . . region-array ⇝ *object*
Segmented dark MSERs.

▷ **MSERLight** (output_object) . . . . . region-array ⇝ *object*
Segmented light MSERs.

▷ **Polarity** (input_control) . . . . . string ⇝ *string*
The polarity of the returned MSERs.
Default: 'both'
List of values: `Polarity` ∈ {'dark', 'light', 'both'}

▷ **MinArea** (input_control) . . . . . number(-array) ⇝ *integer* / real
Minimal size of an MSER.
Default: 10
Suggested values: `MinArea` ∈ {1, 10, 100, 10000}
Value range: 1 ≤ MinArea

▷ **MaxArea** (input_control) . . . . . number(-array) ⇝ *integer* / real
Maximal size of an MSER.
Default: []
Suggested values: `MaxArea` ∈ {1, 10, 100, 10000}
Value range: 1 ≤ MaxArea

▷ **Delta** (input_control) . . . . . number ⇝ *integer* / real
Amount of thresholds for which a region needs to be stable.
Default: 15
Suggested values: `Delta` ∈ {5, 10, 20, 50}
Value range: 1 ≤ `Delta` ≤ 65535

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
List of generic parameter names.
Default: []
List of values: `GenParamName` ∈ {'max_variation', 'min_diversity', 'may_touch_border', 'min_gray', 'max_gray'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *string* / real / integer
List of generic parameter values.
Default: []
Suggested values: `GenParamValue` ∈ {0.5, 0.8, 'true', 'false', 30, 50, 200, 230}

Example

```
read_image (Image,'pellets')
segment_image_mser (Image, MSERDark, MSERLight, 'light',\
                    1000, 10000, 3, [],[])
```

Result

`segment_image_mser` returns 2 (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

`select_shape`, `select_gray`

Alternatives

`auto_threshold`, `binary_threshold`, `char_threshold`, `local_threshold`, `watersheds`, `regiongrowing`

References

J. Matas, O. Chum, M. Urban, and T. Pajdla: "Robust wide baseline stereo from maximally stable extremal regions."; Proc. of British Machine Vision Conference, pages 384-396; 2002.

Module

Foundation

24.4 Region Growing

```
expand_gray ( Regions, Image,
    ForbiddenArea : RegionExpand : Iterations, Mode, Threshold : )
```

Fill gaps between regions (depending on gray value or color) or split overlapping regions.

`expand_gray` closes gaps between the input regions, which resulted from the suppression of small regions in a segmentation operator, (mode *'image'*), for example, or separates overlapping regions *'region'*). Both uses result from the expansion of regions. The operator works by adding a one pixel wide "strip" to a region, in which the gray values or color are different from the gray values or color of neighboring pixels on the region's border by at most `Threshold` (in each channel). For images of type *'cyclic'* (e.g., direction images), also points with a gray value difference of at least $255 - $ `Threshold` are added to the output region.

The expansion takes place only in regions, which are designated as not "forbidden" (parameter `ForbiddenArea`). The number of iterations is determined by the parameter `Iterations`. By passing *'maximal'*, `expand_gray` iterates until convergence, i.e., until no more changes occur. By passing 0 for this parameter, all non-overlapping regions are returned. The two modes of operation (*'image'* and *'region'*) are different in the following ways:

'image' The input regions are expanded iteratively until they touch another region or the image border, or the expansion stops because of too high gray value differences. Because `expand_gray` processes all regions simultaneously, gaps between regions are distributed evenly to all regions with a similar gray value. Overlapping regions are split by distributing the area of overlap evenly to both regions.

'region' No expansion of the input regions is performed. Instead, only overlapping regions are split by distributing the area of overlap evenly to regions having a matching gray value or color.

Attention

Because regions are only expanded into areas having a matching gray value or color, usually gaps will remain between the output regions, i.e., the segmentation is not complete.

Parameters

▷ **`Regions`** (input_object) ..... region(-array) ⇝ *object*
Regions for which the gaps are to be closed, or which are to be separated.

▷ **`Image`** (input_object) ..... (multichannel-)image ⇝ *object :* byte / cyclic
Image (possibly multi-channel) for gray value or color comparison.

▷ **`ForbiddenArea`** (input_object) ..... region ⇝ *object*
Regions in which no expansion takes place.

▷ **`RegionExpand`** (output_object) ..... region(-array) ⇝ *object*
Expanded or separated regions.

▷ **`Iterations`** (input_control) ..... string ⇝ *string* / integer
Number of iterations.
Default: 'maximal'
Suggested values: `Iterations` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'maximal'}
Value range: $1 \leq$ `Iterations` ≤ 500 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **`Mode`** (input_control) ..... string ⇝ *string*
Expansion mode.
Default: 'image'
List of values: `Mode` ∈ {'image', 'region'}

▷ **Threshold** (input_control) . . . . . integer(-array) ⇝ *integer*
Maximum difference between the gray value or color at the region's border and a candidate for expansion.
Default: 32
Suggested values: Threshold ∈ {5, 10, 15, 20, 25, 30, 40, 50}
Value range: 1 ≤ Threshold ≤ 255 (lin)
Minimum increment: 1
Recommended increment: 5

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  HImage   image (argv[1]);
  HRegion  empty_region;
  HWindow  win;

  image.Display (win);

  HRegionArray seg = (image >= 100).Connection ();

  seg.Display (win);
  HRegionArray exp = seg.ExpandGray1 (image, empty_region,
                                      "maximal", "image", 32);
  win.SetDraw ("margin");
  win.SetColored (12);
  exp.Display (win);
  win.Click ();

  return (0);
}
```

Result

expand_gray always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>), the behavior in case of an empty input region via set_system('empty_region_result',<Result>), and the behavior in case of an empty result region via set_system('store_empty_region',<'true'/'false'>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

connection, regiongrowing, pouring, class_ndim_norm

Possible Successors

select_shape

See also

expand_gray_ref, expand_region

Module

Foundation

```
expand_gray_ref ( Regions, Image,
    ForbiddenArea : RegionExpand : Iterations, Mode, RefGray,
    Threshold : )
```

Fill gaps between regions (depending on gray value or color) or split overlapping regions.

`expand_gray_ref` closes gaps between the input regions, which resulted from the suppression of small regions in a segmentation operator, (mode *'image'*), for example, or separates overlapping regions *'region'*). Both uses result from the expansion of regions. The operator works by adding a one pixel wide "strip" to a region, in which the gray values or color are different from a reference gray value or color by at most `Threshold` (in each channel). For images of type *'cyclic'* (e.g., direction images), also points with a gray value difference of at least $255 - $ `Threshold` are added to the output region.

The expansion takes place only in regions, which are designated as not "forbidden" (parameter `ForbiddenArea`). The number of iterations is determined by the parameter `Iterations`. By passing *'maximal'*, `expand_gray_ref` iterates until convergence, i.e., until no more changes occur. By passing 0 for this parameter, all non-overlapping regions are returned. The two modes of operation (*'image'* and *'region'*) are different in the following ways:

'image' The input regions are expanded iteratively until they touch another region or the image border, or the expansion stops because of too high gray value differences. Because `expand_gray_ref` processes all regions simultaneously, gaps between regions are distributed evenly to all regions with a similar gray value. Overlapping regions are split by distributing the area of overlap evenly to both regions.

'region' No expansion of the input regions is performed. Instead, only overlapping regions are split by distributing the area of overlap evenly to regions having a matching gray value or color.

Attention

Because regions are only expanded into areas having a matching gray value or color, usually gaps will remain between the output regions, i.e., the segmentation is not complete.

Parameters

▷ **`Regions`** (input_object) . . . . . region(-array) ⇝ *object*
Regions for which the gaps are to be closed, or which are to be separated.

▷ **`Image`** (input_object) . . . . . (multichannel-)image ⇝ *object :* byte / cyclic
Image (possibly multi-channel) for gray value or color comparison.

▷ **`ForbiddenArea`** (input_object) . . . . . region ⇝ *object*
Regions in which no expansion takes place.

▷ **`RegionExpand`** (output_object) . . . . . region(-array) ⇝ *object*
Expanded or separated regions.

▷ **`Iterations`** (input_control) . . . . . string ⇝ *string* / integer
Number of iterations.
Default: 'maximal'
Suggested values: `Iterations` ∈ {'maximal', 1, 2, 3, 4, 5, 7, 10, 15, 20, 30, 50, 70, 100, 150, 200, 300, 500}
Value range: $1 \leq$ `Iterations` ≤ 500 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **`Mode`** (input_control) . . . . . string ⇝ *string*
Expansion mode.
Default: 'image'
List of values: `Mode` ∈ {'image', 'region'}

▷ **`RefGray`** (input_control) . . . . . integer(-array) ⇝ *integer*
Reference gray value or color for comparison.
Default: 128
Suggested values: `RefGray` ∈ {1, 10, 20, 50, 100, 128, 200, 255}
Value range: $1 \leq$ `RefGray` ≤ 255 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **Threshold** (input_control) . . . . . integer(-array) ⇝ *integer*
Maximum difference between the reference gray value or color and a candidate for expansion.
Default: 32
Suggested values: Threshold ∈ {4, 10, 15, 20, 25, 30, 40}
Value range: 1 ≤ Threshold ≤ 255 (lin)
Minimum increment: 1
Recommended increment: 5

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  HImage   image (argv[1]);
  HRegion  empty_region;
  HWindow  win;

  win.SetDraw ("margin");
  win.SetColored (12);

  image.Display (win);

  HRegionArray seg = (image >= 100).Connection ();
  seg.Display (win);

  Tuple iter = "maximal";
  Tuple mode = "image";
  Tuple refg = 128;
  Tuple thrs = 32;

  HRegionArray exp = seg.ExpandGrayRef (image, empty_region,
                                        iter, mode, refg, thrs);
  exp.Display (win);
  win.Click ();

  return (0);
}
```

Result

expand_gray_ref always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result',<Result>), the behavior in case of an empty input region via set_system('empty_region_result',<Result>), and the behavior in case of an empty result region via set_system('store_empty_region',<'true'/'false'>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

connection, regiongrowing, pouring, class_ndim_norm

Possible Successors

select_shape

See also

expand_gray, expand_region

Module

Foundation

regiongrowing (Image : Regions : RasterHeight, RasterWidth, Tolerance, MinSize :)

Segment an image using regiongrowing.

regiongrowing segments images into regions of the same intensity - rastered into rectangles of size RasterHeight × RasterWidth. In order to decide whether two adjacent rectangles belong to the same region only the gray value of their center points is used. If the gray value difference is less then or equal to Tolerance the rectangles are merged into one region.

If g_1 and g_2 are two gray values to be examined, they are merged into the same region if:

$$|g_1 - g_2| < \texttt{Tolerance}$$

For images of type 'cyclic', the following formulas are used:

$$\begin{aligned} (|g_1 - g_2| < \texttt{Tolerance}) \quad &\wedge \quad (|g_1 - g_2| \leq 127) \\ (256 - |g_1 - g_2| < \texttt{Tolerance}) \quad &\wedge \quad (|g_1 - g_2| > 127) \end{aligned}$$

For rectangles larger than one pixel, usually the images should be smoothed with a lowpass filter with a size of at least RasterHeight × RasterWidth before calling regiongrowing (so that the gray values at the centers of the rectangles are "representative" for the whole rectangle). If the image contains little noise and the rectangles are small, the smoothing can be omitted in many cases.

The resulting regions are collections of rectangles of the chosen size RasterHeight × RasterWidth. Only regions containing at least MinSize points are returned.

Regiongrowing is a very fast operation, and thus suited for time-critical applications.

Attention

RasterWidth and RasterHeight are automatically converted to odd values if necessary.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / int4 / real

Input image.

▷ **Regions** (output_object) ..... region-array ⇝ *object*

Segmented regions.

▷ **RasterHeight** (input_control) ..... extent.y ⇝ *integer*

Vertical distance between tested pixels (height of the raster).
Default: 3
Suggested values: RasterHeight ∈ {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}
Value range: 1 ≤ RasterHeight ≤ 99 (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: RasterHeight >= 1 && odd(RasterHeight)

▷ **RasterWidth** (input_control) ..... extent.x ⇝ *integer*

Horizontal distance between tested pixels (height of the raster).
Default: 3
Suggested values: RasterWidth ∈ {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21}
Value range: 1 ≤ RasterWidth ≤ 99 (lin)
Minimum increment: 2
Recommended increment: 2
Restriction: RasterWidth >= 1 && odd(RasterWidth)

▷ **Tolerance** (input_control) ..... number ⇝ *real* / integer
Points with a gray value difference less then or equal to tolerance are accumulated into the same object.
Default: 6.0
Suggested values: Tolerance ∈ {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 12.0, 14.0, 18.0, 25.0}
Value range: 0 ≤ Tolerance (lin)
Minimum increment: 0.01
Recommended increment: 1.0

▷ **MinSize** (input_control) ..... integer ⇝ *integer*
Minimum size of the output regions.
Default: 100
Suggested values: MinSize ∈ {1, 5, 10, 20, 50, 100, 200, 500, 1000}
Value range: 1 ≤ MinSize
Minimum increment: 1
Recommended increment: 5

Example

```
read_image(Image,'fabrik')
mean_image(Image,Mean,RasterHeight,RasterWidth)
regiongrowing(Mean,Result,RasterHeight,RasterWidth,6.0,100)
```

Complexity

Let N be the number of found regions and M the number of points in one of these regions. Then the runtime complexity is $O(N * log(M) * M)$.

Result

regiongrowing returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

binomial_filter, mean_image, gauss_filter, smooth_image, median_image, anisotropic_diffusion

Possible Successors

select_shape, reduce_domain, select_gray

Alternatives

regiongrowing_n, regiongrowing_mean, label_to_region

Module

Foundation

regiongrowing_mean (Image : Regions : StartRows, StartColumns, Tolerance, MinSize :)

Perform a regiongrowing using mean gray values.

regiongrowing_mean performs a regiongrowing using the mean gray values of a region, starting from points given by StartRows and StartColumns. At any point in the process the mean gray value of the current region is calculated. Gray values at the boundary of the region are added to the region if they differ from the current mean by less than Tolerance. Regions smaller than MinSize are suppressed.

If no starting points are given (empty tuples), the expansion process starts at the upper leftmost point, and is continued with the first unprocessed point after a region has been created.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / int4
Input image.

▷ **Regions** (output_object) . . . . . region-array ⇝ *object*
Segmented regions.

▷ **StartRows** (input_control) . . . . . point.y(-array) ⇝ *integer*
Row coordinates of the starting points.
Default: []
Value range: 0 ≤ StartRows
Minimum increment: 1
Recommended increment: 1

▷ **StartColumns** (input_control) . . . . . point.x(-array) ⇝ *integer*
Column coordinates of the starting points.
Default: []
Value range: 0 ≤ StartColumns
Minimum increment: 1
Recommended increment: 1

▷ **Tolerance** (input_control) . . . . . number ⇝ *real*
Maximum deviation from the mean.
Default: 5.0
Suggested values: `Tolerance` ∈ {0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 12.0, 15.0, 17.0, 20.0, 25.0, 30.0, 40.0}
Restriction: Tolerance > 0.0

▷ **MinSize** (input_control) . . . . . integer ⇝ *integer*
Minimum size of a region.
Default: 100
Suggested values: `MinSize` ∈ {0, 10, 30, 50, 100, 500, 1000, 2000}
Value range: 0 ≤ MinSize
Minimum increment: 1
Recommended increment: 100

Result

`regiongrowing_mean` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`binomial_filter`, `gauss_filter`, `sigma_image`, `anisotropic_diffusion`, `median_image`, `mean_image`

Possible Successors

`select_shape`, `reduce_domain`, `opening`, `expand_region`

Alternatives

`regiongrowing`, `regiongrowing_n`

Module

Foundation

regiongrowing_n (MultiChannelImage : Regions : Metric, MinTolerance, MaxTolerance, MinSize :)

Segment an image using regiongrowing for multi-channel images.

`regiongrowing_n` performs a multi-channel regiongrowing. The n channels give rise to an n-dimensional feature vector. Neighboring points are aggregated into the same region if the difference of their feature vectors with respect to the given metric lies in the interval [`MinTolerance`, `MaxTolerance`]. Only neighbors of the 4-neighborhood are examined. The following metrics can be used:

Let g_A denote the gray value in the feature vector A at point a of the image, and likewise be g_B the gray value in the feature vector B at point a neighboring point b. Let $g(d)$ be the gray value with index d. Furthermore, let $MinT$ denote `MinTolerance` and $MaxT$ denote `MaxTolerance`.

'1-norm': Sum of absolute values

$$MinT \leq \frac{1}{n} \sum |g_A - g_B| \leq MaxT$$

'2-norm': Euclidean distance

$$MinT \leq \sqrt{\frac{\sum (g_A - g_B)^2}{n}} \leq MaxT$$

'3-norm': p - Norm with p = 3

$$MinT \leq \sqrt[3]{\frac{\sum (g_A - g_B)^3}{n}} \leq MaxT$$

'4-norm': p - Norm with p = 4

$$MinT \leq \sqrt[4]{\frac{\sum (g_A - g_B)^4}{n}} \leq MaxT$$

'n-norm': Minkowski distance

$$MinT \leq \sqrt[n]{\frac{\sum (g_A - g_B)^n}{n}} \leq MaxT$$

'max-diff': Supremum distance

$$MinT \leq \max \{|g_A - g_B|\} \leq MaxT$$

'min-diff': Infimum distance

$$MinT \leq \min \{|g_A - g_B|\} \leq MaxT$$

'variance': Variance of gray value differences

$$MinT \leq Var(g_A - g_B) \leq MaxT$$

'dot-product': Dot product

$$MinT \leq \frac{1}{n} \sqrt{\sum (g_A g_B)} \leq MaxT$$

'correlation': Correlation

$$\begin{aligned} m_A &= \frac{1}{n} \sum g_A \\ Var_A &= \frac{1}{n} \sqrt{\sum (g_A - m_A)^2} \\ m_B &= \frac{1}{n} \sum g_B \\ Var_B &= \frac{1}{n} \sqrt{\sum (g_B - m_B)^2} \\ MinT &\leq \frac{1}{n^2} \sum \frac{(g_A - m_A)(g_B - m_B)}{(Var_A Var_B)} \leq MaxT \end{aligned}$$

'mean-diff': Difference of arithmetic means

$$\begin{aligned} a &= \frac{1}{n} \sum g_A \\ b &= \frac{1}{n} \sum g_B \\ MinT &\leq |a - b| \leq MaxT \end{aligned}$$

'mean-ratio': Ratio of arithmetic means

$$\begin{aligned} a &= \frac{1}{n}\sum g_A \\ b &= \frac{1}{n}\sum g_B \\ MinT &\leq \min\left\{\frac{a}{b},\frac{b}{a}\right\} \leq MaxT \end{aligned}$$

'length-diff': Difference of the vector lengths

$$\begin{aligned} a &= \sqrt{\frac{\sum g_A^2}{n}} \\ b &= \sqrt{\frac{\sum g_B^2}{n}} \\ MinT &\leq |a-b| \leq MaxT \end{aligned}$$

'length-ratio': Ratio of the vector lengths

$$\begin{aligned} a &= \sqrt{\frac{\sum g_A^2}{n}} \\ b &= \sqrt{\frac{\sum g_B^2}{n}} \\ MinT &\leq \min\left\{\frac{a}{b},\frac{b}{a}\right\} \leq MaxT \end{aligned}$$

'n-norm-ratio': Ratio of the vector lengths w.r.t the p-norm with $p = n$

$$\begin{aligned} a &= \sqrt[n]{\frac{\sum g_A^n}{n}} \\ b &= \sqrt[n]{\frac{\sum g_B^n}{n}} \\ MinT &\leq \min\left\{\frac{a}{b},\frac{b}{a}\right\} \leq MaxT \end{aligned}$$

'gray-max-diff': Difference of the maximum gray values

$$\begin{aligned} a &= \max\{|g_A|\} \\ b &= \max\{|g_B|\} \\ MinT &\leq |a-b| \leq MaxT \end{aligned}$$

'gray-max-ratio': Ratio of the maximum gray values

$$\begin{aligned} a &= \max\{|g_A|\} \\ b &= \max\{|g_B|\} \\ MinT &\leq \min\left\{\frac{a}{b},\frac{b}{a}\right\} \leq MaxT \end{aligned}$$

'gray-min-diff': Difference of the minimum gray values

$$\begin{aligned} a &= \min\{|g_A|\} \\ b &= \min\{|g_B|\} \\ MinT &\leq |a-b| \leq MaxT \end{aligned}$$

'gray-min-ratio': Ratio of the minimum gray values

$$\begin{aligned} a &= \min\{|g_A|\} \\ b &= \min\{|g_B|\} \\ MinT &\leq \min\left\{\frac{a}{b}, \frac{b}{a}\right\} \leq MaxT \end{aligned}$$

'variance-diff': Difference of the variances over all gray values (channels)

$$MinT \leq |Var(g_A) - Var(g_B)| \leq MaxT$$

'variance-ratio': Ratio of the variances over all gray values (channels)

$$MinT \leq \frac{Var(g_B)}{Var(g_A)} \leq MaxT$$

'mean-abs-diff': Difference of the sum of absolute values over all gray values (channels)

$$\begin{aligned} a &= \sum_{d,k,k<d} |g_A(d) - g_A(k)| \\ b &= \sum_{d,k,k<d} |g_B(d) - g_B(k)| \\ MinT &\leq \frac{|a-b|}{\text{number of sums}} \leq MaxT \end{aligned}$$

'mean-abs-ratio': Ratio of the sum of absolute values over all gray values (channels)

$$\begin{aligned} a &= \sum_{d,k,k<d} |g_A(d) - g_A(k)| \\ b &= \sum_{d,k,k<d} |g_B(d) - g_B(k)| \\ MinT &\leq \min\left\{\frac{a}{b}, \frac{b}{a}\right\} \leq MaxT \end{aligned}$$

'max-abs-diff': Difference of the maximum distance of the components

$$\begin{aligned} a &= \max\{g_A(d), g_A(k)\} \\ b &= \max\{g_B(d), g_B(k)\} \\ MinT &\leq \min\left\{\frac{a}{b}, \frac{b}{a}\right\} \leq MaxT \end{aligned}$$

'max-abs-ratio': Ratio of the maximum distance of the components

$$\begin{aligned} a &= \max\{g_A(d), g_A(k)\} \\ b &= \max\{g_B(d), g_B(k)\} \\ MinT &\leq \min\left\{\frac{a}{b}, \frac{b}{a}\right\} \leq MaxT \end{aligned}$$

'min-abs-diff': Difference of the minimum distance of the components

$$\begin{aligned} a &= \min\{g_A(d), g_A(k)\}, k < d \\ b &= \min\{g_B(d), g_B(k)\}, k < d \\ MinT &\leq |a-b| \leq MaxT \end{aligned}$$

'min-abs-ratio': Ratio of the minimum distance of the components

$$\begin{aligned} a &= \min\{g_A(d), g_A(k)\}, k < d \\ b &= \min\{g_B(d), g_B(k)\}, k < d \\ MinT &\leq \min\left\{\frac{a}{b}, \frac{b}{a}\right\} \leq MaxT \end{aligned}$$

'plane': The following has to hold for all d_1,$d_2 \in [1, n]$:

$$\begin{aligned} g_A(d_1) &> g_A(d_2) \Rightarrow g_B(d_1) > g_B(d_2) \\ g_A(d_1) &< g_A(d_2) \Rightarrow g_B(d_1) < g_B(d_2) \end{aligned}$$

Regions with an area less than `MinSize` are suppressed.

Parameters

▷ **`MultiChannelImage`** (input_object) . . . . . (multichannel-)image(-array) ⇝ *object :* byte
Input image.

▷ **`Regions`** (output_object) . . . . . region-array ⇝ *object*
Segmented regions.

▷ **`Metric`** (input_control) . . . . . string ⇝ *string*
Metric for the distance of the feature vectors.
Default: '2-norm'
List of values: `Metric` ∈ {'1-norm', '2-norm', '3-norm', '4-norm', 'n-norm', 'max-diff', 'min-diff', 'variance', 'dot-product', 'correlation', 'mean-diff', 'mean-ratio', 'length-diff', 'length-ratio', 'n-norm-ratio', 'gray-max-diff', 'gray-max-ratio', 'gray-min-diff', 'gray-min-ratio', 'variance-diff', 'variance-ratio', 'mean-abs-diff', 'mean-abs-ratio', 'max-abs-diff', 'max-abs-ratio', 'min-abs-diff', 'min-abs-ratio', 'plane'}

▷ **`MinTolerance`** (input_control) . . . . . number ⇝ *real* / integer
Lower threshold for the features' distance.
Default: 0.0
Suggested values: `MinTolerance` ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0, 25.0, 30.0}

▷ **`MaxTolerance`** (input_control) . . . . . number ⇝ *real* / integer
Upper threshold for the features' distance.
Default: 20.0
Suggested values: `MaxTolerance` ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0, 25.0, 30.0}

▷ **`MinSize`** (input_control) . . . . . integer ⇝ *integer*
Minimum size of the output regions.
Default: 30
Suggested values: `MinSize` ∈ {1, 10, 25, 50, 100, 200, 500, 1000}
Value range: 1 ≤ MinSize
Minimum increment: 1
Recommended increment: 5

Result

`regiongrowing_n` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

compose2, compose3

Alternatives

class_2dim_sup, class_ndim_norm

See also

regiongrowing_mean

Module

Foundation

24.5 Threshold

This chapter describes threshold operators.

Concept of Threshold Operators

One way to perform a segmentation of an image is to use threshold operators. In doing so, regions that fulfill a certain threshold condition, depending on gray values, are determined within an image.

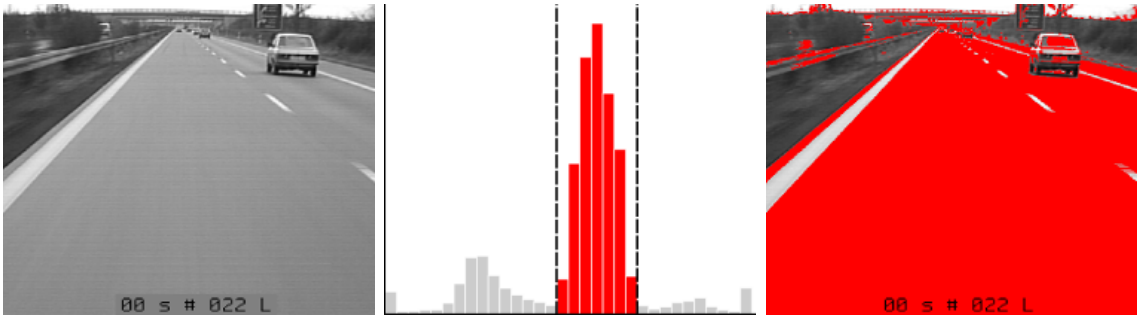
In order to fit different tasks and image properties there is a set of threshold operators provided.

The following paragraphs give an overview of the operators by differentiating them into histogram-based and local methods and having a closer look at the most important ones.

Histogram-based Threshold Operators

Histogram-based threshold segmentation does not take the position but only the pixel value into account. Therefore thresholds are determined by adjusting them to the histogram of an image.

Threshold By using threshold, you can select all pixels within user set gray value intervals.

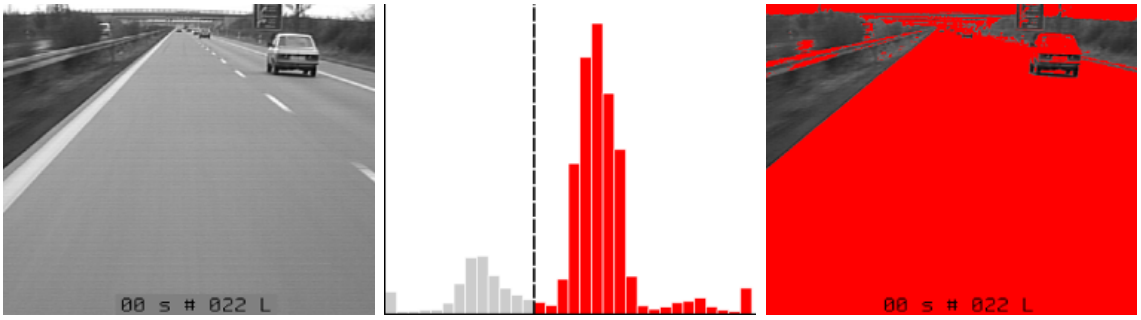


(1) (2) (3)

(1) Input image, (2) histogram of the input image with manually determined thresholds MinGray and MaxGray, (3) resulting segmentation of the input image.

The operator fast_threshold also works with two manually determined thresholds but uses another computing algorithm.

Binary Threshold To divide your image into light and dark regions, binary_threshold automatically computes a threshold to separate foreground from background.

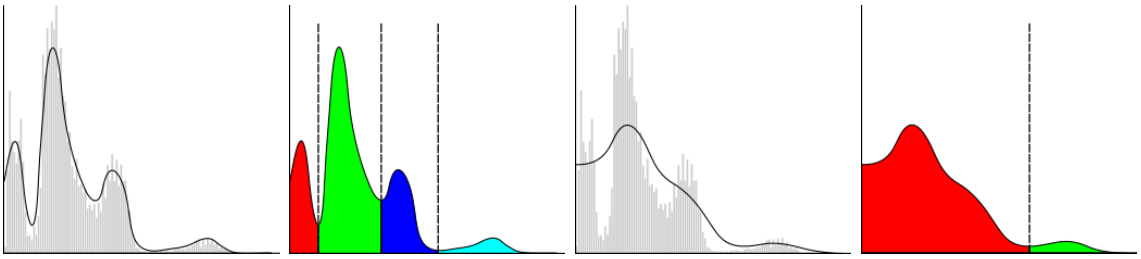


(1) (2) (3)

(1) Input image, (2) histogram of the input image with automatically determined binary threshold, (3) resulting segmentation of the input image.

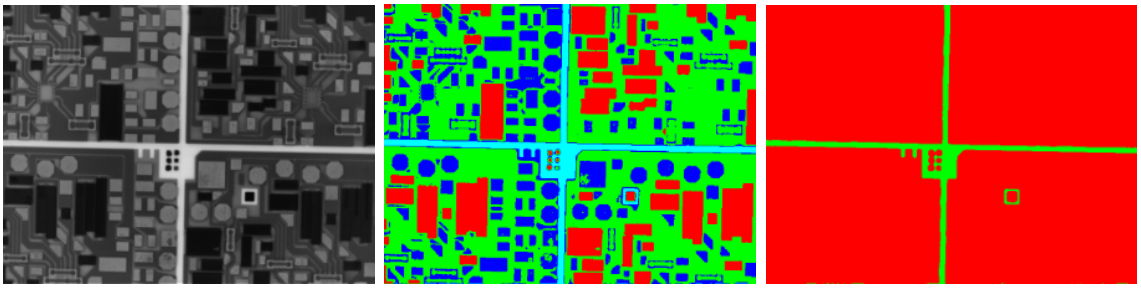
A segmentation using `threshold_sub_pix` also divides the image into foreground and background, but puts out the separating border with sub-pixel accuracy. The threshold has to be set manually.

Automatic Threshold `auto_threshold` computes local minima in the image histogram to determine thresholds. By smoothing the histogram you have influence on the number of classes found in the input image.



(1) (2) (3) (4)

(1) Slightly smoothed histogram, (2) results in higher number of local minima, (3) with further smoothing, (4) number of local minima decreases.

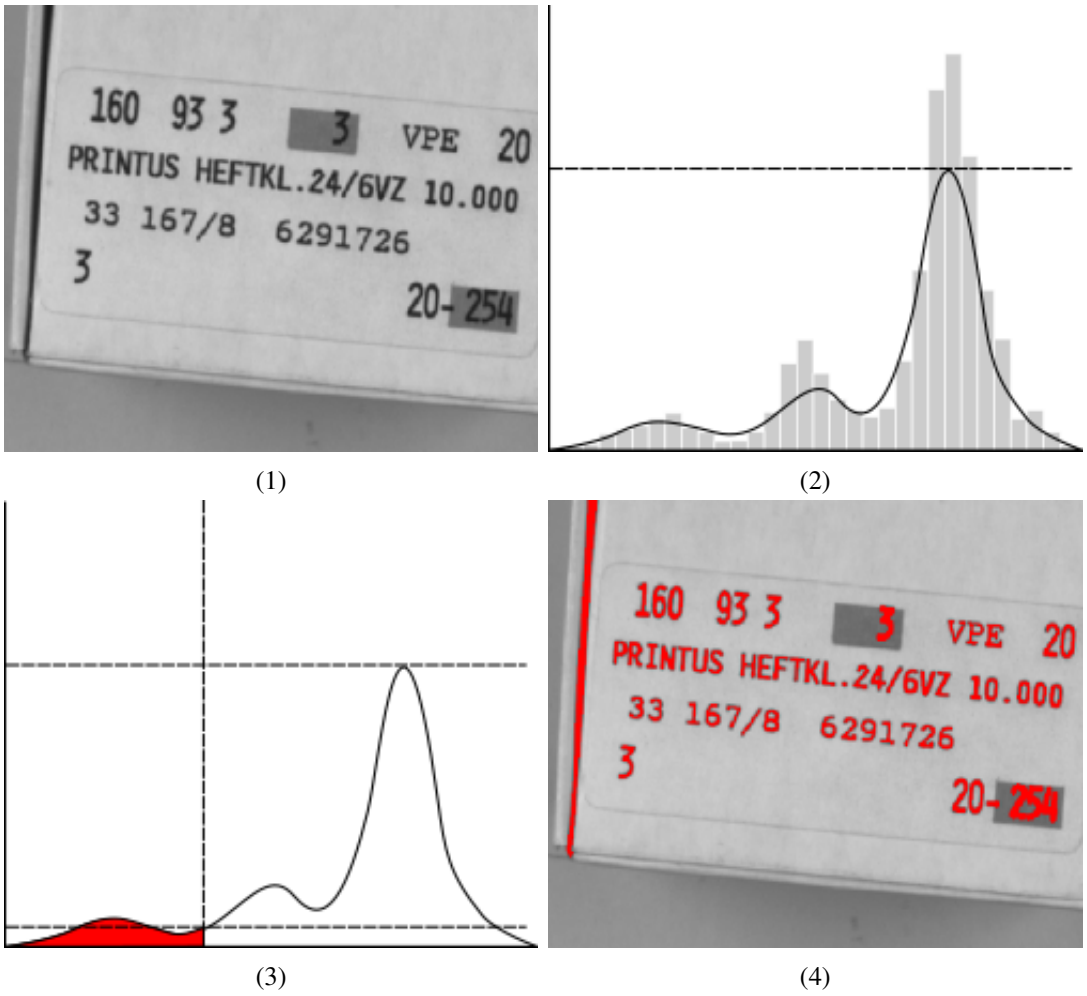


(1) (2) (3)

(1) Input image, (2) four classes are extracted from slightly smoothed histogram, (3) whereas in this case a further smoothed histogram results in two regions.

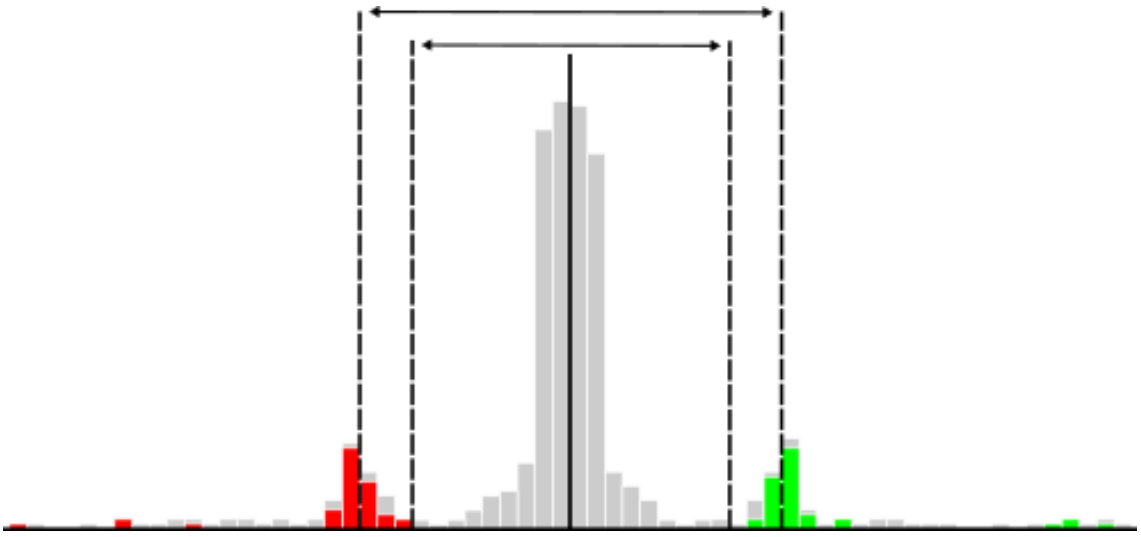
Use `histo_to_thresh` to get the gray values of local minima in the histogram.

Character Threshold To do a segmentation of dark text on light background, `char_threshold` is a useful tool. The maximum peak in the histogram corresponds to the light background. Assuming the text is darker than the background, you examine the smoothed histogram left of the maximum. The parameter `Percent` determines how far from the maximum the threshold is set, taking the frequencies of the gray values into account.

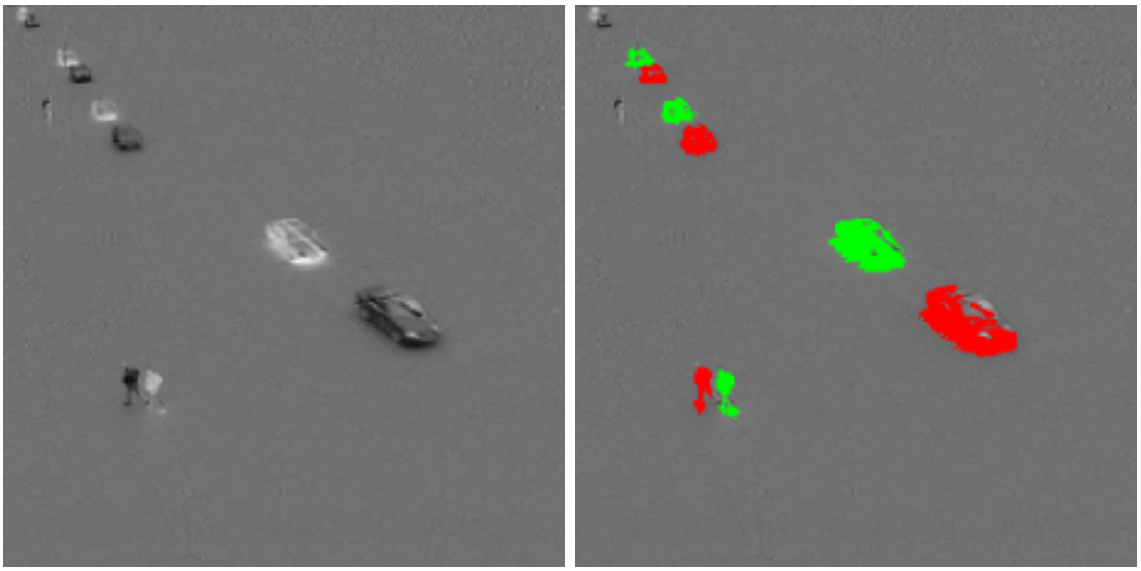


(1) Input image, (2) histogram is smoothed and global maximum is identified, (3) the parameter `Percent` determines threshold left of the maximum, (4) resulting image.

Dual-Threshold Subtracting an image from another or using an edge detecting operator like `laplace_of_gauss` usually leads to negative values in the resulting image. The operator `dual_threshold` is suited for the segmentation of signed images while also taking minimum region size into account.



Histogram of a signed image with `Threshold` (inner boundaries) and `MinGray`. Pixels are only selected if they fulfill the conditions regarding gray values and region size.



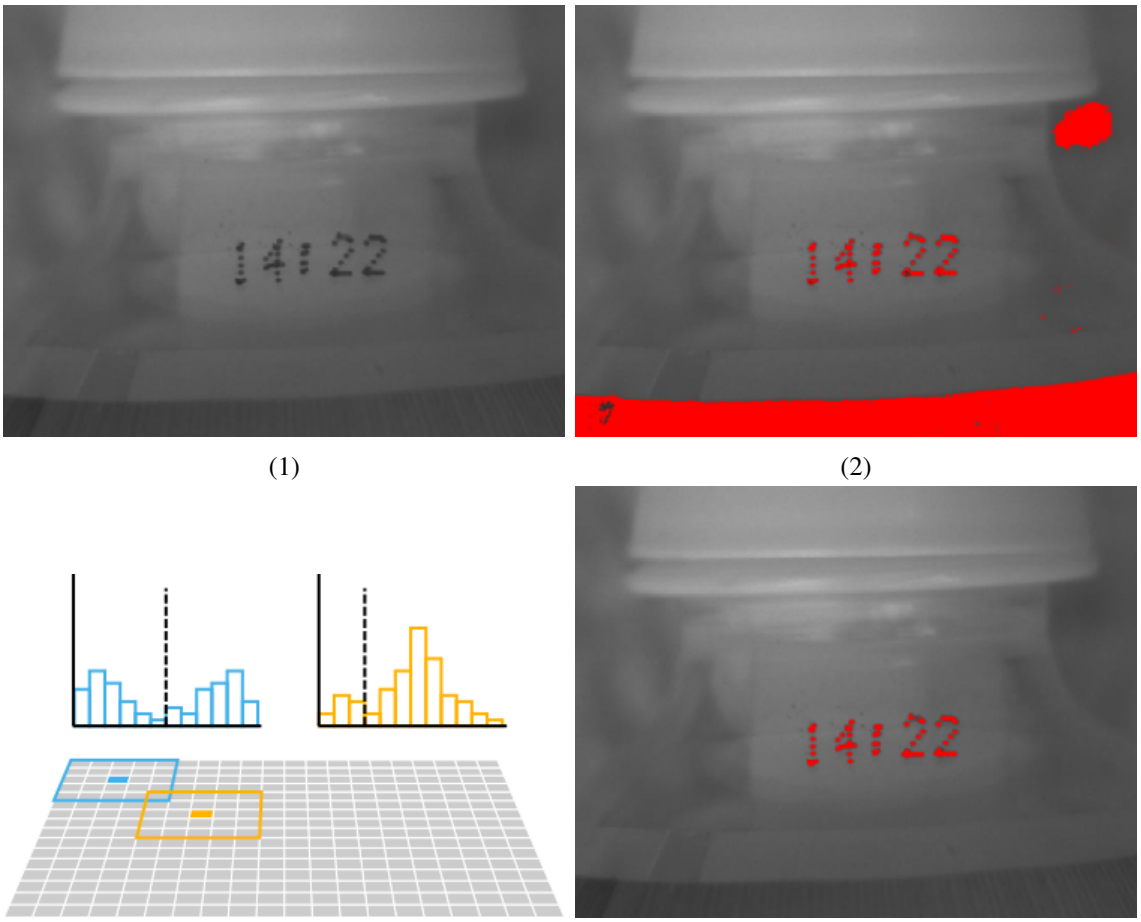
(1) (2)

(1) Signed input image (created with sub_image), (2) result after applying a dual threshold.

Local Threshold Operators

Unlike histogram-based threshold operators, a local threshold also takes position or neighborhood of pixels into account to assign them to the appropriate region. Instead of global thresholds that are applied to each pixel, it is sometimes be useful to adapt the threshold to local features of the image.

Local Threshold local_threshold takes the local mean and standard deviation into account and computes an individual threshold for each pixel. The size of the neighborhood is set by the user. This operator is especially suited for the segmentation of text, when lighting conditions or the background are not homogeneous.



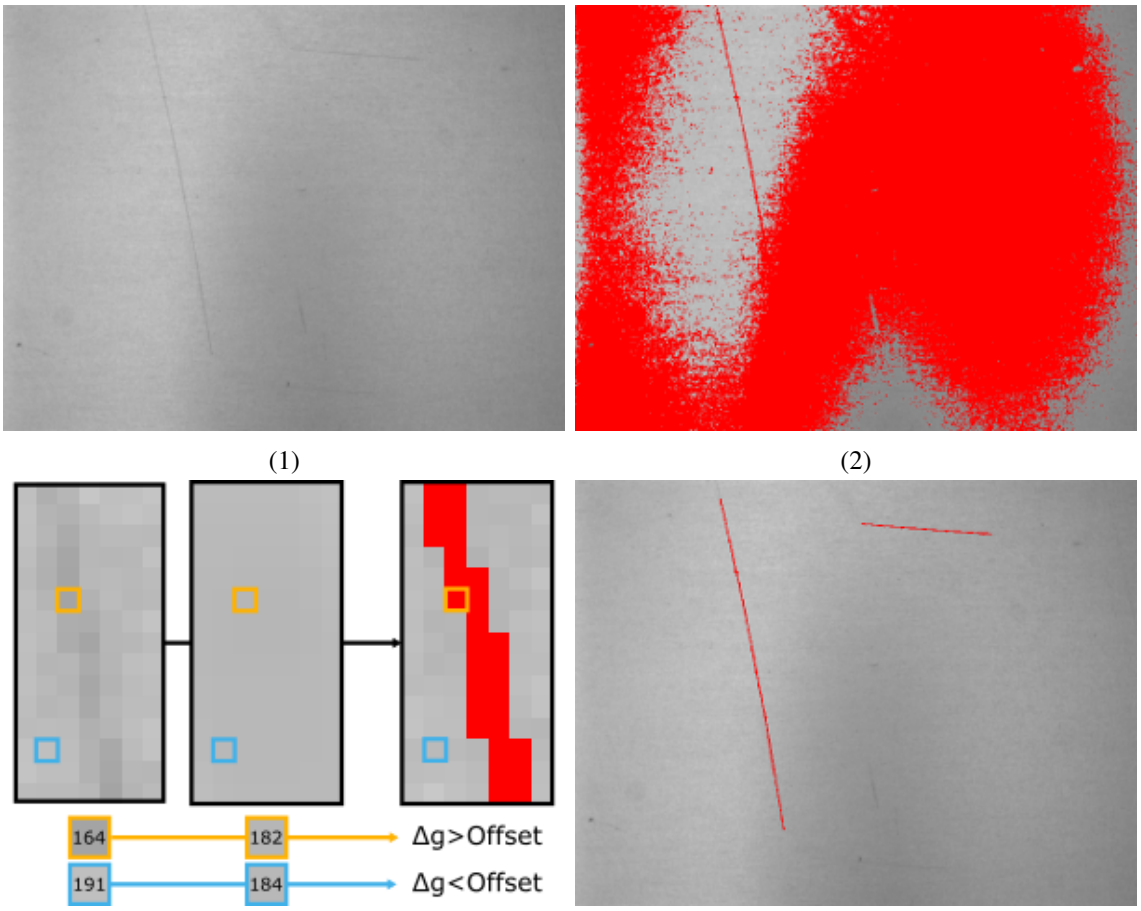
(1) (2)

(3) (4)

(1) Input image, (2) segmentation with global thresholds does not choose the desired text solely, (3) adapting the threshold to the neighborhood of each pixel individually, (4) segmentation of the text.

The operator `var_threshold` works in a similar way, except it selects those points of the image that fulfill specified conditions regarding their local standard deviation and brightness.

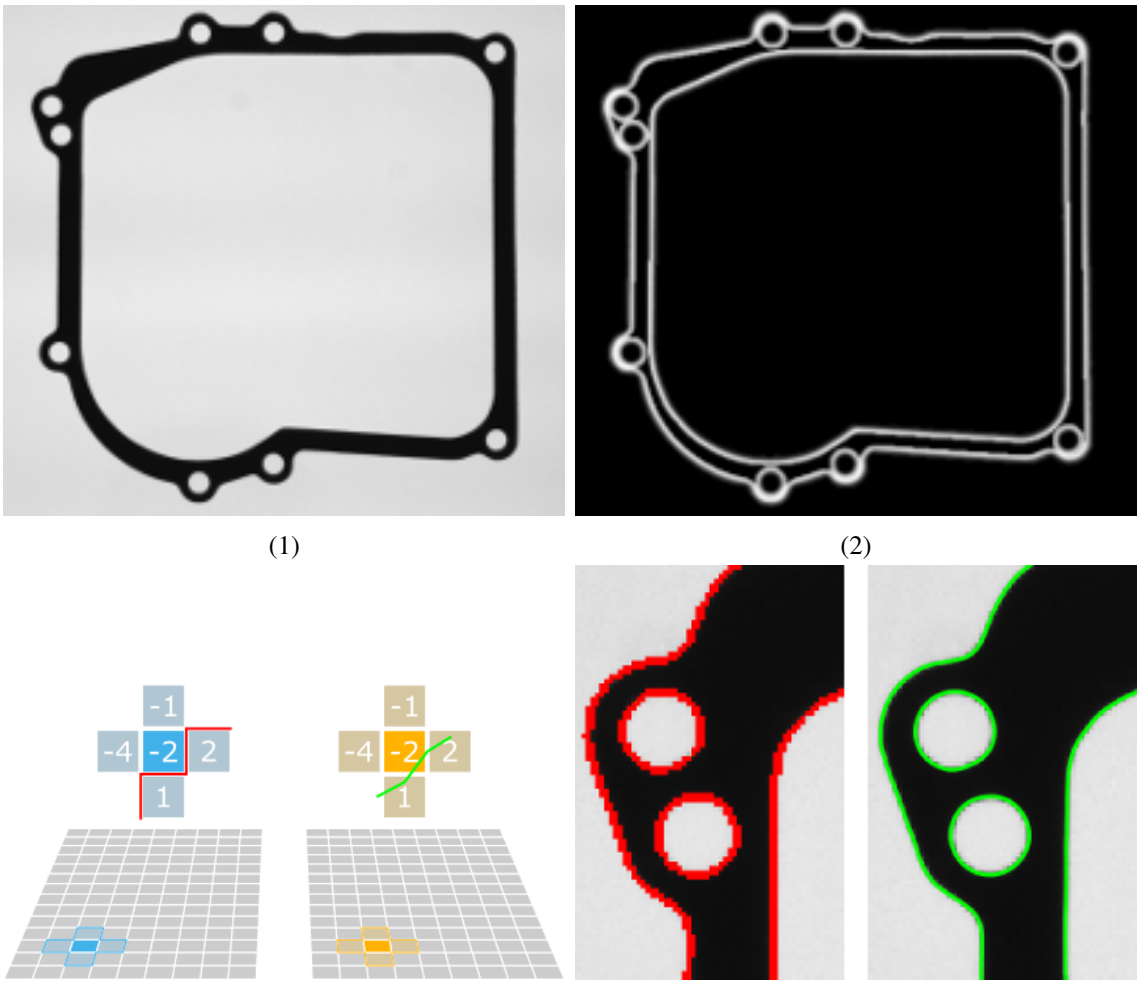
Dynamic Threshold With `dyn_threshold` you can inspect the differences between images. Usually the input image and a filtered version, e.g., the mean of the image, are compared pixel by pixel. The parameter `LightDark` is used to determine what kind of changes in the image are relevant. The sensitivity of the operator is controlled by the parameter `Offset`.



(1) Input image, (2) segmentation of scratches with global threshold is not possible due to inhomogeneous lighting conditions, (3) inspection of the gray value difference between the original image (left) and mean of the image (middle) pixel by pixel to segment regions (right) exceeding `Offset` (image detail), (4) result after selecting wanted regions. For more detail have a look at the example `surface_scratch.hdev`.

A similar operator is `check_difference`. The operator displays the absolute differences between two images. It is especially suited for change detection of consecutively acquired images.

Zero Crossing Threshold Operators like `laplace_of_gauss` that are used for edge detection return signed images, where edges are located at the zero crossings. `zero_crossing` and `zero_crossing_sub_pix` can be used to extract those edges, taking the individual 4-neighborhood into account.



(1) Input image, (2) domain is reduced and Laplace-of-Gaussian filter is applied, (3) the zero-crossing operators can detect edges with pixel (red) and sub-pixel accuracy (green), (4) image details of results using `zero_crossing` (red) and `zero_crossing_sub_pix` (green).

auto_threshold (Image : Regions : Sigma :)

Segment an image using thresholds determined from its histogram.

`auto_threshold` segments a single-channel image using multiple thresholding. First, the absolute histogram of the gray values is determined. Then, relevant minima are extracted from the histogram, which are used successively as parameters for a thresholding operation. The thresholds used for byte images are 0, 255, and all minima extracted from the histogram (after the histogram has been smoothed with a Gaussian filter with standard deviation `Sigma`). For each gray value interval *one* region is generated. Thus, the number of regions is the number of minima + 1. For uint2 images, the above procedure is used analogously. However, here the highest threshold is 65535. Furthermore, for uint2 images the value of `Sigma` (virtually) refers to a histogram with 256 values, although internally histograms with a higher resolution are used. This is done to facilitate switching between image types without having to change the parameter `Sigma`. For float images the thresholds are the minimum and maximum gray value in the image and all minima extracted from the histogram. Here, the scaling of the parameter `Sigma` refers to the original gray values of the image. The larger the value of `Sigma` is chosen, the fewer regions will be extracted. This operator is useful if the regions to be extracted exhibit similar gray values (homogeneous regions).

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **Regions** (output_object) . . . . . region-array ⇝ *object*
Regions with gray values within the automatically determined intervals.

▷ **Sigma** (input_control) . . . . . number ⇝ *real* / integer
Sigma for the Gaussian smoothing of the histogram.
Default: 2.0
Suggested values: Sigma ∈ {0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0}
(lin)
Minimum increment: 0.01
Recommended increment: 0.3

Example

```
read_image (Image, 'fabrik')
median_image (Image, Median, 'circle', 3, 'mirrored')
auto_threshold (Median, Seg, 2.0)
connection (Seg, Connected)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

anisotropic_diffusion, median_image, illuminate

Possible Successors

connection, select_shape, select_gray

Alternatives

binary_threshold, char_threshold

See also

gray_histo, gray_histo_abs, histo_to_thresh, smooth_funct_1d_gauss, threshold

Module

Foundation

binary_threshold (Image : Region : Method,
LightDark : UsedThreshold)

Segment an image using binary thresholding.

binary_threshold segments a single-channel Image using an automatically determined global threshold and returns the segmented region in Region. This is, e.g., useful for the segmentation of characters on a homogeneously illuminated background. binary_threshold also returns the used threshold in UsedThreshold.

The used threshold is determined by the method that is given in Method. Currently the operator provides the following two methods: *'max_separability'* and *'smooth_histo'*. Both methods should only be used for images that have a bimodal histogram.

The method *'smooth_histo'* provides the same functionality that has been provided by the operator bin_threshold. The method *'max_separability'* tends to determine smaller values for UsedThreshold. Furthermore, it is less sensitive to thin isolated peaks in the histogram that are far from the rest of the spectrum and often, it is faster than *'smooth_histo'*.

Maximize separability

By selecting Method = *'max_separability'*, automatic thresholding based on the gray-level histogram according to Otsu (see the paper in References) is invoked. The algorithm first calculates the histogram of the image and then uses statistical moments to find the optimal threshold T^* that divides the pixels into foreground and background and maximizes the separability between these two classes. This method is only available for byte and uint2 images.

- If LightDark = *'light'* all pixels with gray values greater or equal to T^* are selected.

- If LightDark = *'dark'* all pixels with gray values smaller than T^* are selected.

Histogram smoothing

By selecting Method = *'smooth_histo'* binary_threshold determines the threshold T^* in the following way: First, the relative histogram of the gray values is determined. Then, relevant minima are extracted from the histogram, which are used as parameters for a thresholding operation. In order to reduce the number of minima, the histogram is smoothed with a Gaussian, as in auto_threshold. The mask size is enlarged until there is only one minimum in the smoothed histogram. Then, the threshold T^* is set to the position of this minimum.

- If LightDark = *'light'* all pixels $p(r, c)$ with gray values greater or equal to T^* are selected.
- If LightDark = *'dark'* all pixels $p(r, c)$ with gray values smaller than T^* are selected.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2
Input Image.

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Segmented output region.

▷ **Method** (input_control) ..... string ⇝ *string*
Segmentation method.
Default: 'max_separability'
List of values: Method ∈ {'max_separability', 'smooth_histo'}

▷ **LightDark** (input_control) ..... string ⇝ *string*
Extract foreground or background?
Default: 'dark'
List of values: LightDark ∈ {'dark', 'light'}

▷ **UsedThreshold** (output_control) ..... number(-array) ⇝ *integer* / string
Used threshold.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Successors

connection, select_shape, select_gray

Alternatives

auto_threshold, char_threshold, local_threshold

See also

gray_histo, threshold

References

N. Otsu, "A threshold selection method from gray level histograms", IEEE Trans. Syst. Man. Cybern., Vol. SMC-9, 62-66 (1979).

Module

Foundation

char_threshold (Image, HistoRegion : Characters : Sigma, Percent : Threshold)

Perform a threshold segmentation for extracting characters.

The main application of char_threshold is to segment single-channel images of dark characters on bright paper. The operator works as follows: First, a histogram of the gray values in the image Image is computed for the points in the region HistoRegion. To eliminate noise, the histogram is smoothed with the given Sigma

(Gaussian smoothing). In the histogram, the background (white paper) corresponds to a large peak at high gray values, while the characters form a small peak at low gray values. In contrast to the operator binary_threshold (with Method='*smooth_histo*'), which locates the minimum between the two peaks, here the threshold for the segmentation is determined in relation to the maximum of the histogram, i.e., the background, with the following condition:

$$histogram[\text{threshold}] * 100.0 < histogram[\text{maximum}] * (100.0 - \texttt{Percent})$$

For example, if you choose Percent = 95 the operator locates the gray value whose frequency is at most 5 percent of the maximum frequency. Because char_threshold assumes that the characters are darker than the background, the threshold is searched for "to the left" of the maximum.

In comparison to binary_threshold, this operator should be used if there is no clear minimum between the histogram peaks corresponding to the characters and the background, respectively, or if there is no peak corresponding to the characters at all. This may happen, e.g., if the image contains only few characters or in the case of a non-uniform illumination.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
  Input image.
- ▷ **HistoRegion** (input_object) ..... region ⇝ *object*
  Region in which the histogram is computed.
- ▷ **Characters** (output_object) ..... region(-array) ⇝ *object*
  Dark regions (characters).
- ▷ **Sigma** (input_control) ..... number ⇝ *real*
  Sigma for the Gaussian smoothing of the histogram.
  Default: 2.0
  Suggested values: Sigma ∈ {0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0}
  Value range: 0.0 ≤ Sigma ≤ 50.0 (lin)
  Minimum increment: 0.01
  Recommended increment: 0.2
- ▷ **Percent** (input_control) ..... number ⇝ *real* / integer
  Percentage for the gray value difference.
  Default: 95
  Suggested values: Percent ∈ {90, 92, 95, 96, 97, 98, 99, 99.5, 100}
  Value range: 0.0 ≤ Percent ≤ 100.0 (lin)
  Minimum increment: 0.1
  Recommended increment: 0.5
- ▷ **Threshold** (output_control) ..... integer(-array) ⇝ *integer*
  Calculated threshold.

Example

```
read_image (Image, 'letters')
char_threshold (Image, Image, Seg, 0.0, 5.0, Threshold)
connection (Seg, Connected)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

anisotropic_diffusion, median_image, illuminate

Possible Successors

connection, select_shape, select_gray

Alternatives

binary_threshold, auto_threshold, gray_histo, smooth_funct_1d_gauss, threshold

Module

Foundation

check_difference (Image, Pattern : Selected : Mode, DiffLowerBound, DiffUpperBound, GrayOffset, AddRow, AddCol :)

Compare two images pixel by pixel.

check_difference selects from the input image Image those pixels ($g_o = g_{\texttt{Image}}$), whose gray value difference to the corresponding pixels in Pattern is inside (outside) of the interval [DiffLowerBound, DiffUpperBound]. The pixels of Pattern are translated by (AddRow, AddCol) with respect to Image. Let g_p be the gray value from Pattern translated by (AddRow, AddCol) with respect to g_o.

If the selected mode Mode is *'diff_inside'*, a pixel g_o is selected if

$$g\_o - g\_p - \texttt{GrayOffset} \geq \texttt{DiffLowerBound} \quad \text{and}$$
$$g\_o - g\_p - \texttt{GrayOffset} \leq \texttt{DiffUpperBound}.$$

If the mode is set to *'diff_outside'*, a pixel g_o is selected if

$$g\_o - g\_p - \texttt{GrayOffset} < \texttt{DiffLowerBound} \quad \text{or}$$
$$g\_o - g\_p - \texttt{GrayOffset} > \texttt{DiffUpperBound}.$$

This test is performed for all points of the domain (region) of Image, intersected with the domain of the translated Pattern. All points fulfilling the above condition are aggregated in the output region. The two images may be of different size. Typically, Pattern is smaller than Image.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
Input image.

▷ **Pattern** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte
Comparison image.

▷ **Selected** (output_object) ..... region(-array) ⇝ *object*
Points in which the two images are similar/different.

▷ **Mode** (input_control) ..... string ⇝ *string*
Mode: return similar or different pixels.
Default: 'diff_outside'
Suggested values: Mode ∈ {'diff_inside', 'diff_outside'}

▷ **DiffLowerBound** (input_control) ..... number ⇝ *integer*
Lower bound of the tolerated gray value difference.
Default: -5
Suggested values: DiffLowerBound ∈ {0, -1, -2, -3, -5, -7, -10, -12, -15, -17, -20, -25, -30}
Value range: -255 ≤ DiffLowerBound ≤ 255 (lin)
Minimum increment: 1
Recommended increment: 2

▷ **DiffUpperBound** (input_control) ..... number ⇝ *integer*
Upper bound of the tolerated gray value difference.
Default: 5
Suggested values: DiffUpperBound ∈ {0, 1, 2, 3, 5, 7, 10, 12, 15, 17, 20, 25, 30}
Value range: -255 ≤ DiffUpperBound ≤ 255 (lin)
Minimum increment: 1
Recommended increment: 2

▷ **GrayOffset** (input_control) ..... number ⇝ *integer*
Offset gray value subtracted from the input image.
Default: 0
Suggested values: GrayOffset ∈ {-30, -25, -20, -17, -15, -12, -10, -7, -5, -3, -2, -1, 0, 1, 2, 3, 5, 7, 10, 12, 15, 17, 20, 25, 30}
Value range: -255 ≤ GrayOffset ≤ 255 (lin)
Minimum increment: 1
Recommended increment: 2

▷ **AddRow** (input_control) ..... point.y ⇝ *integer*
Row coordinate by which the comparison image is translated.
Default: 0
Suggested values: AddRow ∈ {-200, -100, -20, -10, 0, 10, 20, 100, 200}
Value range: -32000 ≤ AddRow ≤ 32000 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **AddCol** (input_control) ..... point.x ⇝ *integer*
Column coordinate by which the comparison image is translated.
Default: 0
Suggested values: AddCol ∈ {-200, -100, -20, -10, 0, 10, 20, 100, 200}
Value range: -32000 ≤ AddCol ≤ 32000 (lin)
Minimum increment: 1
Recommended increment: 1

Complexity

Let A be the number of valid pixels. Then the runtime complexity is $O(A)$.

Result

check_difference returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Successors

connection, select_shape, reduce_domain, select_gray, rank_region, dilation1, opening

Alternatives

sub_image, dyn_threshold

Module

Foundation

dual_threshold (Image : RegionCrossings : MinSize, MinGray, Threshold :)

Threshold operator for signed images.

dual_threshold segments the input image into a region with gray values ≥Threshold ("positive" regions) and a region with gray values ≤ −Threshold ("negative" regions). Only "positive" or "negative" regions having a size larger than MinSize are taken into account. And regions whose maximum gray value is less than MinGray in absolute value are suppressed.

The segmentation performed is not complete, i.e., the "positive" and "negative" regions together do not necessarily cover the entire image: Areas with a gray value between −Threshold and Threshold, −MinGray and MinGray, respectively, are not taken into account.

dual_threshold is usually called after applying a Laplace operator (laplace, laplace_of_gauss, derivate_gauss or diff_of_gauss) to an image or with the difference of two images (sub_image).

The zero crossings of a Laplace image correspond to edges in an image, and are the separating regions of the "positive" and "negative" regions in the Laplace image. They can be determined by calling dual_threshold with Threshold = 1 and then creating the complement regions with complement. The parameter MinGray determines the noise invariance, while MinSize determines the resolution of the edge detection.

Using byte images, only the positive part of the operator is applied. Therefore dual_threshold behaves like a standard threshold operator (threshold) with successive connection and select_gray.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / int1 / int2 / int4 / real
Input image.

▷ **RegionCrossings** (output_object) ..... region-array ⇝ *object*
Positive and negative regions.

▷ **MinSize** (input_control) ..... integer ⇝ *integer*
Regions smaller than MinSize are suppressed.
Default: 20
Suggested values: MinSize ∈ {0, 10, 20, 50, 100, 200, 500, 1000}
Value range: 0 ≤ MinSize ≤ 10000 (lin)
Minimum increment: 1
Recommended increment: 10

▷ **MinGray** (input_control) ..... real ⇝ *real*
Regions whose maximum absolute gray value is smaller than MinGray are suppressed.
Default: 5.0
Suggested values: MinGray ∈ {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 20.0}
Value range: 0 ≤ MinGray (lin)
Minimum increment: 1.0
Recommended increment: 10.0

▷ **Threshold** (input_control) ..... real ⇝ *real*
Regions that have a gray value smaller than Threshold (or larger than -Threshold) are suppressed.
Default: 2.0
Suggested values: Threshold ∈ {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 20.0}
Value range: 0 ≤ Threshold (lin)
Minimum increment: 1.0
Recommended increment: 10.0

Example

```
* Edge detection with the Laplace operator (and edge thinning)
diff_of_gauss(Image,Laplace,2.0,1.6)
* find "`positive"' and "`negative"' regions:
dual_threshold(Laplace,Region,20,2,1)
* The zero runnings are the complement to these image section:
complement(Region,ZeroCrossings)

* Simulation of dual_threshold
dual_threshold(Laplace,Result,MinS,MinG,Threshold)
threshold(Laplace,Tmp1,Threshold,999999)
connection(Tmp1,Tmp2)
select_shape(Tmp2,Tmp3,'area','and',MinS,999999)
select_gray(Laplace,Tmp3,Tmp4,'max','and',MinG,999999)
threshold(Laplace,Tmp5,-999999,-Threshold)
connection(Tmp5,Tmp6)
select_shape(Tmp6,Tmp7,'area','and',MinS,999999)
select_gray(Laplace,Tmp7,Tmp8,'min','and',-999999,-MinG)
concat_obj(Tmp4,Tmp8,Result)
```

Result

dual_threshold returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to

the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

min_max_gray, sobel_amp, binomial_filter, gauss_filter, reduce_domain, diff_of_gauss, sub_image, derivate_gauss, laplace_of_gauss, laplace, expand_region

Possible Successors

connection, dilation1, erosion1, opening, closing, rank_region, shape_trans, skeleton

Alternatives

threshold, dyn_threshold, check_difference

See also

connection, select_shape, select_gray

Module

Foundation

dyn_threshold (OrigImage, ThresholdImage : RegionDynThresh : Offset, LightDark :)

Segment an image using a local threshold.

dyn_threshold selects from the input image those regions in which the pixels fulfill a threshold condition. Let $g_o = g_$OrigImage, and $g_t = g_$ThresholdImage. Then the condition for LightDark = *'light'* is:

$$g_o \geq g_t + \texttt{Offset}$$

For LightDark = *'dark'* the condition is:

$$g_o \leq g_t - \texttt{Offset}$$

For LightDark = *'equal'* it is:

$$g_t - \texttt{Offset} \leq g_o \leq g_t + \texttt{Offset}$$

Finally, for LightDark = *'not_equal'* it is:

$$g_t - \texttt{Offset} > g_o \vee g_o > g_t + \texttt{Offset}$$

Typically, the threshold images are smoothed versions of the original image (e.g., by applying mean_image, binomial_filter, gauss_filter, etc.). Then the effect of dyn_threshold is similar to applying threshold to a highpass-filtered version of the original image (see highpass_image).

With dyn_threshold, contours of an object can be extracted, where the objects' size (diameter) is determined by the mask size of the lowpass filter and the amplitude of the objects' edges:

The larger the mask size is chosen, the larger the found regions become. As a rule of thumb, the mask size should be about twice the diameter of the objects to be extracted. It is important not to set the parameter Offset to zero because in this case too many small regions will be found (noise). Values between 5 and 40 are a useful choice. The larger Offset is chosen, the smaller the extracted regions become.

All points of the input image fulfilling the above condition are stored jointly in one region. If necessary, the connected components can be obtained by calling connection.

Attention

If `Offset` is chosen from -1 to 1 usually a very noisy region is generated, requiring large storage. If `Offset` is chosen too large (> 60, say) it may happen that no points fulfill the threshold condition (i.e., an empty region is returned). If `Offset` is chosen too small ($<$ -60, say) it may happen that all points fulfill the threshold condition (i.e., a full region is returned).

Parameters

- ▷ **`OrigImage`** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
  Input image.
- ▷ **`ThresholdImage`** (input_object) . . . . singlechannelimage(-array) ⇝ *object :* byte / int2 / uint2 / int4 / real
  Image containing the local thresholds.
- ▷ **`RegionDynThresh`** (output_object) . . . . . region(-array) ⇝ *object*
  Segmented regions.
- ▷ **`Offset`** (input_control) . . . . . number ⇝ *real* / integer
  Offset applied to ThresholdImage.
  Default: 5.0
  Suggested values: `Offset` ∈ {1.0, 3.0, 5.0, 7.0, 10.0, 20.0, 30.0}
  Value range: -255.0 ≤ `Offset` ≤ 255.0 (lin)
  Minimum increment: 0.01
  Recommended increment: 5
- ▷ **`LightDark`** (input_control) . . . . . string ⇝ *string*
  Extract light, dark or similar areas?
  Default: 'light'
  List of values: `LightDark` ∈ {'dark', 'light', 'equal', 'not_equal'}

Example

```
* Looking for regions with the diameter D
mean_image(Image,Mean,D*2+1,D*2+1)
dyn_threshold(Image,Mean,Seg,5,'light')
connection(Seg,Regions)
```

Complexity

Let A be the area of the input region. Then the runtime complexity is $O(A)$.

Result

`dyn_threshold` returns `2` (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

`mean_image`, `smooth_image`, `binomial_filter`, `gauss_filter`

Possible Successors

`connection`, `select_shape`, `reduce_domain`, `select_gray`, `rank_region`, `dilation1`, `opening`, `erosion1`

Alternatives

`check_difference`, `threshold`

See also

`highpass_image`, `sub_image`

Module

Foundation

fast_threshold (Image : Region : MinGray, MaxGray, MinSize :)

Fast thresholding of images using global thresholds.

fast_threshold selects the pixels from the input image whose gray values g fulfill the following condition:

$$\texttt{MinGray} \leq g \leq \texttt{MaxGray} \ .$$

To reduce the processing time, the selection is done in two steps: At first, all points lying on selected horizontal lines that are specified by their distance MinSize are processed. In the next step the neighborhood (size $(2 * MinSize + 1) \times (2 * MinSize + 1)$) of all previously selected points are processed.

Attention

On multi-core computers supporting the SSE2 instruction set, threshold is most likely faster than fast_threshold. fast_threshold may only be preferred to threshold if those features are not available, e.g., on embedded platforms.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2 / direction / cyclic / real
Input image.

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Segmented regions.

▷ **MinGray** (input_control) ..... number ⇝ *real* / integer
Lower threshold for the gray values.
Default: 128
Suggested values: MinGray ∈ {0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0}
Value range: 0.0 ≤ MinGray ≤ 255.0 (lin)
Minimum increment: 1
Recommended increment: 5.0

▷ **MaxGray** (input_control) ..... number ⇝ *real* / integer
Upper threshold for the gray values.
Default: 255.0
Suggested values: MaxGray ∈ {0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0}
Value range: 0.0 ≤ MaxGray ≤ 255.0 (lin)
Minimum increment: 1
Recommended increment: 5.0

▷ **MinSize** (input_control) ..... number ⇝ *integer*
Minimum size of objects to be extracted.
Default: 20
Suggested values: MinSize ∈ {5, 10, 15, 20, 25, 30, 40, 50, 60, 70, 100}
Value range: 2 ≤ MinSize ≤ 200 (lin)
Minimum increment: 1
Recommended increment: 2

Complexity

Let A be the area of the output region and $height$ the height of Image. Then the runtime complexity is $O(A + height/\texttt{MinSize})$.

Result

fast_threshold returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

histo_to_thresh, min_max_gray, sobel_amp, binomial_filter, gauss_filter, reduce_domain, fill_interlace

Possible Successors

connection, dilation1, erosion1, opening, closing, rank_region, shape_trans, skeleton

Alternatives

threshold, gen_grid_region, dilation_rectangle1, dyn_threshold

See also

class_2dim_sup, hysteresis_threshold

Module

Foundation

histo_to_thresh (: : Histogramm, Sigma : MinThresh, MaxThresh)

Determine gray value thresholds from a histogram.

histo_to_thresh determines gray value thresholds from a histogram for a segmentation of an image using threshold. The thresholds returned are 0, the maximum gray value in the histogram, and all minima extracted from the histogram. Before the thresholds are determined, the histogram is smoothed with a Gaussian smoothing function.

histo_to_thresh can process the absolute and relative histograms that are returned by gray_histo. Note, however, that here only byte images should be used, because otherwise the returned thresholds cannot easily be transformed to the thresholds for the actual image. For images of type uint2, the histograms should be computed with gray_histo_abs since this facilitates a simple transformation of the thresholds by simply multiplying the thresholds with the quantization selected in gray_histo_abs. For uint2 images, it is important to ensure that the quantization must be chosen in such a manner that the histogram still contains salient information. For example, a 640×480 image with 16 bits per pixel gray value resolution contains on average only $307200/65536 = 4.7$ entries per histogram bin, i.e., the histogram is too sparsely populated to derive any useful statistics from it. To be able to extract useful thresholds from such a histogram, Sigma would have to be set to an extremely large value, which would lead to very high run times and numerical problems. The quantization in gray_histo_abs should therefore normally be chosen such that the histogram contains a maximum of 1024 entries. Hence, for images with more than 10 bits per pixel, the quantization must be chosen greater than 1. The histogram returned by gray_histo_abs should furthermore be restricted to the parts that contain salient information. For example, for an image with 12 bits per pixel, the quantization should be set to 4. Only the first 1024 entries of the computed histogram (which contains 16384 entries in this example) should be passed to histo_to_thresh. Finally, MinThresh must be multiplied by 4 (i.e., the quantization), while MaxThresh must be multiplied by 4 and increased by 3 (i.e., the quantization minus 1).

Parameters

▷ **Histogramm** (input_control) ..... histogram-array ⇝ *integer / real*
Gray value histogram.

▷ **Sigma** (input_control) ..... number ⇝ *real*
Sigma for the Gaussian smoothing of the histogram.
Default: 2.0
Suggested values: Sigma ∈ {0.5, 1.0, 2.0, 3.0, 4.0, 5.0}
Value range: 0.5 ≤ Sigma ≤ 30.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.2

▷ **MinThresh** (output_control) ..... integer-array ⇝ *integer*
Minimum thresholds.

▷ **MaxThresh** (output_control) ..... integer-array ⇝ *integer*
Maximum thresholds.

Example

```
* Calculate thresholds from a byte image and threshold the image.
gray_histo (Image, Image, AbsoluteHisto, RelativeHisto)
histo_to_thresh (AbsoluteHisto, 4, MinThresh, MaxThresh)
threshold (Image, Region, MinThresh, MaxThresh)

* Calculate thresholds from a 12 bit uint2 image and threshold the image.
gray_histo_abs (Image, Image, 4, AbsoluteHisto)
AbsoluteHisto := AbsoluteHisto[0:1023]
histo_to_thresh (AbsoluteHisto, 16, MinThresh, MaxThresh)
MinThresh := MinThresh*4
MaxThresh := MaxThresh*4+3
threshold (Image, Region, MinThresh, MaxThresh)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gray_histo

Possible Successors

threshold

See also

auto_threshold, binary_threshold, char_threshold

Module

Foundation

local_threshold (Image : Region : Method, LightDark, GenParamName, GenParamValue :)

Segment an image using local thresholding.

local_threshold segments a single-channel image Image using the thresholding method given in Method and returns the segmented region in Region. Currently the operator offers only the Method *'adapted_std_deviation'*. This algorithm is a text binarization technique and provides good results for document images.

Adaptive Thresholding

By selecting Method = *'adapted_std_deviation'*, a locally adaptive thresholding based on local mean and standard deviation according to Sauvola (see the paper in References) is invoked. The algorithm is able to segment document images even if they are degraded, e.g., due to inhomogeneous illumination or noise. It enables text binarization on an inhomogeneous background by taking into account the local contrast.

For a segmentation of the dark foreground (see parameter LightDark), for a pixel at position (r, c), a local threshold $T(r, c)$ is calculated within a window of size *'mask_size'*×*'mask_size'* (see the generic parameter *'mask_size'*) as follows:

$$T(r,c) = \mu(r,c)\left(1 + k\left(\frac{\sigma(r,c)}{R} - 1\right)\right)$$

where $\mu(r, c)$ is the local mean value within the window and $\sigma(r, c)$ denotes the corresponding standard deviation. The parameter R (see *'range'*) is the assumed maximum value of the standard deviation (R= 128 for byte images) and k (see *'scale'*) a parameter that controls how much the threshold value $T(r, c)$ differs from the mean value $\mu(r, c)$. If there is high contrast in the neighborhood of a point (r, c) the standard deviation $\sigma(r, c)$ has a value close to R which yields a threshold value $T(r, c)$ close to the local mean $\mu(r, c)$. If the contrast is low, the local threshold is below the local mean value. For dark text on light background containing also darker regions, this lower threshold enables the segmentation of the text even in darker areas.

The parameter LightDark controls, whether light or dark structures are segmented.

- If LightDark = *'dark'*, dark structures on a light background are segmented. Every pixel $p(r, c)$ whose gray value is smaller than the calculated local threshold $T(r, c)$ is selected.
- If LightDark = *'light'*, light structures on a dark background are segmented. The result is essentially the same as if the image would have been inverted and then, LightDark was set to *'dark'*.

By setting GenParamName to one of the following values, additional parameters specific for the *'adapted_std_deviation'* method can be set with GenParamValue:

'mask_size': specifies the mask size, i.e., the size of the neighborhood in which the local threshold is calculated. The smaller the window size the thinner the segmented strokes. *'mask_size'* must be set to a value that is larger than the stroke width of the characters or structures to be segmented. If *'mask_size'* is even, the next larger odd value is used.

Suggested values: *15*, *21*, *31*.

Default: *15*.

'scale': sets the parameter k ($0 \leq k$), that controls how much the threshold value differs from the local mean value. Use smaller values for *'scale'* to also segment structures with a lower contrast to their background. Use larger values to suppress clutter.

Suggested values: *0.2*, *0.3*, *0.5*.

Default: *0.2*.

'range': sets the maximum assumed value of standard deviation R. This parameter should be adapted based on the expected gray value range. As a rule of thumb, the value for *'range'* can be set to $range = 0.5 \cdot (MaxGray - MinGray)$, where MinGray and MaxGray are the minimum and maximum gray values in the image, which can be determined with min_max_gray.

Suggested values: *128*, *32767.5*.

Default: *128* (for byte images), *32767.5* (for uint2 images).

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / uint2
Input Image.

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Segmented output region.

▷ **Method** (input_control) ..... string ⇝ *string*
Segmentation method.
Default: 'adapted_std_deviation'
List of values: Method ∈ {'adapted_std_deviation'}

▷ **LightDark** (input_control) ..... string ⇝ *string*
Extract foreground or background?
Default: 'dark'
List of values: LightDark ∈ {'dark', 'light'}

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
List of generic parameter names.
Default: []
List of values: GenParamName ∈ {'mask_size', 'scale', 'range'}

▷ **GenParamValue** (input_control) ..... attribute.value(-array) ⇝ *integer* / real
List of generic parameter values.
Default: []
Suggested values: GenParamValue ∈ {0.2, 15, 30, 128.0}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

- Automatically parallelized on internal data level.

Possible Successors

connection, select_shape, select_gray

Alternatives

auto_threshold, binary_threshold, char_threshold

See also

gray_histo, threshold

References

J. Sauvola, M. Pietikäinen, “Adaptive document image binarization", Pattern Recognition, 33, 225-236 (2000)

Module

Foundation

threshold (Image : Region : MinGray, MaxGray :)

Segment an image using global threshold.

threshold selects the pixels from the input image whose gray values g fulfill the following condition:

$$\texttt{MinGray} \leq g \leq \texttt{MaxGray} \ .$$

All points of an image fulfilling the condition are returned as one region. If more than one gray value interval is passed (tuples for MinGray and MaxGray), one separate region is returned for each interval. For vector field images, the threshold is not applied to gray values but to the lengths of the vectors. The parameters MinGray and MaxGray can be set to *'min'* or *'max'* in order to leave bottom and top limits, respectively, open.

Attention

The usage of MinGray and MaxGray can be affected by the input image type.

For images of type integer, floating point values in MinGray and MaxGray are truncated. For images of type real or vector field, floating point precision will be used for MinGray and MaxGray, if they are of type double.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / vector_field
Input image.

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Segmented region.

▷ **MinGray** (input_control) ..... number(-array) ⇝ *real* / integer / string
Lower threshold for the gray values or *'min'*.
Default: 128.0
Suggested values: MinGray ∈ {0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0, 'min'}

▷ **MaxGray** (input_control) ..... number(-array) ⇝ *real* / integer / string
Upper threshold for the gray values or *'max'*.
Default: 255.0
Suggested values: MaxGray ∈ {0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0, 'max'}
Restriction: MaxGray >= MinGray

Example

```
read_image(Image,'fabrik')
sobel_dir(Image,EdgeAmp,EdgeDir,'sum_abs',3)
threshold(EdgeAmp,Seg,50,255)
skeleton(Seg,Rand)
connection(Rand,Lines)
select_shape(Lines,Edges,'area','and',10,1000000)
```

Complexity

Let A be the area of the input region. Then the runtime complexity is $O(A)$.

Result

`threshold` returns 2 (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on internal data level.

Possible Predecessors

`histo_to_thresh`, `min_max_gray`, `sobel_amp`, `binomial_filter`, `gauss_filter`, `reduce_domain`, `fill_interlace`

Possible Successors

`connection`, `dilation1`, `erosion1`, `opening`, `closing`, `rank_region`, `shape_trans`, `skeleton`

Alternatives

`class_2dim_sup`, `hysteresis_threshold`, `dyn_threshold`, `binary_threshold`, `char_threshold`, `auto_threshold`, `dual_threshold`

See also

`zero_crossing`, `background_seg`, `regiongrowing`

Module

Foundation

`threshold_sub_pix` (Image : Border : Threshold :)

Extract level crossings from an image with subpixel accuracy.

`threshold_sub_pix` extracts the level crossings at the level `Threshold` of the input image `Image` with subpixel accuracy. The extracted level crossings are returned as XLD-contours in `Border`. In contrast to the operator `threshold`, `threshold_sub_pix` does not return regions, but the lines that separate regions with a gray value less than `Threshold` from regions with a gray value greater than `Threshold`.

For the extraction, the input image is regarded as a surface, in which the gray values are interpolated bilinearly between the centers of the individual pixels. Consistent with the surface thus defined, level crossing lines are extracted for each pixel and linked into topologically sound contours. This means that the level crossing contours are correctly split at junction points. If the image contains extended areas of constant gray value `Threshold`, only the border of such areas is returned as level crossings.

Parameters

▷ **`Image`** (input_object) ..... singlechannelimage ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
Input image.

▷ **`Border`** (output_object) ..... xld_cont-array ⇝ *object*
Extracted level crossings.

▷ **`Threshold`** (input_control) ..... number ⇝ *real* / integer
Threshold for the level crossings.
Default: 128
Suggested values: `Threshold` ∈ {0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0}

Example

```
read_image(Image,'fabrik')
threshold_sub_pix(Image,Border,35)
dev_display(Border)
```

Result

`threshold_sub_pix` usually returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`threshold`

See also

`zero_crossing_sub_pix`

Module

2D Metrology

var_threshold (Image : Region : MaskWidth, MaskHeight, StdDevScale, AbsThreshold, LightDark :)

Threshold an image by local mean and standard deviation analysis.

With `var_threshold`, it's possible to select the pixels of the input `Image` which

- have a high local standard deviation (for a positive `StdDevScale`), or a low local standard deviation (for a negative `StdDevScale`)

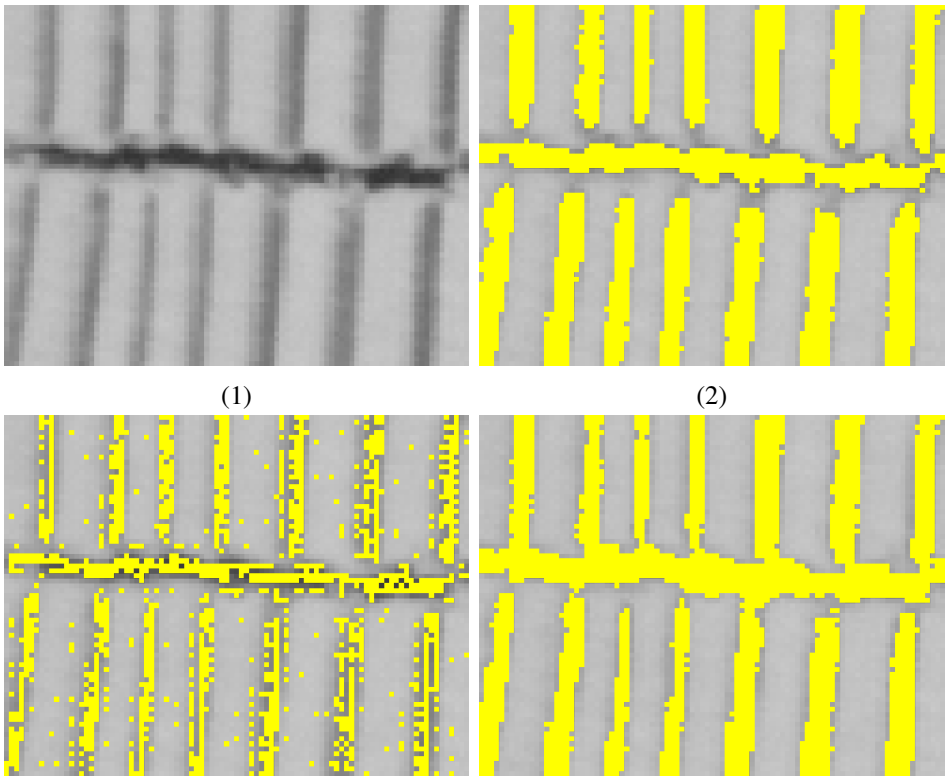
and

- are locally bright or dark, according to `LightDark`.

Thus, it is possible to segment regions on inhomogeneous, noisy, or unevenly illuminated backgrounds.

Hints for the input parameters

`MaskWidth`, `MaskHeight` The size of the filter mask defined by `MaskWidth` and `MaskHeight` determines the maximum size of the objects to be segmented. However, if the mask is chosen too large, objects that are very close might be merged.



(1) (2)

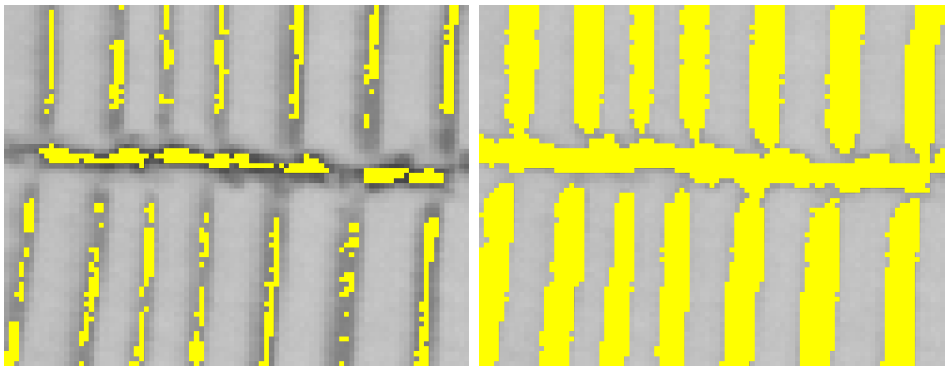
(3) (4)

(1) Original image; the goal is to count the vertical lines. (2) `MaskWidth` := *12*, `MaskHeight` := *12*, `StdDevScale` := *0.1*, all vertical lines are segmented correctly. (3) If the mask size is chosen too low (*3*), the desired regions are not selected properly. (4) If the mask size is too high (*40*), objects that are very close might be merged.

If `MaskWidth` or `MaskHeight` is even, the next larger odd value is used. Altogether, a value of *3* can be considered the minimum sensible value.

`StdDevScale` The local standard deviation is used as a measure of noise in the image. It can be scaled by `StdDevScale` to reflect the desired sensitivity. A higher value means that only pixels that are very different from their surrounding are selected.

For the parameter `StdDevScale` values between -1.0 and 1.0 are sensible choices, with 0.2 as a suggested value. If the parameter is too high or too low, an empty or full region may be returned.



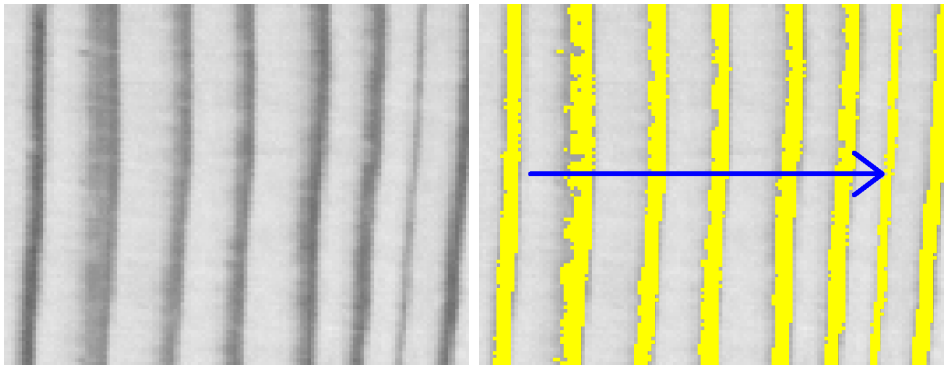
(1) (2)

(1) If `StdDevScale` is too high (*1.3*), the operator is 'picky'; only pixels that are very similar to their surrounding are selected. (2) If `StdDevScale` is too low (*-0.3*), too many pixels that are somewhat similar to their surrounding are incorrectly selected.

`AbsThreshold` In homogeneous areas of an image, the standard deviation is low; thus, the influence of single gray values is high. To reduce the sensitivity of the operator in homogeneous areas, it's possible to adjust `AbsThreshold`. Thus, small gray value changes in homogeneous surroundings can be ignored. Note that for negative values of `StdDevScale`, `AbsThreshold` should also be chosen negative.

LightDark *'light'* or *'dark'* returns all pixels that are lighter or darker than their surrounding, respectively. *'equal'* returns all pixels that are not selected by either option, i.e. the pixels that are relatively equal to their surrounding. *'not_equal'* returns the combined results of *'light'* and *'dark'*, i.e., all pixels that differ from their surrounding.

The calculation



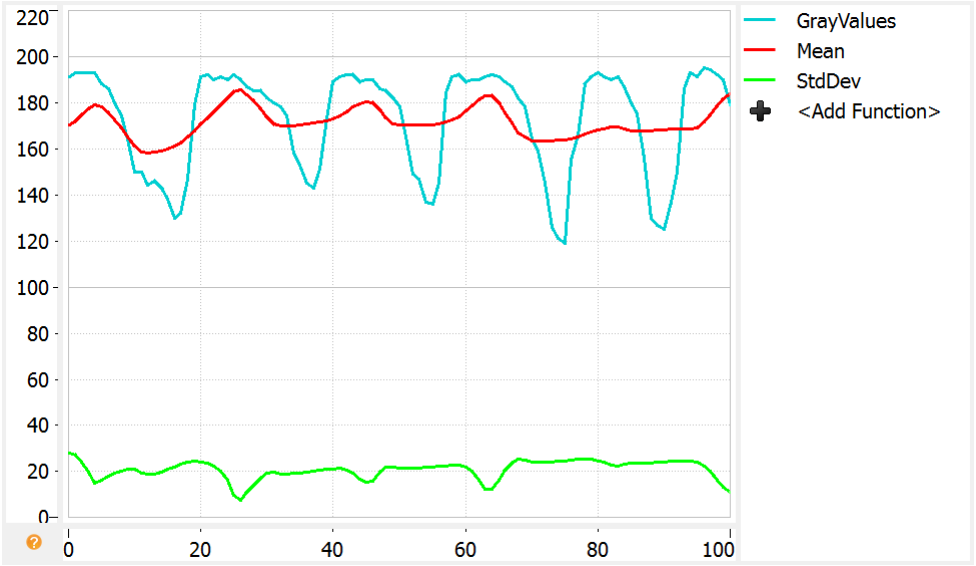
(1) (2)

(1) Initial image. (2) Thresholded image (StdDevScale := *0.6*, MaskWidth := *15*, MaskHeight := *15*, AbsThreshold := *10*). The following images visualize exemplary how the result along the blue arrow came to be.

var_threshold selects from the input image Image those regions Region in which the pixels fulfill a threshold condition. The threshold is calculated from the mean gray value and the standard deviation in a local mask of size MaskWidth x MaskHeight around each pixel (x, y).

Let

- $g(x, y)$ be the gray value at position (x, y) in the input Image,
- $m(x, y)$ the corresponding mean gray value, and
- $d(x, y)$ the corresponding standard deviation in the mask around that pixel.



The original gray values, the corresponding mean gray values, and the corresponding standard deviation in the mask around these pixels.

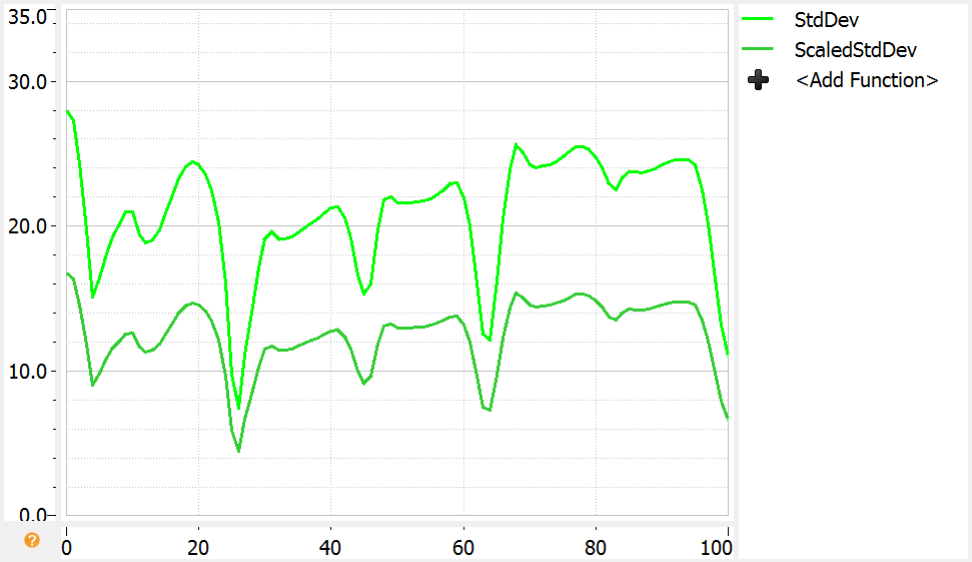
Then, the variable threshold $v(x, y)$ is defined as

$v(x, y) = max(\text{StdDevScale} * d(x, y), \text{AbsThreshold})$ for $\text{StdDevScale} \geq 0$

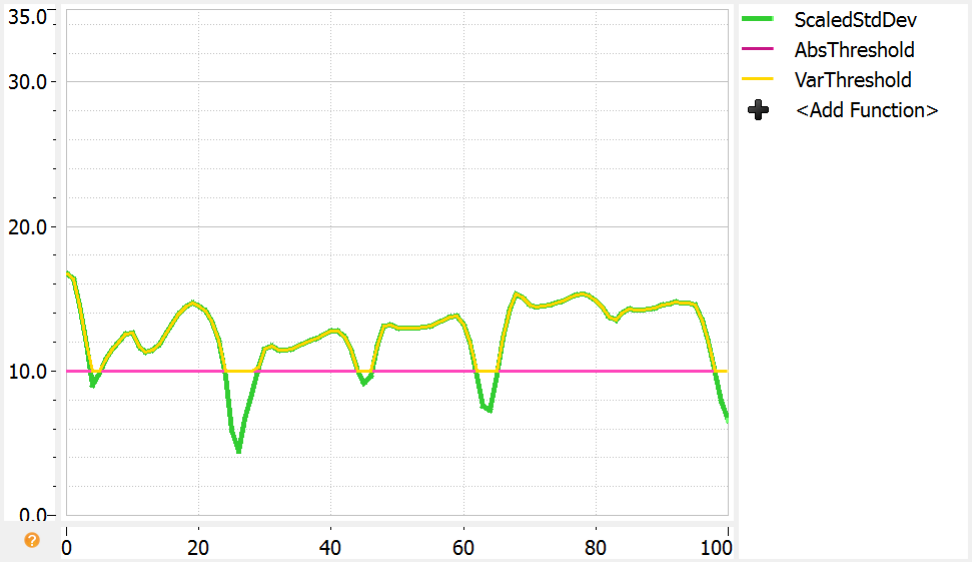
or

$v(x, y) = min(\text{StdDevScale} * d(x, y), \text{AbsThreshold})$ for $\text{StdDevScale} < 0$.

Interpretation: For a positive StdDevScale, each pixel is analyzed. It is determined whether the user-defined AbsThreshold or the scaled standard deviation is greater. The bigger value is chosen as variable threshold $v(x, y)$. For a negative StdDevScale, the corresponding smaller value is chosen.



The standard deviation can be scaled with StdDevScale.

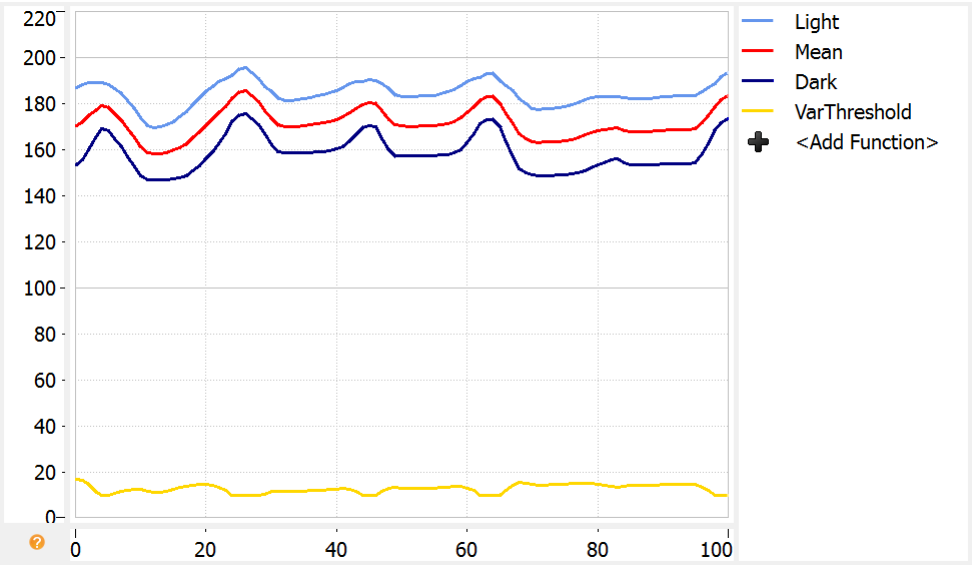


The variable threshold is chosen based on the scaled standard deviation and AbsThreshold.

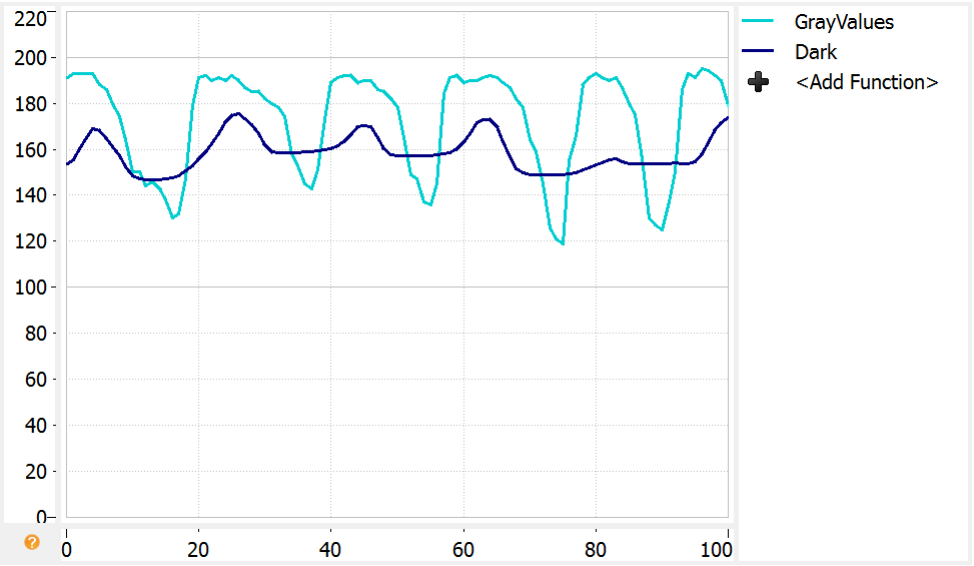
Which pixels are chosen based on the variable threshold is defined by the parameter LightDark:

- *'light'*:
  $g(x, y) \geq m(x, y) + v(x, y)$.
  Interpretation: If the pixel is brighter by $v(x, y)$ than its surrounding, it is selected.
- *'dark'*:
  $g(x, y) \leq m(x, y) - v(x, y)$.
  Interpretation: If the pixel is darker by $v(x, y)$ than its surrounding, it is selected.
  LightDark = *'equal'*:
  $m(x, y) - v(x, y) \leq g(x, y) \leq m(x, y) + v(x, y)$.
  Interpretation: Select exactly those pixels that are not selected by *'light'* and *'dark'*, i.e., the pixels that are relatively equal to their surrounding.

- *'not_equal'*:
  $m(x,y) - v(x,y) > g(x,y) \quad \vee \quad g(x,y) > m(x,y) + v(x,y)$.
  Interpretation: Select all pixels of *'light'* and *'dark'*, i.e., all pixels that differ by $v(x,y)$ from their surrounding.



'light' and *'dark'* are calculated based on the corresponding mean gray value and the variable threshold.



For example, if *'dark'* is selected, all pixels where the 'Dark' curve is above the 'GrayValues' curve would be selected.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **`Image`** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / int2 / int4 / uint2 / real
Input image.

▷ **`Region`** (output_object) . . . . . region(-array) ⇝ *object*
Segmented regions.

▷ **`MaskWidth`** (input_control) . . . . . extent.x ⇝ *integer*
Mask width for mean and deviation calculation.
Default: 15
Suggested values: `MaskWidth` ∈ {9, 11, 13, 15}
Restriction: MaskWidth >= 1

▷ **MaskHeight** (input_control) . . . . . extent.y ⇝ *integer*
Mask height for mean and deviation calculation.
Default: 15
Suggested values: MaskHeight ∈ {9, 11, 13, 15}
Restriction: MaskHeight >= 1

▷ **StdDevScale** (input_control) . . . . . number ⇝ *real* / integer
Factor for the standard deviation of the gray values.
Default: 0.2
Suggested values: StdDevScale ∈ {-0.2, -0.1, 0.1, 0.2}

▷ **AbsThreshold** (input_control) . . . . . number ⇝ *real* / integer
Minimum gray value difference from the mean.
Default: 2
Suggested values: AbsThreshold ∈ {-2, -1, 0, 1, 2}

▷ **LightDark** (input_control) . . . . . string ⇝ *string*
Threshold type.
Default: 'dark'
List of values: LightDark ∈ {'dark', 'light', 'equal', 'not_equal'}

Complexity

Let A be the area of the input region, then the runtime is $O(A)$.

Result

var_threshold returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Alternatives

dyn_threshold, threshold

References

W.Niblack, "An Introduction to Digital Image Processing", Page 115-116, Englewood Cliffs, N.J., Prentice Hall, 1986

Module

Foundation

zero_crossing (Image : RegionCrossing : :)

Extract zero crossings from an image.

zero_crossing returns the zero crossings of the input image as a region. A pixel is accepted as a zero crossing if its gray value (in Image) is zero, or if at least one of its neighbors of the 4-neighborhood has a different sign.

This operator is intended to be used after edge operators returning the second derivative of the image (e.g., laplace_of_gauss), which were possibly followed by a smoothing operator. In this case, the zero crossings are (candidates for) edges.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* int1 / int2 / int4 / real
Input image.

▷ **RegionCrossing** (output_object) . . . . . region(-array) ⇝ *object*
Zero crossings.

Result

`zero_crossing` usually returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.
- Automatically parallelized on domain level.

Possible Predecessors

`laplace`, `laplace_of_gauss`, `derivate_gauss`

Possible Successors

`connection`, `skeleton`, `boundary`, `select_shape`, `fill_up`

Alternatives

`threshold`, `dual_threshold`, `zero_crossing_sub_pix`

Module

Foundation

zero_crossing_sub_pix (Image : ZeroCrossings : :)

Extract zero crossings from an image with subpixel accuracy.

`zero_crossing_sub_pix` extracts the zero crossings of the input image `Image` with subpixel accuracy. The extracted zero crossings are returned as XLD-contours in `ZeroCrossings`. Thus, `zero_crossing_sub_pix` can be used as a sub-pixel precise edge extractor if the input image is a Laplace-filtered image (see `laplace`, `laplace_of_gauss`, `derivate_gauss`).

For the extraction, the input image is regarded as a surface, in which the gray values are interpolated bilinearly between the centers of the individual pixels. Consistent with the surface thus defined, zero crossing lines are extracted for each pixel and linked into topologically sound contours. This means that the zero crossing contours are correctly split at junction points. If the image contains extended areas of constant gray value 0, only the border of such areas is returned as zero crossings.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* int1 / int2 / int4 / real
Input image.

▷ **ZeroCrossings** (output_object) ..... xld_cont-array ⇝ *object*
Extracted zero crossings.

Example

```
* Detection zero crossings of the Laplacian-of-Gaussian
* of an aerial image
read_image(Image,'mreut')
derivate_gauss(Image,Laplace,3,'laplace')
zero_crossing_sub_pix(Laplace,ZeroCrossings)
dev_display(ZeroCrossings)

* Detection of edges, i.e, zero crossings of the Laplacian-of-Gaussian
* that have a large gradient magnitude, in an aerial image
read_image(Image,'mreut')
Sigma := 1.5
* Compensate the threshold for the fact that derivate_gauss(...,'gradient')
* calculates a Gaussian-smoothed gradient, in which the edge amplitudes
* are too small because of the Gaussian smoothing, to correspond to a true
* edge amplitude of 20.
```

```
Threshold := 20/(Sigma*sqrt(2*3.1415926))
derivate_gauss(Image,Gradient,Sigma,'gradient')
threshold(Gradient,Region,Threshold,255)
reduce_domain(Image,Region,ImageReduced)
derivate_gauss(ImageReduced,Laplace,Sigma,'laplace')
zero_crossing_sub_pix(Laplace,Edges)
dev_display(Edges)
```

Result

`zero_crossing_sub_pix` usually returns the value 2 (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`laplace`, `laplace_of_gauss`, `diff_of_gauss`, `derivate_gauss`

Alternatives

`zero_crossing`

See also

`threshold_sub_pix`

Module

2D Metrology

24.6 Topography

`critical_points_sub_pix` (`Image` : : `Filter`, `Sigma`, `Threshold` : `RowMin`, `ColumnMin`, `RowMax`, `ColumnMax`, `RowSaddle`, `ColumnSaddle`)

Subpixel precise detection of critical points in an image.

`critical_points_sub_pix` extracts critical points, i.e., local maxima, local minima, and saddle points, from the image `Image` with subpixel precision. To do so, in each point the input image is approximated by a quadratic polynomial in x and y and subsequently the polynomial is examined for extremal values and saddle points. The partial derivatives, which are necessary for setting up the polynomial, are calculated either with various Gaussian derivatives or using the facet model, depending on `Filter`. In the first case, `Sigma` determines the size of the Gaussian kernels, while in the second case, before being processed the input image is smoothed by a Gaussian whose size is determined by `Sigma`. Therefore, *'facet'* results in a faster extraction at the expense of slightly less accurate results. A point is accepted to be a critical point if the absolute values of both eigenvalues of the Hessian matrix are greater than `Threshold`. The eigenvalues correspond to the curvature of the gray value surface. If both eigenvalues are negative, the point is a local maximum, if both are positive, a local minimum, and if they have different signs, a saddle point.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **`Image`** (input_object) ..... singlechannelimage ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
Input image.

▷ **`Filter`** (input_control) ..... string ⇝ *string*
Method for the calculation of the partial derivatives.
Default: 'facet'
List of values: `Filter` ∈ {'facet', 'gauss'}

▷ **Sigma** (input_control) ..... real ⇝ *real*
Sigma of the Gaussian. If Filter is 'facet', Sigma may be 0.0 to avoid the smoothing of the input image.
Suggested values: Sigma ∈ {0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0}
Restriction: Sigma >= 0.0

▷ **Threshold** (input_control) ..... real ⇝ *real*
Minimum absolute value of the eigenvalues of the Hessian matrix.
Default: 5.0
Suggested values: Threshold ∈ {2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0}
Restriction: Threshold >= 0.0

▷ **RowMin** (output_control) ..... point.y-array ⇝ *real*
Row coordinates of the detected minima.

▷ **ColumnMin** (output_control) ..... point.x-array ⇝ *real*
Column coordinates of the detected minima.

▷ **RowMax** (output_control) ..... point.y-array ⇝ *real*
Row coordinates of the detected maxima.

▷ **ColumnMax** (output_control) ..... point.x-array ⇝ *real*
Column coordinates of the detected maxima.

▷ **RowSaddle** (output_control) ..... point.y-array ⇝ *real*
Row coordinates of the detected saddle points.

▷ **ColumnSaddle** (output_control) ..... point.x-array ⇝ *real*
Column coordinates of the detected saddle points.

Result

critical_points_sub_pix returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during the execution. If the input is empty the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

gen_cross_contour_xld, disp_cross

Alternatives

local_min_sub_pix, local_max_sub_pix, saddle_points_sub_pix

See also

local_min, local_max, plateaus, plateaus_center, lowlands, lowlands_center

Module

Foundation

local_max (Image : LocalMaxima : :)

Detect all local maxima in an image.

local_max extracts all points from Image having a gray value larger than the gray value of all its neighbors and returns them in LocalMaxima. The neighborhood used can be set by set_system(::'neighborhood',<4/8>).

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real
Input image.

▷ **LocalMaxima** (output_object) ..... region(-array) ⇝ *object*
Extracted local maxima as a region.
Number of elements: LocalMaxima == Image

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  using namespace Halcon;
  if (argc != 2)
  {
    cout << "Usage : " << argv[0] << " <name of image>" << endl;
    return (-1);
  }

  HImage   image (argv[1]);
  HWindow  win;

  image.Display (win);

  HImage       cres = image.CornerResponse (5, 0.04);
  HRegionArray maxi = cres.LocalMax ();

  win.SetColored (12);
  maxi.Display (win);
  win.Click ();

  return (0);
}
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image

Possible Successors

get_region_points, connection

Alternatives

nonmax_suppression_amp, plateaus, plateaus_center

See also

monotony, topographic_sketch, corner_response, texture_laws

Module

Foundation

local_max_sub_pix (Image : : Filter, Sigma, Threshold : Row, Column)

Subpixel precise detection of local maxima in an image.

local_max_sub_pix extracts local maxima from the image Image with subpixel precision. To do so, in each point the input image is approximated by a quadratic polynomial in x and y and subsequently the polynomial is examined for local maxima. The partial derivatives, which are necessary for setting up the polynomial, are calculated either with various Gaussian derivatives or using the facet model, depending on Filter. In the first case, Sigma determines the size of the Gaussian kernels, while in the second case, before being processed the input image is smoothed by a Gaussian whose size is determined by Sigma. Therefore, *'facet'* results in a faster extraction at the expense of slightly less accurate results. A point is accepted to be a local maximum if both eigenvalues of the Hessian matrix are smaller than -Threshold. The eigenvalues correspond to the curvature of the gray value surface.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
  Input image.
- ▷ **Filter** (input_control) ..... string ⇝ *string*
  Method for the calculation of the partial derivatives.
  Default: 'facet'
  List of values: Filter ∈ {'facet', 'gauss'}
- ▷ **Sigma** (input_control) ..... real ⇝ *real*
  Sigma of the Gaussian. If Filter is 'facet', Sigma may be 0.0 to avoid the smoothing of the input image.
  Suggested values: Sigma ∈ {0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0}
  Restriction: Sigma >= 0.0
- ▷ **Threshold** (input_control) ..... real ⇝ *real*
  Minimum absolute value of the eigenvalues of the Hessian matrix.
  Default: 5.0
  Suggested values: Threshold ∈ {2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0}
  Restriction: Threshold >= 0.0
- ▷ **Row** (output_control) ..... point.y-array ⇝ *real*
  Row coordinates of the detected maxima.
- ▷ **Column** (output_control) ..... point.x-array ⇝ *real*
  Column coordinates of the detected maxima.

Result

local_max_sub_pix returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during the execution. If the input is empty the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

gen_cross_contour_xld, disp_cross

Alternatives

critical_points_sub_pix, local_min_sub_pix, saddle_points_sub_pix

See also

local_max, plateaus, plateaus_center

Module

Foundation

local_min (Image : LocalMinima : :)

Detect all local minima in an image.

local_min extracts all points from Image having a gray value smaller than the gray value of all its neighbors and returns them in LocalMinima. The neighborhood used can be set by set_system('neighborhood',<4/8>).

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real
Image to be processed.

▷ **LocalMinima** (output_object) ..... region(-array) ⇝ *object*
Extracted local minima as regions.
Number of elements: LocalMinima == Image

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  if (argc != 2)
  {
    cout << "Usage : " << argv[0] << " <name of image>" << endl;
    return (-1);
  }

  HImage   image (argv[1]);
  HWindow  win;

  image.Display (win);

  HImage       cres = image.CornerResponse (5, 0.04);
  HRegionArray mins = cres.LocalMin ();

  win.SetColored (12);
  mins.Display (win);
  win.Click ();

  return (0);
}
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image

Possible Successors

get_region_points, connection

Alternatives

gray_skeleton, lowlands, lowlands_center

See also

monotony, topographic_sketch, corner_response, texture_laws

Module

Foundation

local_min_sub_pix (Image : : Filter, Sigma, Threshold : Row, Column)

Subpixel precise detection of local minima in an image.

local_min_sub_pix extracts local minima from the image Image with subpixel precision. To do so, in each point the input image is approximated by a quadratic polynomial in x and y and subsequently the polynomial is examined for local minima. The partial derivatives, which are necessary for setting up the polynomial, are calculated either with various Gaussian derivatives or using the facet model, depending on Filter. In the first case, Sigma determines the size of the Gaussian kernels, while in the second case, before being processed the input image is smoothed by a Gaussian whose size is determined by Sigma. Therefore, *'facet'* results in a faster extraction at the expense of slightly less accurate results. A point is accepted to be a local minimum if both eigenvalues of the Hessian matrix are greater than Threshold. The eigenvalues correspond to the curvature of the gray value surface.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
  Input image.
- ▷ **Filter** (input_control) ..... string ⇝ *string*
  Method for the calculation of the partial derivatives.
  Default: 'facet'
  List of values: Filter ∈ {'facet', 'gauss'}
- ▷ **Sigma** (input_control) ..... real ⇝ *real*
  Sigma of the Gaussian. If Filter is 'facet', Sigma may be 0.0 to avoid the smoothing of the input image.
  Suggested values: Sigma ∈ {0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0}
  Restriction: Sigma >= 0.0
- ▷ **Threshold** (input_control) ..... real ⇝ *real*
  Minimum absolute value of the eigenvalues of the Hessian matrix.
  Default: 5.0
  Suggested values: Threshold ∈ {2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0}
  Restriction: Threshold >= 0.0
- ▷ **Row** (output_control) ..... point.y-array ⇝ *real*
  Row coordinates of the detected minima.
- ▷ **Column** (output_control) ..... point.x-array ⇝ *real*
  Column coordinates of the detected minima.

Result

local_min_sub_pix returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during the execution. If the input is empty the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

gen_cross_contour_xld, disp_cross

Alternatives

critical_points_sub_pix, local_max_sub_pix, saddle_points_sub_pix

See also

local_min, lowlands, lowlands_center

Module

Foundation

lowlands (Image : Lowlands : :)

Detect all gray value lowlands.

lowlands extracts all points from Image with a gray value less or equal to the gray value of its neighbors (8-neighborhood) and returns them in Lowlands. Each lowland is returned as a separate region.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real
Image to be processed.

▷ **Lowlands** (output_object) ..... region-array ⇝ *object*
Extracted lowlands as regions (one region for each lowland).

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  if (argc != 2)
  {
    cout << "Usage : " << argv[0] << " <name of image>" << endl;
    return (-1);
  }

  HImage   image (argv[1]);
  HWindow  win;

  image.Display (win);

  HImage       cres = image.CornerResponse (5, 0.04);
  HRegionArray mins = cres.Lowlands ();

  win.SetColored (12);
  mins.Display (win);
  win.Click ();

  return (0);
}
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image

Possible Successors

area_center, get_region_points, select_shape

Alternatives

lowlands_center, gray_skeleton, local_min

See also

monotony, topographic_sketch, corner_response, texture_laws

Module

Foundation

lowlands_center (Image : Lowlands : :)

Detect the centers of all gray value lowlands.

lowlands_center extracts all points from Image with a gray value less or equal to the gray value of its neighbors (8-neighborhood) and returns them in Lowlands. If more than one of these points are connected (lowland), their center of gravity is returned. Each lowland is returned as a separate region.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Image to be processed.

▷ **Lowlands** (output_object) ..... region-array ⇝ *object*

Centers of gravity of the extracted lowlands as regions (one region for each lowland).

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  if (argc != 2)
  {
    cout << "Usage : " << argv[0] << " <name of image>" << endl;
    return (-1);
  }

  HImage   image (argv[1]);
  HWindow  win;

  image.Display (win);

  HImage       cres = image.CornerResponse (5, 0.04);
  HRegionArray mins = cres.LowlandsCenter ();

  win.SetColored (12);
  mins.Display (win);
  win.Click ();

  return (0);
}
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image

Possible Successors

area_center, get_region_points, select_shape

Alternatives

lowlands, gray_skeleton, local_min

See also

monotony, topographic_sketch, corner_response, texture_laws

Module

Foundation

plateaus (Image : Plateaus : :)

Detect all gray value plateaus.

plateaus extracts all points from Image with a gray value greater or equal to the gray value of its neighbors (8-neighborhood) and returns them in Plateaus. Each maximum is returned as a separate region.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real
Input image.

▷ **Plateaus** (output_object) ..... region-array ⇝ *object*
Extracted plateaus as regions (one region for each plateau).

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  if (argc != 2)
  {
    cout << "Usage : " << argv[0] << " <name of image>" << endl;
    return (-1);
  }

  HImage   image (argv[1]);
  HWindow  win;

  image.Display (win);

  HImage       cres = image.CornerResponse (5, 0.04);
  HRegionArray maxi = cres.Plateaus ();
```

```
  win.SetColored (12);
  maxi.Display (win);
  win.Click ();

  return (0);
}
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image

Possible Successors

area_center, get_region_points, select_shape

Alternatives

plateaus_center, nonmax_suppression_amp, local_max

See also

monotony, topographic_sketch, corner_response, texture_laws

Module

Foundation

plateaus_center (Image : Plateaus : :)

Detect the centers of all gray value plateaus.

plateaus_center extracts all points from Image with a gray value greater or equal to the gray value of its neighbors (8-neighborhood) and returns them in Plateaus. If more than one of these points are connected (plateau), their center of gravity is returned. Each plateau center is returned as a separate region.

Parameters

▷ **Image** (input_object) ..... singlechannelimage(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real

Input image.

▷ **Plateaus** (output_object) ..... region-array ⇝ *object*

Centers of gravity of the extracted plateaus as regions (one region for each plateau).

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;

int main (int argc, char *argv[])
{
  if (argc != 2)
  {
    cout << "Usage : " << argv[0] << " <name of image>" << endl;
    return (-1);
  }
```

```
  HImage   image (argv[1]);
  HWindow  win;

  image.Display (win);

  HImage       cres = image.CornerResponse (5, 0.04);
  HRegionArray maxi = cres.PlateausCenter ();

  win.SetColored (12);
  maxi.Display (win);
  win.Click ();

  return (0);
}
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image

Possible Successors

area_center, get_region_points, select_shape

Alternatives

plateaus, nonmax_suppression_amp, local_max

See also

monotony, topographic_sketch, corner_response, texture_laws

Module

Foundation

pouring (Image : Regions : Mode, MinGray, MaxGray :)

Segment an image by "pouring water" over it.

pouring regards the input image as a "mountain range." Larger gray values correspond to mountain peaks, while smaller gray values correspond to valley bottoms. pouring segments the input image in several steps. First, the local maxima are extracted, i.e., pixels which either alone or in the form of an extended plateau have larger gray values than their immediate neighbors (in 4-neighborhood). In the next step, the maxima thus found are the starting points for an expansion until "valley bottoms" are reached. The expansion is done as long as there are chains of pixels in which the gray value becomes smaller (like water running downhill from the maxima in all directions). Again, the 4-neighborhood is used, but with a weaker condition (smaller or equal). This means that points at valley bottoms may belong to more than one maximum. These areas are at first not assigned to a region, but rather are split among all competing segments in the last step. The split is done by a uniform expansion of all involved segments, until all ambiguous pixels were assigned. The parameter Mode determines which steps are executed. The following values are possible:

'all' This is the normal mode of operation. All steps of the segmentation are performed. The regions are assigned to maxima, and overlapping regions are split.

'maxima' The segmentation only extracts the local maxima of the input image. No corresponding regions are extracted.

'regions' The segmentation extracts the local maxima of the input image and the corresponding regions, which are uniquely determined. Areas that were assigned to more than one maximum are not split.

In order to prevent the algorithm from splitting a uniform background that is different from the rest of the image, the parameters MinGray and MaxGray determine gray value thresholds for regions in the image that should be regarded as background. All parts of the image having a gray value smaller than MinGray or larger than MaxGray are disregarded for the extraction of the maxima as well as for the assignment of regions. For a complete segmentation of the image, MinGray = 0 and MaxGray = 255 should be selected. MinGray < MaxGray must be observed.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object : byte*
  Input image.
- ▷ **Regions** (output_object) ..... region-array ⇝ *object*
  Segmented regions.
- ▷ **Mode** (input_control) ..... string ⇝ *string*
  Mode of operation.
  Default: 'all'
  List of values: Mode ∈ {'all', 'maxima', 'regions'}
- ▷ **MinGray** (input_control) ..... integer ⇝ *integer*
  All gray values smaller than this threshold are disregarded.
  Default: 0
  Suggested values: MinGray ∈ {0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110}
  Value range: 0 ≤ MinGray ≤ 254 (lin)
  Minimum increment: 1
  Recommended increment: 10
- ▷ **MaxGray** (input_control) ..... integer ⇝ *integer*
  All gray values larger than this threshold are disregarded.
  Default: 255
  Suggested values: MaxGray ∈ {100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 255}
  Value range: 1 ≤ MaxGray ≤ 255 (lin)
  Minimum increment: 1
  Recommended increment: 10
  Restriction: MaxGray > MinGray

Example

```
* Segment a filtered image
read_image(Image,'particle')
mean_image(Image,Mean,11,11)
pouring(Mean,Seg,'all',0,255)
dev_display(Mean)
dev_set_colored(12)
dev_display(Seg)

* Segment an image while masking the dark background
read_image(Image,'particle')
mean_image(Image,ImageMean,15,15)
pouring(Mean,Seg,'all',90,255)
dev_display(Mean)
dev_set_colored(12)
dev_display(Seg)
```

Complexity

Let N be the number of pixels in the input image and M be the number of found segments, where the enclosing rectangle of the segment i contains m_i pixels. Furthermore, let K_i be the number of chords in segment i. Then the runtime complexity is

$$O(3 * N + sum_M(3 * m_i) + sum_M(K_i)) \ .$$

Result

pouring usually returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: mutually exclusive (runs in parallel with other non-exclusive operators, but not with itself).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image, mean_image

Alternatives

watersheds, local_max, watersheds_threshold, watersheds_marker

See also

histo_2dim, expand_region, expand_gray, expand_gray_ref

Module

Foundation

saddle_points_sub_pix (Image : : Filter, Sigma, Threshold : Row, Column)

Subpixel precise detection of saddle points in an image.

saddle_points_sub_pix extracts saddle points from the image Image with subpixel precision, i.e., points where along one direction the image intensity is minimal while at the same time along a different direction the image intensity is maximal. To do so, in each point the input image is approximated by a quadratic polynomial in x and y and subsequently the polynomial is examined for saddle points. The partial derivatives, which are necessary for setting up the polynomial, are calculated either with various Gaussian derivatives or using the facet model, depending on Filter. In the first case, Sigma determines the size of the Gaussian kernels, while in the second case, before being processed the input image is smoothed by a Gaussian whose size is determined by Sigma. Therefore, *'facet'* results in a faster extraction at the expense of slightly less accurate results. A point is accepted to be a saddle point if the absolute values of both eigenvalues of the Hessian matrix are greater than Threshold but their signs differ. The eigenvalues correspond to the curvature of the gray value surface.

saddle_points_sub_pix is especially useful for the detection of corners, where fields of different intensity join together like the black and white fields of a chess board. Their high contrast and shape facilitate the location and the determination of the precise position of such corners.

Attention

Note that filter operators may return unexpected results if an image with a reduced domain is used as input. Please refer to the chapter Filters.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / int1 / int2 / uint2 / int4 / real
Input image.

▷ **Filter** (input_control) ..... string ⇝ *string*
Method for the calculation of the partial derivatives.
Default: 'facet'
List of values: Filter ∈ {'facet', 'gauss'}

▷ **Sigma** (input_control) ..... real ⇝ *real*
Sigma of the Gaussian. If Filter is 'facet', Sigma may be 0.0 to avoid the smoothing of the input image.
Suggested values: Sigma ∈ {0.7, 0.8, 0.9, 1.0, 1.2, 1.5, 2.0, 3.0}
Restriction: Sigma >= 0.0

▷ **Threshold** (input_control) ..... real ⇝ *real*
Minimum absolute value of the eigenvalues of the Hessian matrix.
Default: 5.0
Suggested values: Threshold ∈ {2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0}
Restriction: Threshold >= 0.0

▷ **Row** (output_control) ..... point.y-array ⇝ *real*
Row coordinates of the detected saddle points.

▷ **Column** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinates of the detected saddle points.

Result

`saddle_points_sub_pix` returns `2` (`H_MSG_TRUE`) if all parameters are correct and no error occurs during the execution. If the input is empty the behavior can be set via `set_system ('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Successors

`gen_cross_contour_xld`, `disp_cross`

Alternatives

`critical_points_sub_pix`, `local_min_sub_pix`, `local_max_sub_pix`

Module

Foundation

watersheds (Image : Basins, Watersheds : :)

Extract watersheds and basins from an image.

`watersheds` segments an image based on the topology of the gray values. The image is interpreted as a "mountain range." Higher gray values correspond to "mountains," while lower gray values correspond to "valleys." In the resulting mountain range watersheds are extracted. These correspond to the bright ridges between dark basins. On output, the parameter `Basins` contains these basins, while `Watersheds` contains the watersheds, which are at most one pixel wide. `Watersheds` always is a single region per input image, while `Basins` contains a separate region for each basin.

It is advisable to apply a smoothing operator (e.g., `binomial_filter` or `gauss_filter`) to the input image before calling `watersheds` in order to reduce the number of output regions. A more sophisticated way to reduce the number of output regions is to merge neighboring basins based on a threshold criterion by using `watersheds_threshold` instead (for more details please refer to the documentation of `watersheds_threshold`).

Attention

If the image contains many fine structures or is noisy, many output regions result, and thus the runtime increases considerably.

Parameters

▷ **Image** (input_object) . . . . . singlechannelimage(-array) ⇝ *object :* byte / uint2 / real
Input image.

▷ **Basins** (output_object) . . . . . region-array ⇝ *object*
Segmented basins.

▷ **Watersheds** (output_object) . . . . . region(-array) ⇝ *object*
Watersheds between the basins.

Example

```
#include "HIOStream.h"
#if !defined(USE_IOSTREAM_H)
using namespace std;
#endif
#include "HalconCpp.h"
using namespace Halcon;
```

```
int main (int argc, char *argv[])
{
  HImage   image (argv[1]), gauss;
  HWindow  win;

  cout << "Gauss of original " << endl;
  gauss  = image.GaussImage (9);
  image.Display (win);

  HRegion        watersheds;
  HRegionArray   basins = gauss.Watersheds (&watersheds);

  win.SetColored (12);
  basins.Display (win);
  win.Click ();

  return (0);
}
```

Result

`watersheds` always returns `2` (`H_MSG_TRUE`). The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with `set_system`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`binomial_filter`, `gauss_filter`, `smooth_image`, `invert_image`

Possible Successors

`expand_region`, `select_shape`, `reduce_domain`, `opening`

Alternatives

`watersheds_threshold`, `pouring`, `watersheds_marker`

References

L. Vincent, P. Soille: "Watersheds in Digital Space: An Efficient Algorithm Based on Immersion Simulations"; IEEE Transactions on Pattern Analysis and Machine Intelligence; vol. 13, no. 6; pp. 583-598; 1991.

Module

Foundation

watersheds_marker (Image, Markers : Basins : :)

Extract watersheds and combine basins based on markers.

`watersheds_marker` segments an image based on the topology of the gray values and returns the resulting regions in `Basins`. The image is interpreted as a "mountain range". Higher gray values correspond to "mountains", while lower gray values correspond to "valleys". The image, interpreted in this particular way, is flooded, starting with the regions defined in `Markers`, until the entire image is filled. Thus, each pixel is assigned to the marker to which it has the path with the lowest maximum height. If this is the case for multiple markers, the pixel is assigned to either of them.

The resulting basins, one per input region in `Markers`, are returned in `Basins`. Note that some of those regions might be empty. This can happen if, for example, two marker regions are in the same watershed basin. If a pixel is contained in multiple marker regions, only the last marker region is used as seed for that pixel. It is advised that the marker regions do not overlap.

The flooding is performed only for the domain of Image. Parts of the marker regions outside the domain of Image are ignored.

It is advisable to apply a smoothing operator (e.g., binomial_filter or gauss_filter) to the input image before calling watersheds_marker. To segment an image, it is also often reasonable to run an edge extractor (such as edges_image or edges_color) on the image, and pass the resulting amplitude image to watersheds_marker.

Attention

If the image contains many fine structures or is noisy, many watershed regions need to be processed internally, and thus the runtime increases considerably.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
  Input image.
- ▷ **Markers** (input_object) ..... region-array ⇝ *object*
  Initial markers from which to flood the image.
- ▷ **Basins** (output_object) ..... region-array ⇝ *object*
  Basins for all markers.

Example

```
read_image (Image, 'printer_chip/printer_chip_01')
edges_image (Image, ImaAmp, ImaDir, 'canny', 1, 'nms', 20, 40)

* Compute background marker
full_domain (ImaAmp, ImageFull)
erosion_circle (Image, RegionErosion, 2.5)
difference (Image, RegionErosion, Boundary)

* Compute foreground markers
threshold (Image, Region, 200, 255)
erosion_circle (Region, RegionErosion1, 7.5)
connection (RegionErosion1, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and', 1500, 99999)

* Apply marker-based watersheds
concat_obj (Boundary, SelectedRegions, MarkerRegions)
watersheds_marker (ImageFull, MarkerRegions, Basins)

* Display results
dev_display (Image)
dev_set_color (['#FF000055', '#00FF0055', '#0000FF55', '#FF00FF55', \
               '#FFFF0055', '#00FFFF55', '#80FF0055', '#0080FF55'])
dev_display (MarkerRegions)
dev_set_draw ('fill')
dev_display (Basins)
```

Result

If all input parameters are valid, watersheds_marker returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image, invert_image, edges_image

Possible Successors

expand_region, select_shape, reduce_domain, opening

Alternatives

watersheds, pouring, watersheds_threshold

Module

Foundation

watersheds_threshold (Image : Basins : Threshold :)

Extract watershed basins from an image using a threshold.

The operator watersheds_threshold segments regions (basins) that are separated from each other by a watershed that has a height of at least Threshold.

In the first step, watersheds_threshold computes the watersheds without applying a threshold, resulting in the same basins that would be obtained when calling watersheds (for more details please refer to the description of watersheds). In the second step, the basins are successively merged if they are separated by a watershed that is smaller than Threshold. Let B_1 and B_2 be the minimum gray values of two neighboring basins and W the minimum gray value of the watershed that separates the two basins. The watershed is eliminated and the two basins are merged if

$$\max\{W - B_1, W - B_2\} < \texttt{Threshold}.$$

The thus obtained basins are returned in Basins.

If Threshold is set to *0*, watersheds_threshold is comparable to watersheds except that no watersheds but only expanded basins are returned. If Threshold is set to the maximum gray value range of Image then no two basins are separated by a watershed exceeding Threshold, and hence, Basins will contain only one region.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
  Image to be segmented.
- ▷ **Basins** (output_object) ..... region-array ⇝ *object*
  Segments found (dark basins).
- ▷ **Threshold** (input_control) ..... number ⇝ *integer* / real
  Threshold for the watersheds.
  Default: 10
  Suggested values: Threshold ∈ {0, 5, 10, 20, 30, 50}
  Restriction: Threshold >= 0

Result

watersheds always returns 2 (H_MSG_TRUE). The behavior with respect to the input image and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with set_system. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

binomial_filter, gauss_filter, smooth_image, invert_image

Possible Successors

expand_region, select_shape, reduce_domain, opening

Alternatives

watersheds, pouring, watersheds_marker

Module

Foundation

Chapter 25

System

25.1 Compute Devices

activate_compute_device (: : DeviceHandle :)

Activate a compute device.

activate_compute_device activates the compute device DeviceHandle for the current HALCON thread. All subsequent HALCON operators called in this thread are executed on DeviceHandle if they provide an implementation (see get_operator_info) for this device. Use deactivate_compute_device to deactivate DeviceHandle.

Currently, only one compute device can be active for a HALCON thread. If a different device is active before activate_compute_device is called, this device is deactivated automatically.

Parameters

▷ **DeviceHandle** (input_control) ..... compute_device ⇝ *handle*
Compute device handle.

Example

```
open_compute_device (DeviceIdentifier, DeviceHandle)
read_image (Image, 'rings_and_nuts')
*
* Gaussian convolution on a compute device
activate_compute_device (DeviceHandle)
derivate_gauss (Image, DerivGauss, 5, 'none')
*
* Gaussian convolution on the CPU
deactivate_compute_device (DeviceHandle)
derivate_gauss (Image, DerivGauss, 5, 'none')
```

Result

activate_compute_device returns the value 2 (H_MSG_TRUE) if DeviceHandle is valid. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

Possible Predecessors

open_compute_device

See also

deactivate_compute_device, deactivate_all_compute_devices

Module

Foundation

deactivate_all_compute_devices (: : :)

Deactivate all compute devices.

deactivate_all_compute_devices deactivates all compute devices for the current HALCON thread.

Result

deactivate_all_compute_devices returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

activate_compute_device, deactivate_compute_device

Module

Foundation

deactivate_compute_device (: : DeviceHandle :)

Deactivate a compute device.

deactivate_compute_device deactivates the compute device DeviceHandle for the current HALCON thread. Use activate_compute_device to reactivate the device.

Parameters

▷ **DeviceHandle** (input_control) ..... compute_device ⇝ *handle*
Compute device handle.

Result

deactivate_compute_device returns the value 2 (H_MSG_TRUE) if DeviceHandle is valid. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

Possible Predecessors

activate_compute_device

See also

deactivate_all_compute_devices

Module

Foundation

get_compute_device_info (: : DeviceIdentifier, InfoName : Info)

Get information on a compute device.

The operator get_compute_device_info returns information on a compute device. In contrast to get_compute_device_param, only static information is queried, so that the device does not have to be opened (see open_compute_device) and activated (see activate_compute_device).

The following information can be queried:

'vendor' Vendor of the compute device.

'name' Name of the compute device.

'platform_version' Version of the compute device platform. E.g., OpenCL version in the case of OpenCL devices.

'driver_version' Version of the device driver.

'extensions' Supported OpenCL extensions.

'image_support' *'true'* if the device supports image objects.

'image2d_max_width' Maximum width of OpenCL image objects.

'image2d_max_height' Maximum height of OpenCL image objects.

'max_mem_alloc_size' Maximum size (in bytes) of an OpenCL memory block.

Parameters

▷ **DeviceIdentifier** (input_control) ..... integer ⇝ *integer*
Compute device handle.

▷ **InfoName** (input_control) ..... string ⇝ *string*
Name of Information to query.
Default: 'name'
List of values: InfoName ∈ {'driver_version', 'extensions', 'image_support', 'image2d_max_width', 'image2d_max_height', 'max_mem_alloc_size', 'name', 'platform_version', 'vendor'}

▷ **Info** (output_control) ..... string(-array) ⇝ *string* / integer / real
Returned information.

Result

The operator get_compute_device_info returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

query_available_compute_devices

Possible Successors

activate_compute_device

Module

Foundation

get_compute_device_param (: : DeviceHandle, GenParamName : GenParamValue)

Query compute device parameters.

get_compute_device_param returns parameters of the compute device DeviceHandle for the current HALCON thread in GenParamValue.

The following parameters can be queried (for more information see set_compute_device_param) by setting them for GenParamName:

'alloc_pinned' If *'true'*, the output image matrices of all operators are created in page locked (so called 'pinned') memory.

'asynchronous_execution' If *'true'*, operators executed on the compute device do not wait for the device computations to finish but return after initiation the computations.

'buffer_cache_capacity' Maximum size (in bytes) of the compute device buffer cache.

'buffer_cache_used' Current size (in bytes) of the compute device buffer cache.

'image_cache_capacity' Maximum size (in bytes) of the compute device image cache.

'image_cache_used' Current size (in bytes) of the compute device image cache.

'pinned_mem_cache_capacity' Maximum size (in bytes) of the page locked (pinned) memory cache.

'pinned_mem_cache_used' Current size (in bytes) of the page locked (pinned) memory cache.

Parameters

▷ **DeviceHandle** (input_control) . . . . . compute_device ⇝ *handle*
Compute device handle.

▷ **GenParamName** (input_control) . . . . . string ⇝ *string*
Name of the parameter to query.
Default: 'buffer_cache_capacity'
List of values: GenParamName ∈ {'alloc_pinned', 'asynchronous_execution', 'buffer_cache_capacity', 'buffer_cache_used', 'image_cache_capacity', 'image_cache_used', 'pinned_mem_cache_capacity', 'pinned_mem_cache_used'}

▷ **GenParamValue** (output_control) . . . . . string(-array) ⇝ *string* / integer / real
Value of the parameter.

Result

The operator get_compute_device_param returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

Possible Predecessors

activate_compute_device

See also

set_compute_device_param, get_compute_device_info

Module

Foundation

init_compute_device (: : DeviceHandle, Operators :)

Initialize a compute device.

init_compute_device initializes a compute device and prepares a list of operators given in Operators for the execution on this device. Further the device is activated for the current thread. If init_compute_device is not called or operators other than those in Operators are used on the device, the initialization is performed on demand.

Use get_operator_info to test if an operator qualifies for execution on the compute device.

Attention

Be aware that the execution time of init_compute_device depends on the number of Operators and may last up to several seconds.

Parameters

▷ **DeviceHandle** (input_control) . . . . . compute_device ⇝ *handle*
Compute device handle.

▷ **Operators** (input_control) . . . . . string-array ⇝ *string*
List of operators to prepare.
Default: 'all'
List of values: Operators ∈ {'all', 'derivate_gauss', 'sobel_amp'}

Result

The operator init_compute_device returns the value 2 (H_MSG_TRUE) if the initialization was successful. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

Possible Predecessors

query_available_compute_devices, open_compute_device

Possible Successors

activate_compute_device

Module

Foundation

open_compute_device (: : DeviceIdentifier : DeviceHandle)

Open a compute device.

open_compute_device opens the compute device referenced by DeviceIdentifier and returns DeviceHandle. Enable computation on DeviceHandle via activate_compute_device.

DeviceIdentifier must be obtained by query_available_compute_devices.

Parameters

▷ **DeviceIdentifier** (input_control) . . . . . integer ⇝ *integer*
Compute device Identifier.

▷ **DeviceHandle** (output_control) . . . . . compute_device ⇝ *handle*
Compute device handle.

Result

open_compute_device returns the value 2 (H_MSG_TRUE) if DeviceIdentifier is valid. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

query_available_compute_devices

Possible Successors

activate_compute_device, init_compute_device

Module

Foundation

query_available_compute_devices (: : : DeviceIdentifier)

Get the list of available compute devices.

Returns the list of available compute devices. Use get_compute_device_info to query information on a specific device. Use open_compute_device to open a device in HALCON.

At present, HALCON only supports OpenCL compatible GPUs supporting the OpenCL extension cl_khr_byte_addressable_store and image objects. If you are not sure whether a certain device is supported, please refer to the manufacturer.

Be aware that currently it is not possible to use OpenCL via the Windows Remote Desktop, because Windows Remote Desktop does not allow access to the graphics driver. You may use a VNC solution to use OpenCL on a remote computer. Of course you can use OpenCL via ssh on a Linux machine.

It is recommended to install the latest graphics drivers available for your GPU. To access the GPU on a Linux system you should be member of the 'video' group.

Parameters

▷ **DeviceIdentifier** (output_control) ..... integer-array ⇝ *integer*
List of available compute devices.

Example

```
query_available_compute_devices (DeviceIdentifiers)
for Index := 0 to |DeviceIdentifiers|-1 by 1
  get_compute_device_info (DeviceIdentifiers[Index], 'name', DeviceName)
  get_compute_device_info (DeviceIdentifiers[Index], 'vendor', DeviceVendor)
  if (DeviceVendor == 'NVIDIA Corporation' and \
    DeviceName == 'GeForce 8800 Ultra')
    open_compute_device (DeviceIdentifiers[Index], DeviceHandle)
    break
  endif
endfor
*
init_compute_device (DeviceHandle, 'derivate_gauss')
read_image (Image, 'rings_and_nuts')
*
* Gaussian convolution on a compute device
activate_compute_device (DeviceHandle)
derivate_gauss (Image, DerivGauss, 5, 'none')
*
* Gaussian convolution on the CPU
deactivate_compute_device (DeviceHandle)
derivate_gauss (Image, DerivGauss, 5, 'none')
```

Result

query_available_compute_devices returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: mutually exclusive (runs in parallel with other non-exclusive operators, but not with itself).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_compute_device_info, open_compute_device

Module

Foundation

release_all_compute_devices (: : :)

Close all compute devices.

release_all_compute_devices must not be used in HDevelop!

release_all_compute_devices closes all compute devices for the current HALCON thread and frees all associated resources. release_all_compute_devices is solely for the purpose of freeing all device related resources before unloading the HALCON library, if the library was loaded using a mechanism like LoadLibrary or dlopen. In all other cases you must not use release_all_compute_devices.

Result

release_all_compute_devices returns the value 2 (H_MSG_TRUE) if all devices were closed successfully. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

release_compute_device, deactivate_compute_device, deactivate_all_compute_devices

Module

Foundation

release_compute_device (: : DeviceHandle :)

Close a compute_device.

release_compute_device must not be used in HDevelop!

release_compute_device closes a compute devices for the current HALCON thread and frees all resources associated to DeviceHandle. release_compute_device is solely for the purpose of freeing all device related resources before unloading the HALCON library, if the library was loaded using a mechanism like LoadLibrary or dlopen. In all other cases you must not use release_compute_device!

Parameters

▷ **DeviceHandle** (input_control) ..... compute_device ⇝ *handle*
Compute device handle.

Result

release_compute_device returns the value 2 (H_MSG_TRUE) if DeviceHandle was closed successfully. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

See also

release_all_compute_devices, deactivate_compute_device, deactivate_all_compute_devices

Module

Foundation

set_compute_device_param (: : DeviceHandle, GenParamName, GenParamValue :)

Set parameters of an compute device.

set_compute_device_param sets parameters of the compute device DeviceHandle for the current HALCON thread.

The following parameters can be set:

'alloc_pinned' If *'true'*, the output image matrices of all operators (executed on the compute device or not) called in the current HALCON thread are created in page locked (so called 'pinned') memory. This accelerates the transfer between host and device memory. However, too excessive usage of page locked memory may have a negative impact on overall system performance. You should activate page locked memory allocation if the output image of of the next operator call is to be transferred to the device. Page locked memory allocation should be deactivated for all operator calls whose output images are not required on the compute device (see example).

List of values: *'true'*, *'false'*.

Default: *'true'*.

'asynchronous_execution' If *'true'*, operators executed on the compute device do not wait for the device computations to finish but return after initiating the computations. All device computations are synchronized as soon as the output of a compute device operation is used on the CPU (e.g., by disp_image).

List of values: *'true'*, *'false'*.

Default: *'true'*.

'buffer_cache_capacity' Maximum size (in bytes) of the compute device buffer cache.

Default: 1/3 of the available device memory

'image_cache_capacity' Maximum size (in bytes) of the compute device image cache.

Default: 1/3 of the available device memory

'pinned_mem_cache_capacity' Maximum size (in bytes) of the page locked (pinned) memory cache.

Default: *32000000* (32MByte)

Parameters

▷ **DeviceHandle** (input_control) . . . . . compute_device ⇝ *handle*
Compute device handle.

▷ **GenParamName** (input_control) . . . . . string ⇝ *string*
Name of the parameter to set.
Default: 'buffer_cache_capacity'
List of values: GenParamName ∈ {'alloc_pinned', 'asynchronous_execution', 'buffer_cache_capacity', 'image_cache_capacity', 'pinned_mem_cache_capacity'}

▷ **GenParamValue** (input_control) . . . . . string(-array) ⇝ *string* / integer / real
New parameter value.

Example

```
activate_compute_device (DeviceHandle)
read_image (Image, 'fuse')
set_compute_device_param (DeviceHandle, 'alloc_pinned', 'true')
* filter on compute device, output image is page locked
derivate_gauss (Image, DerivGauss, 3, 'gradient')
* filter result on the CPU, output image should not be page locked
set_compute_device_param (DeviceHandle, 'alloc_pinned', 'false')
median_image (DerivGauss, ImageMedian, 'circle', 1, 'mirrored')
```

Result

The operator `set_compute_device_param` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

Possible Predecessors

`activate_compute_device`

See also

`get_compute_device_param`

Module

Foundation

25.2 Database

`count_relation` (: : `RelationName` : `NumOfTuples`)

Number of entries in the HALCON database.

`count_relation` returns the number of entries `NumOfTuples` in the relation `RelationName` of the HALCON database since the database was switched on with the `set_system` parameter *'database'*. If the database is disabled, `NumOfTuples` is 0 and in addition a warning is raised if *'do_low_error'* is set with `set_system`. The state of the database can be queried by the parameter *'database'* of `get_system`.

The HALCON database is organized in five tables called relations. The entries in this relations are called tuples (not to mix up with HALCON tuples naming object arrays). If enabled, the HALCON database contains the basic relations for region data, image matrices, and XLDs, as well as the container relations for HALCON objects and HALCON tuples (i.e., object arrays). The HALCON objects region and image are constructed from elements from the region-data relations and the image-matrix relations. A region consists of a pointer to a tuple in the region-data relation. An image consists of a pointer to a tuple in the region-data relation and additionally of one or more pointers to tuples in the matrix relation. Multi-channel images have multiple matrix pointers.

Both regions and images are called objects. A region can be considered as the special case of an iconic object having no image matrices. For reasons of an efficient memory management, the tuples of the region-data relation and the image-matrix relation will be used by different objects together. Therefore there may be more images than image matrices. Only the three low-level relations *'image'*, *'region'*, and *'XLD'* are of relevance to the memory consumption. Image objects (regions as well as images) consist only of references on region and matrix data and therefore only need a couple of bytes of memory.

Possible values for `RelationName`:

'image': Image matrices. One matrix may be the component of multiple images (no redundant storage).

'region': Regions and image domains. One region may be the component of multiple image objects (no redundant storage).

'XLD': eXtended Line Description: Contours, polygons, parallels, lines, etc. XLD data types don't have gray values and are stored with subpixel accuracy.

'object': Iconic objects. Composed of a region (see *'region'*) and optionally image matrices (see *'image'*).

'tuple': In the compact mode, tuples of iconic objects are stored as a surrogate in this relation. Instead of working with the individual object keys, only this tuple key is used. It depends on the host language, whether the objects are passed individually (e.g., C++) or as tuples (e.g., C).

Certain database objects will be created already by the operator reset_obj_db and therefore have to be available all the time (the undefined gray value component, the objects 'full' (FULL_REGION in HALCON/C) and 'empty' (EMPTY_REGION in HALCON/C) as well as the herein included empty and full region). By calling get_channel_info, the operator therefore appears correspondingly also as 'creator' of the full and empty region. The operator can be used for example to check the completeness of the clear_obj operation.

Attention

Collecting database information is not thread-safe when passing iconic objects between threads, meaning when deleting objects in a different thread than generating them.

Parameters

▷ **RelationName** (input_control) ..... string ⇝ *string*
Relation of interest of the HALCON database.
Default: 'object'
List of values: RelationName ∈ {'image', 'region', 'XLD', 'object', 'tuple'}

▷ **NumOfTuples** (output_control) ..... integer ⇝ *integer*
Number of tuples in the relation.

Example

```
* Close the graphics window in order to close the graphics stack,
* which would influence the measurement.
dev_close_window ()
* Enable the measurement.
set_system ('database','true')
*
count_relation ('image', I1)
count_relation ('region', R1)
count_relation ('XLD', X1)
count_relation ('object', O1)
count_relation ('tuple', T1)
*
* Result:
* I1 = 0
* R1 = 0
* X1 = 0
* O1 = 0
* T1 = 0
*
read_image (Patras, 'patras')
*
count_relation ('image', I2)
count_relation ('region', R2)
count_relation ('XLD', X2)
count_relation ('object', O2)
count_relation ('tuple', T2)
*
* I2 = 3  (three channels of the rgb image 'patras')
* R2 = 1  (the image domain of the image 'patras')
* X2 = 0  (no XLD data)
* O2 = 1  (the iconic object holding the channels and the domain)
* T2 = 0  (no empty object or object array )
```

Result

If the parameter is correct, the operator count_relation returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

reset_obj_db

Possible Successors

set_system

See also

clear_obj

Module

Foundation

get_modules (: : : UsedModules, ModuleKey)

Query of used modules and the module key.

get_modules returns the modules UsedModules that were used up to this point in the HALCON session. Based on the used modules, a key is generated that is returned in ModuleKey. Each operator belongs to exactly one module. The information which modules where used can be reset by calling set_system with the *'reset_used_modules'* parameter.

Parameters

▷ **UsedModules** (output_control) . . . . . string-array ⇝ *string*
Names of used modules.

▷ **ModuleKey** (output_control) . . . . . integer ⇝ *integer*
Key for license manager.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

reset_obj_db (: : DefaultImageWidth, DefaultImageHeight, DefaultChannels :)

Reset the HALCON system for iconic objects.

The operator reset_obj_db reinitializes the HALCON system for iconic objects (which was set up during the HALCON initialization process). With this operator the five relations (gray value objects, region objects, XLD objects, image objects and object tuples) which are used for image processing with HALCON will be installed (see also count_relation). In case the HALCON database was enabled (set_system ('database','true')) and the relations already exist, all data entries listed in the relations will get deallocated!

The parameters DefaultImageWidth and DefaultImageHeight provide the initial values for the global maximum image size. If the first created object is an image, (e.g., read_image), the set values will only be overruled, if they are smaller than the size of the created object. If on the other hand the first object to be created is a region, the values will only be adjusted in case the new image is larger than the set values. This is not only the case for the first image which is created or read: the global image size will always be enlarged, if larger images are created.

The global image size is of consequence for the opening of windows (open_window) and the memory estimate and clipping of regions. Whenever the clip mode is activated, regions will be clipped according to the global

image size (set_system('clip_region','true')). This can lead to problems if images of various sizes are used. In this case only the fact that a region is smaller or of the same size as the largest image can be guaranteed.

The parameter DefaultChannels returns the most frequent number of channels of an image object. This value can be set to 0 if for the most part regions are used. If more channels than those having been set at the initialization are necessary for one image, the number will be enlarged dynamically for this image. If less channels than those having been set at the initialization are necessary for the image, the superfluous channels will be set as undefined. For the user this will seem as if they were non existent, however, memory is allocated unnecessarily.

The parameter values can be queried using the operator get_system.

Attention

If the operator reset_obj_db is not called at the beginning of a HALCON session, HALCON will be initialized automatically by the operator reset_obj_db(128,128,0) causing side effects on region clipping, accordingly. When the database was enabled by the parameter value *'database'* of the operator set_system, calling reset_obj_db will deallocate all iconic objects.

Parameters

▷ **DefaultImageWidth** (input_control) ..... integer ⇝ *integer*
Default image width (in pixels).
Default: 128
Suggested values: DefaultImageWidth ∈ {64, 128, 320, 640, 800, 1280}

▷ **DefaultImageHeight** (input_control) ..... integer ⇝ *integer*
Default image height (in pixels).
Default: 128
Suggested values: DefaultImageHeight ∈ {64, 128, 240, 480, 600, 1024}

▷ **DefaultChannels** (input_control) ..... integer ⇝ *integer*
Usual number of channels.
Default: 0
Suggested values: DefaultChannels ∈ {0, 1, 2, 3, 4, 5, 6, 7}

Result

The operator reset_obj_db returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception will be raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

get_channel_info, count_relation

Module

Foundation

25.3 Encrypted Item

read_encrypted_item (: : FileName : EncryptedItemHandle)

Read an encrypted item from a file.

The operator read_encrypted_item reads the file FileName and creates a new encrypted item in EncryptedItemHandle that is an identical copy of the saved item. The file FileName must have been created by the operator write_encrypted_item.

The default HALCON file extension for encrypted item is '.henc'.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Name of the file.
Default: 'encrypted_item.henc'
File extension: .bin

▷ **EncryptedItemHandle** (output_control) . . . . . encrypted_item ⇝ *handle*
Handle of the encrypted item.

Result

The operator read_encrypted_item returns the value 2 (H_MSG_TRUE) if the named file was found and correctly read. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

decrypt_serialized_item

See also

write_encrypted_item, encrypt_serialized_item

Module

Foundation

write_encrypted_item (: : EncryptedItemHandle, FileName :)

Write an encrypted item to a file.

The operator write_encrypted_item writes the encrypted item EncryptedItemHandle to the file FileName. The item can be read again with read_encrypted_item.

The default HALCON file extension for encrypted item is '.henc'.

Parameters

▷ **EncryptedItemHandle** (input_control) . . . . . encrypted_item ⇝ *handle*
Handle of the encrypted item.

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the file.
Default: 'encrypted_item.henc'
File extension: .bin

Result

The operator write_encrypted_item returns the value 2 (H_MSG_TRUE) if the passed handle is valid and if the encrypted item can be written into the named file. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

encrypt_serialized_item

See also

read_encrypted_item, decrypt_serialized_item

Module

Foundation

25.4 Error Handling

get_check (: : : Check)

State of the HALCON control modes.

Executing the operator get_check the user can inquire what kind of control modes are currently activated and which are not. Check gives the tuple containing the names of the control modes (see also set_check) which are preceded by a tilde (~, e.g., '~data'), if the corresponding control is deactivated.

Parameters

▷ **Check** (output_control) ..... string-array ⇝ *string*
Tuple of the currently activated control modes.

Result

get_check always returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_check

See also

set_check

Module

Foundation

get_error_text (: : ErrorCode : ErrorMessage)

Inquiry after the error text of a HALCON error number.

The operator get_error_text returns the error message for the corresponding HALCON error code. This is indeed the same text which will be given during an exception. The operator get_error_text is especially useful if the error treatment is programmed by the users themselves (see also set_check('~give_error')).

Attention

Unknown error codes will trigger a standard message.

Parameters

▷ **ErrorCode** (input_control) ..... integer ⇝ *integer*
HALCON error code.
Restriction: 1 <= ErrorCode && ErrorCode <= 36000

▷ **ErrorMessage** (output_control) ..... string ⇝ *string*
Corresponding error message.

Example

```
Herror   err;
char       message[MAX_STRING];

set_check("~give_error");
err = send_region(region,socket);
set_check("give_error");
if (err != H_MSG_TRUE) {
```

```
get_error_text((Hlong)err,message);
  fprintf(stderr,"my error message: %s\n",message);
  exit(1);
}
```

Result

The operator get_error_text returns the value 2 (H_MSG_TRUE), if the parameters are correct. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_check

See also

set_check

Module

none

get_extended_error_info (: : : OperatorName, ErrorCode, ErrorMessage)

Returns the extended error information for the calling thread's last HALCON error.

get_extended_error_info returns the last error code that occurred in the calling thread. OperatorName contains the operator name of the HALCON operator that has set this error code. ErrorCode holds the actual error code and ErrorMessage the corresponding error message. If no error code has been set so far or a HALCON error comes with no extended error information, OperatorName and ErrorMessage contain an empty string and ErrorCode is set to *0*. Please refer to 'Result' slot of the particular operator reference to determine whether and on which occasion an operator provides extended error information.

get_extended_error_info is thread-local, i.e., the extended error information is maintained on a per thread basis. Setting an extended error code in one thread does not affect the extended error information in any other thread. This also implies that an error code can only be retrieved within the thread that was calling the operator setting the extended error. A thread-local extended error will be cleared (and set again when applicable) as soon as an error occurs in any other successive operator call within the same thread, if get_system is called from within the same thread with the *'tsp_clear_extended_error'* parameter, or the thread terminates.

Parameters

▷ **OperatorName** (output_control) ..... string ⇝ *string*
Operator that set the error code.

▷ **ErrorCode** (output_control) ..... integer ⇝ *integer*
Extended error code.

▷ **ErrorMessage** (output_control) ..... string ⇝ *string*
Extended error message.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

get_error_text

Module

none

get_spy (: : Class : Value)

Current configuration of the HALCON debugging-tool.

The operator get_spy returns the current configuration of spy, the HALCON debugging tool. The available control modes (possible choices for Class) as well as the corresponding tuning possibilities (possible values for Value) can be called up by using the operator query_spy. You will find a more detailed description under set_spy.

Parameters

▷ **Class** (input_control) . . . . . string ⇝ *string*
Control mode
Default: 'mode'
List of values: Class ∈ {'db', 'error', 'input_control', 'internal', 'log_file', 'mode', 'operator', 'output_control', 'parameter_values', 'time'}

▷ **Value** (output_control) . . . . . string ⇝ *string* / integer / real
State of the control mode.

Result

The operator get_spy returns the value 2 (H_MSG_TRUE) if the parameter Class is correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

reset_obj_db

See also

set_spy, query_spy

Module

Foundation

query_spy (: : : Classes, Values)

Inquiring for possible settings of the HALCON debugging tool.

The operator query_spy returns all possible settings of spy, the HALCON debugging tool, i.e. all the available control modes (Classes) as well as the corresponding possible ways of setting (Values). For a more detailed description of spy see operator set_spy.

Attention

The values of Values cannot be used as direct input for set_spy, as they are transmitted as a symbolic constant.

Parameters

▷ **Classes** (output_control) . . . . . string-array ⇝ *string*
Available control modes (see also set_spy).

▷ **Values** (output_control) . . . . . string-array ⇝ *string*
Corresponding state of the control modes.

Result

query_spy always returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

reset_obj_db

See also

set_spy, get_spy

Module

Foundation

set_check (: : Check :)

Activating and deactivating of HALCON control modes.

With the help of the operator set_check different control modes of the HALCON system can be activated or deactivated. If a certain control mode is activated, parameters etc., will be checked at runtime. Whenever an inconsistency is hereby detected, the program will be interrupted by an exception.

It is recommendable to activate the control modes during the development of a program and to deactivate them only after a successfully concluded test run. For if the control mode is deactivated and an error occurs, the system may react in an unpredictable way.

The HALCON system provides various possible control modes which can be activated and deactivated independently. By calling the operator set_check with the name (Check) of the desired control mode, this control mode is activated; the control mode is deactivated by passing its name prefixed with a tilde (~, e.g., '~data').

Except for the control mode *'memory'* that is set exclusively for the whole system, all control modes are set thread-specific by default. When preceding the name of these control modes with a *'ref_'* prefix, only the reference value of the particular mode is set. The reference value is used to initialize the respective mode when starting a new user thread.

Available control modes:

'color': If this control mode is activated, only colors may be used which are supported by the display for the currently active window. Otherwise an error message is displayed.

In case of deactivated control mode and non existent colors, the em nearest color is used (see also set_color, set_gray, set_rgb).

'text': If this control mode is activated, it will check the coordinates during the setting of the text cursor as well as during the display of strings (write_string) to the effect whether a part of a sign would lie outside the window frame (a fact which is not forbidden in principle by the system).

If the control mode is deactivated, the text will be clipped at the window frame.

'data': (For program development)

Checks the consistency of image objects (regions and gray value components.

'interface': If this control mode is activated, the interface between the host language and the HALCON procedures will be checked in course (e.g., typifying and counting of the values).

'database': This is a consistency check of objects (e.g., checks whether an object which shall be canceled does indeed exist or not.)

'give_error': Determines whether errors shall trigger exceptions or not. If this control modes is deactivated, the application program must provide a suitable error treatment itself. Please note that errors which are not reported usually lead to undefined output parameters which may cause an unpredictable reaction of the program. Details about how to handle exceptions in the different HALCON language interfaces can be found in the HALCON "Programmer's Guide" and the "HDevelop User's Guide".

'father': If this control mode is activated when calling the operators open_window, HALCON allows only the usage of the number of another HALCON window as the father window of the new window; otherwise it allows also the usage of IDs of operating system windows as the father window. This control mode is only relevant for windows of type 'WIN32-Window' and 'X-Window'.

'region': (For program development)

Checks the consistency of chords (this may lead to a notable speed reduction of routines).

'clear': Normally, if a list of objects shall be canceled by using clear_obj, an exception will be raised, in case individual objects do not or no longer exist. If the *'clear'* mode is activated, such objects will be ignored.

'memory': (For program development)
Checks the memory blocks freed by the HALCON memory management on consistency and overwriting of memory borders.

'all': Activates all control modes.

'none': Deactivates all control modes.

'default': Default settings: ['give_error','database']

Parameters

▷ **Check** (input_control) ..... string(-array) ⇝ *string*
Desired control mode.
Default: 'default'
List of values: Check ∈ {'color', 'text', 'database', 'data', 'interface', 'give_error', 'father', 'region', 'clear', 'memory', 'all', 'none', 'default'}

Result

The operator set_check returns the value 2 (H_MSG_TRUE), if the parameters are correct. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

get_check, set_color, set_rgb, set_hsi, write_string

Module

none

set_spy (: : Class, Value :)

Control of the HALCON Debugging Tools.

The operator set_spy is the HALCON debugging tool. This tool allows the flexible control of the input and output data of HALCON-operators - in graphical as well as in textual form. The data control is activated by using

```
set_spy(::'mode','on':),
```

and deactivated by using

```
set_spy(::'mode','off':).
```

The debugging tool can further be activated with the help of the environment variable HALCONSPY. The definition of this variable corresponds to calling up *'mode'* and *'on'*.

The following control modes (Class) can be tuned using Value:

Class = *'operator'*
When a routine is called, its name and the names of its parameters will be given (in TRIAS notation).
Value: *'on'* or *'off'*.
Default: 'off'.

Class = *'input_control'*
When a routine is called, the names and values of the input control parameters will be given.
Value: *'on'* or *'off'*.
Default: 'off'.

Class = *'output_control'*
When a routine is called, the names and values of the output control parameters are given.
Value: *'on'* or *'off'*.
Default: 'off'.

Class = *'parameter_values'*
Additional information on *'input_control'* and *'output_control'*: indicates how many values per parameter shall be displayed at most (maximum tuple length of the output).
Value: tuple length (integer)
Default: 4

Class = *'db'*
Information concerning the 4 relations in the HALCON-database. This is especially valuable in looking for forgotten clear_obj.
Value: *'on'* or *'off'*.
Default: 'off'.

Class = *'time'*
Processing time of the operator
Value: *'on'* or *'off'*.
Default: 'off'.

Class = *'log_file'*
Spy can hereby divert the text output into a file having been opened with open_file.
Value: a file handle (see open_file)

Class = *'error'*
If *'error'* is activated and an internal error occurs, spy will show the internal procedures (file/line) concerned.
Value: *'on'* or *'off'*.
Default: 'off'.

Class = *'internal'*
If *'internal'* is activated, spy will display the internal procedures and their parameters (file/line) while an HALCON-operator is processed.
Value: *'on'* or *'off'*.
Default: 'off'.

Each output starts with the thread handle, followed by a global counter that marks the order of the outputs. In multi-threaded applications, this information can be used to assign the output to individual user threads and to reconstruct the chronological sequence.

Attention

Note that under Windows the output on stdout works only in combination with a console application.

Parameters

▷ **Class** (input_control) . . . . . string ⇝ *string*
Control mode
Default: 'mode'
List of values: Class ∈ {'db', 'error', 'input_control', 'internal', 'log_file', 'mode', 'operator', 'output_control', 'parameter_values', 'time'}

▷ **Value** (input_control) . . . . . string ⇝ *string* / integer / real
State of the control mode to be set.
Default: 'on'
Suggested values: Value ∈ {'on', 'off', 1, 2, 3, 4, 5, 10, 50, 0.0, 1.0, 2.0, 5.0, 10.0}

Example

```
/* init spy: Setting of the wished control modi */
set_spy("mode","on");
set_spy("operator","on");
set_spy("input_control","on");
set_spy("output_control","on");
/* calling of program section, that will be examined */
set_spy("mode","off");
```

Result

The operator `set_spy` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`reset_obj_db`

See also

`get_spy`, `query_spy`

Module

Foundation

25.5 I/O Devices

`close_io_channel` (: : `IOChannelHandle` :)

Close I/O channels.

The operator `close_io_channel` closes transmission channels opened by `open_io_channel`. The references to the channels are passed in `IOChannelHandle`. In particular, allocated memory and reserved device resources are released.

Parameters

▷ **`IOChannelHandle`** (input_control) . . . . . io_channel(-array) ⇝ *handle*
Handles of the opened I/O channels.

Result

If the parameters are valid, the operator `close_io_channel` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator `get_extended_error_info`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- IOChannelHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`open_io_channel`

Possible Successors

`close_io_device`

Module

Foundation

close_io_device (: : IODeviceHandle :)

Close the specified I/O device.

The operator close_io_device closes the I/O device specified by IODeviceHandle. In particular, all associated channels are closed (i.e., close_io_channel is called implicitly on all channels that had been opened for this device) and allocated memory for data buffers is released.

Parameters

▷ **IODeviceHandle** (input_control) ..... io_device ⇝ *handle*
Handle of the opened I/O device.

Result

If the parameters are valid, the operator close_io_device returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- IODeviceHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

open_io_device

Module

Foundation

control_io_channel (: : IOChannelHandle, Action,
Argument : Result)

Perform an action on I/O channels.

control_io_channel performs an action on the specified I/O channels. The supported parameters are interface-specific and listed in the corresponding documentation of the used I/O interface that can be found in the directory doc/html/reference/io.

Parameters

▷ **IOChannelHandle** (input_control) ..... io_channel(-array) ⇝ *handle*
Handles of the opened I/O channels.

▷ **Action** (input_control) ..... string ⇝ *string*
Name of the action to perform.

▷ **Argument** (input_control) ..... string-array ⇝ *string* / integer / real
List of arguments for the action.
Default: []

▷ **Result** (output_control) ..... string-array ⇝ *string* / integer / real
List of values returned by the action.

Result

If the parameters are valid, the operator control_io_channel returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. In this case an extended error information may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_io_channel

Module

Foundation

control_io_device (: : IODeviceHandle, Action, Argument : Result)

Perform an action on the I/O device.

control_io_device performs an action on the I/O device. The supported parameters are interface-specific and listed in the corresponding documentation of the used I/O interface that can be found in the directory doc/html/reference/io.

Parameters

▷ **IODeviceHandle** (input_control) ..... io_device ⇝ *handle*
Handle of the opened I/O device.

▷ **Action** (input_control) ..... string ⇝ *string*
Name of the action to perform.

▷ **Argument** (input_control) ..... string(-array) ⇝ *string* / integer / real
List of arguments for the action.
Default: []

▷ **Result** (output_control) ..... string-array ⇝ *string* / integer / real
List of result values returned by the action.

Result

If the parameters are valid, the operator control_io_device returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. In this case an extended error information may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_io_device

Module

Foundation

control_io_interface (: : IOInterfaceName, Action, Argument : Result)

Perform an action on the I/O interface.

control_io_interface performs an action on the I/O interface. The supported parameters are interface-specific and listed in the corresponding documentation of the used I/O interface that can be found in the directory doc/html/reference/io.

The special value *'keep_open'* for Action is used to control when the interface should be unloaded. If Argument is set to *'true'*, the interface will remain loaded until the HALCON process is terminated. If it is set to *'false'*, the interface will be unloaded as soon as all its device instances are closed. This is the default behavior.

Attention

On Windows Systems, error dialog boxes from the operating system can occur when dependency modules of the interface are not found, e.g., the according SDK was not installed. The occurrence of the error boxes can be controlled by setting Windows' Error Mode. Please refer to the description of SetErrorMode within the Windows MSDN documentation.

Parameters

- ▷ **IOInterfaceName** (input_control) . . . . . string ⇝ *string*
  HALCON I/O interface name.
  Default: []
  Suggested values: IOInterfaceName ∈ {'ADLINK-DAQPilot', 'ADLINK-EOS', 'Advantech', 'Contec', 'Hilscher-cifX', 'Interface', 'Linux-GPIO', 'NIDAQmx', 'OPC_Classic', 'OPC_UA'}
- ▷ **Action** (input_control) . . . . . string ⇝ *string*
  Name of the action to perform.
- ▷ **Argument** (input_control) . . . . . string(-array) ⇝ *string* / integer / real
  List of arguments for the action.
  Default: []
- ▷ **Result** (output_control) . . . . . string-array ⇝ *string* / integer / real
  List of results returned by the action.

Result

If the parameters are valid, the operator control_io_interface returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. In this case an extended error information may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

query_io_interface

Module

Foundation

get_io_channel_param (: : IOChannelHandle, GenParamName : GenParamValue)

Query specific parameters of I/O channels.

get_io_channel_param queries settings of specific I/O transmission channels. The query parameters are passed in GenParamName, the corresponding configuration values are returned in GenParamValue. IOChannelHandle specifies the channels returned in open_io_channel for a specific device instance.

Please check the directory doc/html/reference/io for documentation about your specific I/O device interface, where all supported channel-specific parameters are listed.

Parameters

- ▷ **IOChannelHandle** (input_control) . . . . . io_channel(-array) ⇝ *handle*
  Handles of the opened I/O channels.
- ▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
  Parameter names.
  Default: 'param_name'
  Suggested values: GenParamName ∈ {'io_channel_name', 'param_name'}
- ▷ **GenParamValue** (output_control) . . . . . string-array ⇝ *string* / integer / real / handle
  Parameter values.

Result

If the parameters are valid, the operator `get_io_channel_param` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator `get_extended_error_info`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_io_channel`

Possible Successors

`read_io_channel`, `write_io_channel`

See also

`open_io_channel`, `query_io_device`, `set_io_channel_param`

Module

Foundation

get_io_device_param (: : IODeviceHandle, GenParamName : GenParamValue)

Query settings of an I/O device instance.

`get_io_device_param` queries configuration settings of a specific device instance. The query parameters are passed in `GenParamName`, the corresponding configuration values are returned in `GenParamValue`. `IODeviceHandle` specifies the device returned in `open_io_device`.

Please check the directory `doc/html/reference/io` for documentation about your specific I/O device interface, where all supported device specific parameters are listed.

Parameters

▷ **IODeviceHandle** (input_control) . . . . . io_device ⇝ *handle*
Handle of the opened I/O device.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Parameter names.
Default: 'param_name'
Suggested values: `GenParamName` ∈ {'io_device_name', 'param_name'}

▷ **GenParamValue** (output_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real / handle
Parameter values.

Result

If the parameters are valid, the operator `get_io_device_param` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator `get_extended_error_info`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_io_device`, `set_io_device_param`

Possible Successors

`open_io_channel`

See also

open_io_device, query_io_interface, set_io_device_param

Module

Foundation

open_io_channel (: : IODeviceHandle, IOChannelName, GenParamName, GenParamValue : IOChannelHandle)

Open and configure I/O channels.

The operator open_io_channel opens and configures transmission channels of an opened I/O device instance. The device instance must have been opened by open_io_device before and is passed to IODeviceHandle. The transmission channels themselves are addressed by their names passed to IOChannelName. Available names can be queried using query_io_device with query parameter *'io_channel_name'*. The channels can be configured by the parameters GenParamName and GenParamValue. A reference to each transmission channel is returned in IOChannelHandle. If the instance of the channel is not needed any more, it should be released and closed via the operator close_io_channel. Besides, it will be closed automatically when closing the associated device instance by close_io_device.

Having opened a transmission channel, values can be read and written by the operators read_io_channel and write_io_channel on the channel.

An opened channel can be reconfigured by using the operators set_io_channel_param and get_io_channel_param.

Please check the directory doc/html/reference/io for documentation about your specific I/O device interface, where all supported device specific parameters are listed.

Parameters

- ▷ **IODeviceHandle** (input_control) ..... io_device ⇝ *handle*
  Handle of the opened I/O device.
- ▷ **IOChannelName** (input_control) ..... string(-array) ⇝ *string*
  HALCON I/O channel names of the specified device.
- ▷ **GenParamName** (input_control) ..... string-array ⇝ *string*
  Parameter names.
  Default: []
- ▷ **GenParamValue** (input_control) ..... string-array ⇝ *string* / integer / real
  Parameter values.
  Default: []
- ▷ **IOChannelHandle** (output_control) ..... io_channel(-array) ⇝ *handle*
  Handles of the opened I/O channel.

Example

```
query_io_device (IODeviceHandle, [], 'io_channel_names.digital_output', \
                 ChannelOutputNames)
open_io_channel (IODeviceHandle, ChannelOutputNames[0], [], [], \
                 IOChannelHandle)
write_io_channel (IOChannelHandle, 1, Status)
```

Result

If the parameters are valid, the operator open_io_channel returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

open_io_device, query_io_device

Possible Successors

set_io_channel_param, read_io_channel, write_io_channel

Module

Foundation

open_io_device (: : IOInterfaceName, IODeviceName, GenParamName, GenParamValue : IODeviceHandle)

Open and configure an I/O device.

The operator open_io_device opens and configures the chosen I/O device interface for the specific device. The device interface is specified by the base name of the underlying library and passed to IOInterfaceName. The interface library will be loaded dynamically when configuring the first specific device for this device interface. The specific device itself is addressed by a name passed to IODeviceName. Available names can be queried by query_io_interface. The device can be configured by the parameters GenParamName and GenParamValue. A reference to the device instance is returned in IODeviceHandle. If the instance of the device is not needed any more, it should be released and closed via the operator close_io_device.

Having opened a specific device, a transmission channel can be opened by calling open_io_channel. Values on the I/O channel can be read and written by the operators read_io_channel and write_io_channel on an opened transmission channel, afterwards.

An opened device instance can be reconfigured by using the operators set_io_device_param and get_io_device_param.

Please check the directory doc/html/reference/io for documentation about your specific I/O device interface, where all supported device specific parameters are listed.

Attention

On Windows Systems, error dialog boxes from the operating system can occur when dependency modules of the interface are not found, e.g., the according SDK was not installed. The occurrence of the error boxes can be controlled by setting Windows' Error Mode. Please refer to the description of SetErrorMode within the Windows MSDN documentation.

Parameters

▷ **IOInterfaceName** (input_control) ..... string ⇝ *string*
HALCON I/O interface name.
Default: []
Suggested values: IOInterfaceName ∈ {'ADLINK-DAQPilot', 'ADLINK-EOS', 'Advantech', 'Contec', 'Hilscher-cifX', 'Interface', 'Linux-GPIO', 'NIDAQmx', 'OPC_Classic', 'OPC_UA'}

▷ **IODeviceName** (input_control) ..... tuple ⇝ *string*
I/O device name.
Default: []

▷ **GenParamName** (input_control) ..... string-array ⇝ *string*
Dynamic parameter names.
Default: []

▷ **GenParamValue** (input_control) ..... string-array ⇝ *string* / integer / real
Dynamic parameter values.
Default: []

▷ **IODeviceHandle** (output_control) ..... io_device ⇝ *handle*
Handle of the opened I/O device.

Example

```
* Select a suitable i/o device interface of name IOInterfaceName
query_io_interface (IOInterfaceName, 'io_device_names', DeviceNames)
open_io_device (IOInterfaceName, DeviceNames[0], [], [], IODeviceHandle)
query_io_device (IODeviceHandle, [], 'io_channel_names.digital_input', \
                 ChannelInputNames)
open_io_channel (IODeviceHandle, ChannelInputNames[0], [], [], \
                 IOChannelHandle)
read_io_channel (IOChannelHandle, Value, Status)
```

Result

If the parameters are valid, the operator `open_io_device` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator `get_extended_error_info`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`query_io_interface`

Possible Successors

`set_io_device_param`, `get_io_device_param`, `open_io_channel`

Module

Foundation

query_io_device (: : IODeviceHandle, IOChannelName, Query : Result)

Query information about channels of the specified I/O device.

The operator `query_io_device` returns information about transmission channels of a specified I/O device. The handle to the I/O device is passed to in `IODeviceHandle`, whereas the channels are addressed by `IOChannelName`. The desired information about the I/O channels is specified via `Query`. A list of all supported channel names is usually returned when passing *'io_channel_name'* to `Query` while the input parameter `IOChannelName` is ignored.

Please check the directory `doc/html/reference/io` for documentation about your specific I/O device interface, where all supported channel-specific parameters are listed.

Parameters

▷ **IODeviceHandle** (input_control) ..... io_device ⇝ *handle*
Handle of the opened I/O device.

▷ **IOChannelName** (input_control) ..... string(-array) ⇝ *string*
Channel names to query.

▷ **Query** (input_control) ..... string-array ⇝ *string*
Name of the query.
Default: 'param_name'
List of values: `Query` ∈ {'io_channel_names', 'param_name'}

▷ **Result** (output_control) ..... string-array ⇝ *string* / integer / real
List of values (according to `Query`).

Example

```
* Select a suitable i/o device interface of name IOInterfaceName
query_io_interface (IOInterfaceName, 'io_device_names', DeviceNames)
open_io_device (IOInterfaceName, DeviceNames[0], [], [], IODeviceHandle)
query_io_device (IODeviceHandle, [], 'io_channel_names.digital_input', \
                 ChannelInputNames)
open_io_channel (IODeviceHandle, ChannelInputNames[0], [], [], \
                 IOChannelHandle)
read_io_channel (IOChannelHandle, Value, Status)
```

Result

If the parameters are valid, the operator `query_io_device` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator `get_extended_error_info`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`open_io_device`

Module

Foundation

query_io_interface (: : IOInterfaceName, Query : Result)

Query information about the specified I/O device interface.

The operator `query_io_interface` returns information about the I/O device interface `IOInterfaceName`. The desired information is specified via `Query`. If applicable, `Result` contains a list of supported values.

Generally, when passing the value *'interface_name'* for `Query`, all available I/O interfaces are returned. The value of the parameter `IOInterfaceName` is ignored in this case.

Please check the directory `doc/html/reference/io` for documentation about your specific I/O device interface, where all supported interface specific parameters are listed.

Attention

On Windows Systems, error dialog boxes from the operating system can occur when dependency modules of the interface are not found, e.g., the according SDK was not installed. The occurrence of the error boxes can be controlled by setting Windows' Error Mode. Please refer to the description of SetErrorMode within the Windows MSDN documentation.

Parameters

▷ **IOInterfaceName** (input_control) . . . . . string ⇝ *string*
HALCON I/O interface name.
Default: []
Suggested values: `IOInterfaceName` ∈ {'ADLINK-DAQPilot', 'ADLINK-EOS', 'Advantech', 'Contec', 'Hilscher-cifX', 'Interface', 'Linux-GPIO', 'NIDAQmx', 'OPC_Classic', 'OPC_UA'}

▷ **Query** (input_control) . . . . . string(-array) ⇝ *string*
Parameter name of the query.
Default: 'io_device_names'
Suggested values: `Query` ∈ {'interface_name', 'io_device_names', 'param_name', 'revision'}

▷ **Result** (output_control) . . . . . string-array ⇝ *string* / integer / real
List of result values (according to `Query`).

Example

```
* Select a suitable i/o device interface of name IOInterfaceName
query_io_interface (IOInterfaceName, 'io_device_names', DeviceNames)
open_io_device (IOInterfaceName, DeviceNames[0], [], [], IODeviceHandle)
query_io_device (IODeviceHandle, [], 'io_channel_names.digital_input', \
                 ChannelInputNames)
open_io_channel (IODeviceHandle, ChannelInputNames[0], [], [], \
                 IOChannelHandle)
read_io_channel (IOChannelHandle, Value, Status)
```

Result

If the parameters are valid, the operator `query_io_interface` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator `get_extended_error_info`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`open_io_device`

Module

Foundation

read_io_channel (: : IOChannelHandle : Value, Status)

Read a value from the specified I/O channels.

`read_io_channel` reads input values from the specified transmission channels. The channels are passed in `IOChannelHandle` and can be opened and configured by `open_io_channel`, the values are returned in `Value`. The parameter `Status` returns for each read value in `Value` a quality message. The value *0* indicates that the corresponding value of `Value` could be read. Any other status value depends on the interface. Please refer to the interface-specific documentation in the directory `doc/html/reference/io` for further explanation.

Parameters

▷ **IOChannelHandle** (input_control) ..... io_channel(-array) ⇝ *handle*
Handles of the opened I/O channels.

▷ **Value** (output_control) ..... tuple-array ⇝ *integer* / real / string / handle
Read value.

▷ **Status** (output_control) ..... integer-array ⇝ *integer*
Status of read value.

Result

If the parameters are valid, the operator `read_io_channel` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator `get_extended_error_info`.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_io_channel`, `set_io_channel_param`

Possible Successors

close_io_channel

See also

write_io_channel, query_io_device, set_io_channel_param

Module

Foundation

set_io_channel_param (: : IOChannelHandle, GenParamName, GenParamValue :)

Set specific parameters of I/O channels.

set_io_channel_param (re)configures the given transmission channels. The configuration parameters are passed in GenParamName, the corresponding values are passed in GenParamValue. IOChannelHandle specifies the transmission channels returned in open_io_channel.

Please check the directory doc/html/reference/io for documentation about your specific I/O device interface, where all supported channel-specific parameters are listed.

Parameters

▷ **IOChannelHandle** (input_control) ..... io_channel(-array) ⇝ *handle*
Handles of the opened I/O channels.

▷ **GenParamName** (input_control) ..... string-array ⇝ *string*
Parameter names.
Default: []

▷ **GenParamValue** (input_control) ..... string-array ⇝ *string* / integer / real / handle
Parameter values to set.
Default: []

Result

If the parameters are valid, the operator set_io_channel_param returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_io_channel

Possible Successors

read_io_channel, write_io_channel

See also

open_io_channel, query_io_device, get_io_channel_param, read_io_channel, write_io_channel

Module

Foundation

set_io_device_param (: : IODeviceHandle, GenParamName, GenParamValue :)

Configure a specific I/O device instance.

set_io_device_param (re)configures a specific device instance. The configuration parameters are passed in GenParamName, the corresponding values are passed in GenParamValue. IODeviceHandle specifies the device specified in open_io_device.

Please check the directory doc/html/reference/io for documentation about your specific I/O device interface, where all supported device specific parameters are listed.

Parameters

▷ **IODeviceHandle** (input_control) . . . . . io_device ⇝ *handle*
Handle of the opened I/O device.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Parameter names.
Default: []

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real / handle
Parameter values to set.
Default: []

Result

If the parameters are valid, the operator set_io_device_param returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_io_device

Possible Successors

open_io_channel

See also

open_io_device, query_io_interface, get_io_device_param

Module

Foundation

write_io_channel (: : IOChannelHandle, Value : Status)

Write a value to the specified I/O channels.

write_io_channel writes values to the specified transmission channels. The channels are passed in IOChannelHandle and can be opened and configured by open_io_channel, the values are passed to Value. The parameter Status returns for each written value in Value a quality message. The value *0* indicates that the corresponding value of Value could be written. Any other status value depends on the interface. Please refer to the interface-specific documentation in the directory doc/html/reference/io for further explanation.

Parameters

▷ **IOChannelHandle** (input_control) . . . . . io_channel(-array) ⇝ *handle*
Handles of the opened I/O channels.

▷ **Value** (input_control) . . . . . tuple-array ⇝ *integer* / real / string / handle
Write values.

▷ **Status** (output_control) . . . . . integer-array ⇝ *integer*
Status of written values.

Result

If the parameters are valid, the operator write_io_channel returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. In this case, an extended error information may be set and can be queried with the operator get_extended_error_info.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_io_channel, set_io_channel_param

Possible Successors

close_io_channel

See also

read_io_channel, open_io_channel, query_io_device, set_io_channel_param

Module

Foundation

25.6 Information

get_chapter_info (: : Chapter : Info)

Get information concerning the chapters on operators.

The operator get_chapter_info gives information concerning the chapters on operators. If instead of Chapter the empty string is transmitted, the routine will provide in Info the names of all chapters. If on the other hand a certain chapter or a chapter and its subchapter(s) are indicated (by a tuple of names), the corresponding subchapters or - in case there are no further subchapters - the names of the corresponding operators will be given. The organization of the chapters on operators is the same as the organization of chapters and subchapters in the HALCON-manual. Please note: The chapters on operators respectively the subchapters concerning an individual operator can be called by using the operator get_operator_info(::<Name>,'chapter',Info:). The Online-texts will be taken from the files operators_[LANG].ref, operators_[LANG].sta, operators_[LANG].key, operators_[LANG].num and operators_[LANG].idx , which will be searched by HALCON in the currently used directory or the directory 'help_dir' (see also get_system and set_system).

Attention

The encoding of the result is UTF-8.

Parameters

▷ **Chapter** (input_control) ..... string(-array) ⇝ *string*
Operator class or subclass of interest.
Default: ''

▷ **Info** (output_control) ..... string-array ⇝ *string*
Operator classes (Chapter = '') or operator subclasses respectively operators.

Result

If the parameter values are correct and the helpfile is available, the operator get_chapter_info returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_system, set_system

See also

get_operator_info, get_system, set_system

Module

Foundation

get_keywords (: : OperatorName : Keywords)

Get keywords which are assigned to operators.

The operator get_keywords returns all the keywords in the online-texts corresponding to those operators which have the indicated substring OperatorName in their name. If instead of OperatorName the empty string is transmitted, the operator get_keywords returns all keywords. The keywords of an individual operator can also be called by using the operator get_operator_info. The online-texts will be taken from the files operators_[LANG].ref, operators_[LANG].sta, operators_[LANG].key, operators_[LANG].num and operators_[LANG].idx, which are searched by HALCON in the currently used directory and in the directory 'help_dir' (see also get_system and set_system).

Attention

The encoding of the result is UTF-8.

Parameters

▷ **OperatorName** (input_control) ..... proc_name ⇝ *string*
Substring in the names of those operators for which keywords are needed.
Default: 'get_keywords'

▷ **Keywords** (output_control) ..... string-array ⇝ *string*
Keywords for the operators.

Result

The operator get_keywords returns the value 2 (H_MSG_TRUE) if the parameters are correct and the helpfiles are available. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_chapter_info

Alternatives

get_operator_info

See also

get_operator_name, search_operator, get_param_info

Module

Foundation

get_operator_info (: : OperatorName, Slot : Information)

Get information concerning a HALCON-operator.

With the help of the operator get_operator_info the online-texts concerning a certain operator can be called (see also get_operator_name). The form of information available for all operators (Slot) can be called using the operator query_operator_info. For the time being the following slots are available:

'short': Short description of the operator.

'abstract': Description of the operator.

'chapter': Name(s) of the chapter(s) in the operator hierarchy (chapter, subchapter in the HALCON manual).

'functionality': Functionality is equivalent to the object class to which the operator can be assigned.

'keywords': Keywords of the operator (optional).

'example': Example for the use of the operator (optional). The operator 'example.LANGUAGE' (LANGUAGE ∈ {trias,c,c++,c#,vb.net}) calls up examples for a certain language if available. If the language is not indicated or if no example is available in this language, the TRIAS-example will be returned.

'complexity': Complexity of the operator (optional).

'effect': Not in use so far.

'parallelization': Characteristic timeout and parallel behavior of an operator.

'execution information': Characteristic timeout and parallel behavior of an operator.

'parallel_method': Method of automatic operator parallelization.

'interrupt_mode': Modes of interruption the operator can deal with. See `set_operator_timeout` and `interrupt_operator` on how to use and set these interrupt modes.

'alternatives': Alternative operators (optional).

'see_also': Operators containing further information (optional).

'predecessor': Possible and sensible predecessor

'successor': Possible and sensible successor

'result_state': Return value of the operator (`2` (`H_MSG_TRUE`), `3` (`H_MSG_FALSE`), `5` (`H_MSG_FAIL`), `4` (`H_MSG_VOID`) or EXCEPTION).

'attention': Restrictions and advice concerning the correct use of the operator (optional).

'parameter': Names of the parameter of the operator (see also `get_param_info`).

'references': Literary references (optional).

'module': The module to which the operator is assigned.

'dynamic_modules': List of modules the operator can be assigned to by dynamic licensing (depending on its usage).

'html_path': The directory where the HTML documentation of the operator resides.

'warning': Possible warnings for using the operator.

'compute_device': List of compute devices supported by the operator.

The texts will be taken from the `operators_[LANG].ref`, `operators_[LANG].sta`, `operators_[LANG].key`, `operators_[LANG].num` and `operators_[LANG].idx`, which will be searched by HALCON in the currently used directory or in the directory 'help_dir' (respectively 'user_help_dir') (see also `get_system` and `set_system`).

Attention

The encoding of the result is UTF-8.

Parameters

▷ **`OperatorName`** (input_control) ..... proc_name ⇝ *string*
Name of the operator on which more information is needed.
Default: 'get_operator_info'

▷ **`Slot`** (input_control) ..... string ⇝ *string*
Desired information.
Default: 'abstract'
List of values: `Slot` ∈ {'short', 'abstract', 'chapter', 'chapter_id', 'functionality', 'parallelization', 'execution information', 'parallel_method', 'interrupt_mode', 'compute_device', 'complexity', 'predecessor', 'successor', 'alternatives', 'parameter', 'see_also', 'keywords', 'example', 'attention', 'result_state', 'references', 'module', 'html_path', 'warning', 'dynamic_modules'}

▷ **`Information`** (output_control) ..... string(-array) ⇝ *string*
Information (empty if no information is available)

Result

The operator `get_operator_info` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct and the helpfiles are availabe. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_keywords, search_operator, get_operator_name, query_operator_info, query_param_info, get_param_info

Possible Successors

get_param_names, get_param_num, get_param_types

Alternatives

get_param_names

See also

query_operator_info, get_param_info, get_operator_name, get_param_num, get_param_types

Module

Foundation

get_operator_name (: : Pattern : OperatorNames)

Get operators with the given string as a substring of their name.

The operator get_operator_name takes a string (Pattern) as input and searches all HALCON-operators having this string as a substring in their name. If an empty string is entered, the names of all operators available will be returned.

Parameters

▷ **Pattern** (input_control) ..... string ⇝ *string*
Substring of the seeked names (empty <=> all names).
Default: 'info'

▷ **OperatorNames** (output_control) ..... string-array ⇝ *string*
Detected operator names.

Result

The operator get_operator_name returns the value 2 (H_MSG_TRUE) if the helpfiles are available. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_operator_info, get_param_names, get_param_num, get_param_types

Alternatives

search_operator

See also

get_operator_info, get_param_names, get_param_num, get_param_types

Module

Foundation

get_param_info (: : OperatorName, ParamName, Slot : Information)

Get information concerning the operator parameters.

The operator get_param_info is used for calling up the online-texts assigned to a parameter of an indicated operator. The form of information available for each parameter (Slot), can be called up by using the operator query_param_info. Possible slots are given in the list below. For a more extensive description of the entries and their possible values we refer to the "Extension Package Programmer's Manual".

'description': Description of the parameter.

'parameter_class': Parameter classes.

'input_object'
'output_object'
'input_control'
'output_control'

'type_list': Permitted type(s) of data for parameter values.

'default_type': Default-type for parameter values (for control parameters only). This type of parameter is the one HALCON/C uses in the "simple mode". If *'none'* is indicated, the "tuple mode" must be used.

'sem_type': Semantic type of the parameter. This is important to allow the assignment of the parameters to object classes in object-oriented languages (C++, .NET, COM). If more than one parameter belongs semantically to one type, this fact is indicated as well.

'default_value': Default value for the parameter (for input control parameters only). It is the question of mere information only (the parameter value must be transmitted explicitly, even if the default value is used). This entry serves only as a notice, a point of departure for own experiments. The values have been selected so that they normally do not cause any errors but generate something that makes sense.

'modified':
'true', if the operator modifies this input parameter.

'multivalue':
'true', if an array of values must be passed,
'false', if exactly one value must be passed, and
'optional' allows both.

'multichannel':
'true', in case the input image object may be multichannel.

'multiinstance':
'true', if an array of semantic tuples (e.g., poses, camera parameters, etc.) must be passed,
'false', if exactly one semantic tuple must be passed, and
'optional' allows both.

'mixed_type': For control parameters exclusively and only if value tuples (*'multivalue'*-*'true'*/*'optional'*) and various types of data are permitted for the parameter values ('type_list' having more than one value). In this case `Slot` indicates, whether values of various types may be mixed in one tuple (*'true'* or *'false'*).

'values': Selection of values (optional).

'value_list': In case a parameter can take only a limited number of values, this fact will be indicated explicitly (optional).

'valuemin': Minimum value of a value interval.

'valuemax': Maximum value of a value interval.

'valuefunction': Function describing the course of the values for a series of tests (lin, log, quadr, ...).

'steprec': Recommended step width for the parameter values in a series of tests.

'stepmin': Minimum step width of the parameter values in a series of tests.

'valuenumber': Expression describing the number of parameters as such or in relation to other parameters.

'assertion': Expression describing the parameter values as such or in relation to other parameters.

'cd_type_list.[compute_device]': List of input image types the compute device implementation of the operator supports for a specific device (use 'cd_type_list.opencl' for OpenCL devices).

'cd_value_list.[compute_device]': List of input control parameters the compute device implementation of the operator supports for a specific device (use 'cd_value_list.opencl' for OpenCL devices).

The online-texts will be taken from the files `operators_[LANG].ref`, `operators_[LANG].sta`, `operators_[LANG].key`, `operators_[LANG].num` and `operators_[LANG].idx`, which will be searched by HALCON in the currently used directory or the directory 'help_dir' (see also `get_system` and `set_system`).

Attention

The encoding of the result is UTF-8.

Parameters

▷ **OperatorName** (input_control) ..... proc_name ⇝ *string*
Name of the operator on whose parameter more information is needed.
Default: 'get_param_info'

▷ **ParamName** (input_control) ..... string ⇝ *string*
Name of the parameter on which more information is needed.
Default: 'Slot'

▷ **Slot** (input_control) ..... string ⇝ *string*
Desired information.
Default: 'description'
List of values: `Slot` ∈ {'assertion', 'default_type', 'default_value', 'description', 'mixed_type', 'modified', 'multichannel', 'multivalue', 'sem_type', 'step_min', 'step_rec', 'type_list', 'value_function', 'value_list', 'value_max', 'value_min', 'value_number', 'values', 'cd_type_list.opencl', 'cd_value_list.opencl'}

▷ **Information** (output_control) ..... string(-array) ⇝ *string*
Information (empty in case there is no information available).

Result

The operator `get_param_info` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct and the help-files are available. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_keywords`, `search_operator`

Alternatives

`get_param_names`, `get_param_num`, `get_param_types`

See also

`query_param_info`, `get_operator_info`, `get_operator_name`

Module

Foundation

get_param_names (: : OperatorName : InpObjPar, OutpObjPar, InpCtrlPar, OutpCtrlPar)

Get the names of the parameters of a HALCON-operator.

For the HALCON-operator indicated in `OperatorName` the operator `get_param_names` returns the names of the input objects, the output objects, of the input control parameters and the output control parameters.

Parameters

▷ **OperatorName** (input_control) ..... proc_name ⇝ *string*
Name of the operator.
Default: 'get_param_names'

▷ **InpObjPar** (output_control) ..... string-array ⇝ *string*
Names of the input objects.

▷ **OutpObjPar** (output_control) ..... string-array ⇝ *string*
Names of the output objects.

▷ **InpCtrlPar** (output_control) ..... string-array ⇝ *string*
Names of the input control parameters.

▷ **OutpCtrlPar** (output_control) ..... string-array ⇝ *string*
Names of the output control parameters.

Result

The operator get_param_names returns the value 2 (H_MSG_TRUE) if the name of the operator exists and the helpfiles are available. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_keywords, search_operator, get_operator_name, get_operator_info

Possible Successors

get_param_num, get_param_types

Alternatives

get_operator_info, get_param_info

See also

get_param_num, get_param_types, get_operator_name

Module

Foundation

get_param_num (: : OperatorName : CName, InpObjPar, OutpObjPar, InpCtrlPar, OutpCtrlPar, Type)

Get number of the different parameter classes of a HALCON-operator.

The operator get_param_num returns the number of the input and output object parameters, as well as the input and output control parameters for the indicated HALCON-operator. Further, you will receive the name of the C-function (CName) called by the operator. The output parameter Type indicates, whether the operator is a system operator or an user procedure.

Parameters

▷ **OperatorName** (input_control) ..... proc_name ⇝ *string*
Name of the operator.
Default: 'get_param_num'

▷ **CName** (output_control) ..... string ⇝ *string*
Name of the called C-function.

▷ **InpObjPar** (output_control) ..... integer ⇝ *integer*
Number of the input object parameters.

▷ **OutpObjPar** (output_control) ..... integer ⇝ *integer*
Number of the output object parameters.

▷ **InpCtrlPar** (output_control) ..... integer ⇝ *integer*
Number of the input control parameters.

▷ **OutpCtrlPar** (output_control) ..... integer ⇝ *integer*
Number of the output control parameters.

▷ **Type** (output_control) ..... string ⇝ *string*
System operator or user procedure.
Suggested values: Type ∈ {'system', 'user'}

Result

The operator get_param_num returns the value 2 (H_MSG_TRUE) if the name of the operator exists. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

get_keywords, search_operator, get_operator_name, get_operator_info

Possible Successors

get_param_types

Alternatives

get_operator_info, get_param_info

See also

get_param_names, get_param_types, get_operator_name

Module

Foundation

get_param_types (: : OperatorName : InpCtrlParType, OutpCtrlParType)

Get default data type for the control parameters of a HALCON-operator.

The operator get_param_types returns the default data type for each input and output control parameter. The default type of a parameter is the type used in "simple mode" in HALCON/C. This concerns parameters which allow more than one type as for example write_string. Hereby the types of input parameters are combined in the variable InpCtrlParType, whereas the types of output parameters are combined in the variable OutpCtrlParType. The following types are possible:

'integer': an integer.

'integer tuple': an integer or a tuple of integers.

'real': a floating point number.

'real tuple': a floating point number or a tuple of floating point numbers.

'string': a string.

'string tuple': a string or a tuple of strings.

'no_default': individual value of which the type cannot be determined.

'no_default tuple': individual value or tuple of values of which the type cannot be determined.

'default': individual value of unknown type, whereby the systems assumes it to be an integer.

Parameters

▷ **OperatorName** (input_control) ..... proc_name ⇝ *string*
Name of the operator.
Default: 'get_param_types'

▷ **InpCtrlParType** (output_control) ..... string-array ⇝ *string*
Default type of the input control parameters.

▷ **OutpCtrlParType** (output_control) ..... string-array ⇝ *string*
Default type of the output control parameters.

Result

The operator get_param_types returns the value 2 (H_MSG_TRUE) if the parameters are correct and the helpfiles are availabe. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_keywords, search_operator, get_operator_name, get_operator_info

Alternatives

get_param_info

See also

get_param_names, get_param_num, get_operator_info, get_operator_name

Module

Foundation

query_operator_info (: : : Slots)

Query slots concerning information with relation to the operator get_operator_info.

The operator query_operator_info returns the names of those online texts (Slots) which are available online for each operator. The information itself can be called up using

get_operator_info(::<OperatorName>,<Slot>:<Information>).

Parameters

▷ **Slots** (output_control) ..... string-array ⇝ *string*
Slot names of the operator get_operator_info.

Result

The operator query_operator_info always returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_operator_info

See also

get_operator_info

Module

Foundation

query_param_info (: : : Slots)

Query slots of the online-information concerning the operator get_param_info.

The operator query_param_info returns the names of those pieces of information (Slots) which are available online for each parameter (online texts). The online texts themselves can be called up using

get_param_info(::<Procedurname>,<Parametername>,<Slot>:<Information>).

Parameters

▷ **Slots** (output_control) ..... string-array ⇝ *string*
Slot names for the operator get_param_info.

Result

query_param_info always returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_param_info

See also

get_param_info

Module

Foundation

search_operator (: : Keyword : OperatorNames)

Search names of all operators assigned to one keyword.

The operator search_operator returns the names of all operators whose online-texts include the keyword Keyword (see also get_operator_info). All available keywords are called by using the operator get_keywords(::": <keywords>). The online-texts are taken from the files operators_[LANG].ref, operators_[LANG].sta, operators_[LANG].key, operators_[LANG].num and operators_[LANG].idx, which are searched by HALCON in the currently used directory or the directory 'help_dir' (see also get_system and get_system).

Parameters

▷ **Keyword** (input_control) ..... string ⇝ *string*
Keyword for which corresponding operators are searched.
Default: 'Information'

▷ **OperatorNames** (output_control) ..... string-array ⇝ *string*
Operators whose slot 'keyword' contains the keyword.

Result

The operator search_operator returns the value 2 (H_MSG_TRUE) if the parameters are correct and the helpfiles are available. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_keywords

See also

get_keywords, get_operator_info, get_param_info

Module

Foundation

25.7 Memory Block

compare_memory_block (: : MemoryBlocks1,
MemoryBlocks2 : IsEqual)

Compare contents of memory blocks regarding equality.

The operator compare_memory_block compares the contents of two memory block tuples and returns the comparison result in IsEqual. Therefore, each memory block in the tuple MemoryBlocks1 is compared to the respective block in the tuple MemoryBlocks2 with the same index. IsEqual is set to TRUE if the number

of elements of the two tuples matches and if the length and byte content of each pair of memory blocks coincides, otherwise to FALSE.

Attention

Note that `compare_memory_block` compares the memory blocks regarding their actual byte content, not their location in memory.

Parameters

- ▷ **`MemoryBlocks1`** (input_control) ..... memory_block(-array) ⇝ *handle*
  Tuple of reference memory blocks.
- ▷ **`MemoryBlocks2`** (input_control) ..... memory_block(-array) ⇝ *handle*
  Tuple of test memory blocks.
- ▷ **`IsEqual`** (output_control) ..... integer ⇝ *integer*
  Boolean result value.

Result

The operator `compare_memory_block` returns the value `2` (`H_MSG_TRUE`) if the passed handles are valid. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_memory_block_extern`, `create_memory_block_extern_copy`, `image_to_memory_block`, `read_memory_block`

Module

Foundation

`create_memory_block_extern` `( : : Pointer, Size, FreeFunction : MemoryBlockHandle )`

Create a memory block from an external pointer.

It is not recommended to use the operator `create_memory_block_extern` in HDevelop.

`create_memory_block_extern` creates a memory block and returns its handle `MemoryBlockHandle`. `Pointer` is a data pointer to the beginning of the memory block. `Size` controls the size in bytes of the memory block. `FreeFunction` is an optional callback function that frees the memory pointed to by `Pointer`. This function must have the following signature

```
void FreeFunction(void* ptr);
```

and will be called using `__cdecl` calling convention when deleting `MemoryBlockHandle`. Hence, HALCON gains ownership over the memory in this case and releases the memory via the callback function. If the memory shall not be released when deleting `MemoryBlockHandle`, i.e., HALCON shall not own the memory, the NULL-Pointer can be passed.

Attention

This operator does not copy any data. If a copy is required `create_memory_block_extern_copy` can be used.

Parameters

- ▷ **`Pointer`** (input_control) ..... pointer ⇝ *integer*
  Data pointer of the memory block.
- ▷ **`Size`** (input_control) ..... integer ⇝ *integer*
  Size of the memory block.

▷ **FreeFunction** (input_control) . . . . . pointer ⇝ *integer*
Function to free the memory block.
Default: 0

▷ **MemoryBlockHandle** (output_control) . . . . . memory_block ⇝ *handle*
Handle of the memory block.

Result

If the parameters are valid, the operator `create_memory_block_extern` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`get_memory_block_ptr`, `compare_memory_block`, `memory_block_to_image`, `write_memory_block`

Alternatives

`create_memory_block_extern_copy`

Module

Foundation

create_memory_block_extern_copy (: : Pointer, Size : MemoryBlockHandle)

Create a memory block from an external pointer by copying.

`create_memory_block_extern_copy` creates a memory block by copying bytes and returns its handle `MemoryBlockHandle`. `Pointer` is a data pointer to the beginning of the memory to be copied. `Size` determines the number of bytes to be copied and with that the size of the memory block in bytes. Note that the memory copied by this operator is released when deleting `MemoryBlockHandle`, whereas the original memory in `Pointer` that is copied from is not released.

Parameters

▷ **Pointer** (input_control) . . . . . pointer ⇝ *integer*
Data pointer of the memory block.

▷ **Size** (input_control) . . . . . integer ⇝ *integer*
Size of the memory block.

▷ **MemoryBlockHandle** (output_control) . . . . . memory_block ⇝ *handle*
Handle of the memory block.

Result

If the parameters are valid, the operator `create_memory_block_extern_copy` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

get_memory_block_ptr, compare_memory_block, memory_block_to_image, write_memory_block

Alternatives

create_memory_block_extern

Module

Foundation

get_memory_block_ptr (: : MemoryBlockHandle : Pointer, Size)

Return the data pointer and size of a memory block.

It is not recommended to use the operator get_memory_block_ptr in HDevelop.

get_memory_block_ptr returns the data pointer to the beginning of a memory block in Pointer and its size in bytes in Size. MemoryBlockHandle is the handle of the memory block.

Parameters

- ▷ **MemoryBlockHandle** (input_control) ..... memory_block ⇝ *handle*
  Handle of the memory block.
- ▷ **Pointer** (output_control) ..... pointer ⇝ *integer*
  Data pointer to the beginning of the memory block.
- ▷ **Size** (output_control) ..... integer ⇝ *integer*
  Size of the memory block.

Result

If the parameters are valid, the operator get_memory_block_ptr returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_memory_block_extern, create_memory_block_extern_copy, image_to_memory_block, read_memory_block

Module

Foundation

read_memory_block (: : FileName : MemoryBlockHandle)

Read a memory block from a file.

The operator read_memory_block reads the file FileName and creates a new memory block in MemoryBlockHandle that is an identical copy of the bytes in the file.

The default HALCON file extension for memory block files is '.bin'.

Parameters

- ▷ **FileName** (input_control) ..... filename.read ⇝ *string*
  Name of the file.
  Default: 'memory_block.bin'
  File extension: .bin
- ▷ **MemoryBlockHandle** (output_control) ..... memory_block ⇝ *handle*
  Memory block handle.

Result

The operator read_memory_block returns the value 2 (H_MSG_TRUE) if the named file was found and correctly read. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

get_memory_block_ptr, compare_memory_block, memory_block_to_image

See also

write_memory_block

Module

Foundation

write_memory_block (: : MemoryBlockHandle, FileName :)

Write a memory block to a file.

The operator write_memory_block writes the byte content of the memory block MemoryBlockHandle to the file FileName. The memory block can be read again with read_memory_block.

The default HALCON file extension for memory block files is '.bin', but any file extension can be chosen, including none.

Attention

Choosing a file extension that does not fit the byte content of the memory block MemoryBlockHandle only changes the name of the file, not the bytes written to disk.

Parameters

▷ **MemoryBlockHandle** (input_control) ..... memory_block ⇝ *handle*
Memory block handle.

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Name of the file.
Default: 'memory_block.bin'
File extension: .bin

Result

The operator write_memory_block returns the value 2 (H_MSG_TRUE) if the passed handle is valid and if the memory block was successfully written into the named file. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_memory_block_extern, create_memory_block_extern_copy, image_to_memory_block

See also

read_memory_block

Module

Foundation

25.8 Multithreading

broadcast_condition (: : ConditionHandle :)

Signal a condition synchronization object.

broadcast_condition restarts all the threads that are waiting on the condition variable ConditionHandle. Nothing happens if no threads are waiting on ConditionHandle.

Parameters

▷ **ConditionHandle** (input_control) ..... condition ⇝ *handle*
Condition synchronization object.

Result

If the condition handle is valid, the operator broadcast_condition returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_condition, wait_condition

See also

wait_condition, timed_wait_condition

Module

Foundation

clear_barrier (: : BarrierHandle :)

Destroy a barrier synchronization object.

clear_barrier destroys the barrier object given in BarrierHandle. No threads should be waiting on the barrier at the time clear_barrier is called. After calling clear_barrier, the barrier can no longer be used. The handle BarrierHandle becomes invalid.

Parameters

▷ **BarrierHandle** (input_control) ..... barrier ⇝ *handle*
Barrier synchronization object.

Result

If the barrier handle is valid, the operator clear_barrier returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- BarrierHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_barrier

See also

create_barrier

Module

Foundation

clear_condition (: : ConditionHandle :)

Destroy a condition synchronization object.

clear_condition destroys the condition object given in ConditionHandle. No threads should be waiting on the condition at the time clear_condition is called. After calling clear_condition, the condition can no longer be used. The handle ConditionHandle becomes invalid.

Parameters

▷ **ConditionHandle** (input_control) ..... condition ⇝ *handle*
Condition synchronization object.

Result

If the condition handle is valid, the operator clear_condition returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ConditionHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_condition

See also

create_condition

Module

Foundation

clear_event (: : EventHandle :)

Clear the event synchronization object.

clear_event destroys the mutex object given by EventHandle. No threads should be waiting on the event at the time clear_event is called. After calling clear_event, the event can no longer be used. The handle EventHandle becomes invalid.

Parameters

▷ **EventHandle** (input_control) . . . . . event ⇝ *handle*
Event synchronization object.

Result

If the event handle is valid, the operator `clear_event` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- EventHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_event`

See also

`create_event`

Module

Foundation

clear_message (: : MessageHandle :)

Close a message handle and release all associated resources.

`clear_message` closes a message handle `MessageHandle` that was previously opened with `create_message` or `dequeue_message`. Any resources owned by the message, in particular the tuple or object data stored in the message, will be released.

Multiple message handles can be closed in a single `clear_message` call, passing them as a tuple to the `MessageHandle` parameter.

Attention

The handle(s) must not be used again after being invalidated using `clear_message`. Using an invalid handle results in undefined behavior.

Parameters

▷ **MessageHandle** (input_control) . . . . . message(-array) ⇝ *handle*
Message handle(s) to be closed.
Number of elements: MessageHandle >= 1
Restriction: MessageHandle != 0

Example

```
MessageHandles := []
for idx := 0 to 4 by 1
  create_message (MessageHandle)
  MessageHandles[idx]:= MessageHandle
endfor
* ...
```

Result

If the message handle(s) passed to the operator are valid, `clear_message` returns `2` (`H_MSG_TRUE`). Otherwise

an exception is raised. If a tuple of handles is passed and some of them are invalid, clear_message attempts to clear as many handles from the tuple as possible before reporting the error.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MessageHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_message, dequeue_message

See also

create_message, set_message_tuple, get_message_tuple, set_message_obj, get_message_obj, set_message_param, get_message_param, enqueue_message, dequeue_message

Module

Foundation

clear_message_queue (: : QueueHandle :)

Close a message queue handle and release all associated resources.

clear_message_queue closes a message queue handle QueueHandle that was previously opened with create_message_queue. Any resources owned by the message queue, in particular message data queued in the message queue will be released.

Multiple message queue handles can be closed in a single clear_message_queue call, passing them as a tuple to the QueueHandle parameter.

Attention

The handle(s) must not be used again after being invalidated using clear_message_queue. Using an invalid handle results in undefined behavior. Operator clear_message_queue must not be called while the handle is used concurrently from other threads, because this would result in undefined behavior.

Parameters

▷ **QueueHandle** (input_control) . . . . . message_queue(-array) ⇝ *handle*
Message queue handle(s) to be closed.
Number of elements: QueueHandle >= 1
Restriction: QueueHandle != 0

Example

```
create_message_queue (ProducerQueue)
create_message_queue (ResultQueue)
* ...
```

Result

If the message queue handle(s) passed to the operator are valid, clear_message_queue returns 2 (H_MSG_TRUE). Otherwise an exception is raised. If a tuple of handles is passed and some of them are invalid, clear_message_queue attempts to clear as many handles from the tuple as possible before reporting the error.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- QueueHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_message_queue, set_message_queue_param

See also

create_message_queue, enqueue_message, dequeue_message, set_message_queue_param, get_message_queue_param, create_message, clear_message, set_message_tuple, get_message_tuple, set_message_obj, get_message_obj

Module

Foundation

clear_mutex (: : MutexHandle :)

Clear the mutex synchronization object.

clear_mutex destroys the mutex object given by MutexHandle. The mutex must be unlocked. After calling clear_mutex, the mutex can no longer be used. The handle MutexHandle becomes invalid.

Parameters

▷ **MutexHandle** (input_control) ..... mutex ⇝ *handle*
Mutex synchronization object.

Result

If the mutex handle is valid, the operator clear_mutex returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MutexHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_mutex

See also

create_mutex

Module

Foundation

create_barrier (: : AttribName, AttribValue, TeamSize : BarrierHandle)

Create a barrier synchronization object.

A barrier is a synchronization object blocking a thread until a previous defined number of threads have reached this barrier. `create_barrier` allocates and initializes the barrier object returned in `BarrierHandle` according to the attributes specified in `AttribName` and `AttribValue`. `AttribName` specifies the attribute class and `AttribValue` the kind of the barrier. The description beneath lists all barrier kinds supported by following parameter classes:

'' empty string sets the default attributes.

'type' specifies what happens if a thread waits at a barrier:

'sleep' simply suspends the calling thread (default).

'spin' a fast barrier implementation for short waits that waits busy.

The `TeamSize` argument specifies the number of threads that must call `wait_barrier` before any of them successfully return from waiting. The value hold in `TeamSize` must be greater than zero.

Parameters

▷ **`AttribName`** (input_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
Barrier attribute.
Default: []
List of values: `AttribName` ∈ {'type'}

▷ **`AttribValue`** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Barrier attribute value.
Number of elements: AttribValue == AttribName
Default: []
List of values: `AttribValue` ∈ {'spin', 'sleep'}

▷ **`TeamSize`** (input_control) . . . . . number ⇝ *integer*
Barrier team size.
Default: 1

▷ **`BarrierHandle`** (output_control) . . . . . barrier ⇝ *handle*
Barrier synchronization object.

Result

`create_barrier` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`wait_barrier`, `clear_barrier`

Module

Foundation

`create_condition ( : : AttribName, AttribValue : ConditionHandle )`

Create a condition variable synchronization object.

A condition variable (short: "condition") is a synchronization device that allows threads to suspend execution and relinquish the processors until some predicate on shared data is satisfied. The basic operations on conditions are: signal the condition (when the predicate becomes true), and wait for the condition, suspending the thread execution until another thread signals the condition.

A condition variable must always be associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

create_condition creates and initializes the condition variable ConditionHandle, using the condition attributes specified in AttribName and AttribValue, or default attributes if AttribName is ''. The current implementation supports no attributes for conditions, hence the AttribName parameter must be set to ''.

Parameters

▷ **AttribName** (input_control) . . . . . number(-array) ⇝ *string* / integer / real
Mutex attribute.
Default: []

▷ **AttribValue** (input_control) . . . . . number(-array) ⇝ *string* / integer / real
Mutex attribute value.
Number of elements: AttribValue == AttribName
Default: []

▷ **ConditionHandle** (output_control) . . . . . condition ⇝ *handle*
Condition synchronization object.

Result

create_condition returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

wait_condition, timed_wait_condition, signal_condition

See also

clear_condition

Module

Foundation

create_event (: : AttribName, AttribValue : EventHandle)

Create an event synchronization object.

An event is a prelocked mutual exclusion that does not belong to a thread, i.e., it can be signaled by a thread different from the locking thread. This understanding is, e.g., close to the definition of an event in WIN32-API or a semaphore with count 0 in POSIX threads.

create_event allocates and initializes the event object according to the attributes specified in AttribName and AttribValue. AttribName specifies the attribute class and AttribValue the kind of the event. The description beneath lists all mutex kinds supported by following classes:

'' empty string sets the default attributes.

'type' specifies what happens if a thread waits on a signal to an event:

'sleep' simply suspends the calling thread (default).

'spin' a fast event implementation that waits busy on a signal. This type is non recursive.

Parameters

▷ **AttribName** (input_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
Mutex attribute.
Default: []
List of values: AttribName ∈ {'type'}

▷ **AttribValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Mutex attribute value.
Number of elements: AttribValue == AttribName
Default: []
List of values: AttribValue ∈ {'spin', 'sleep'}

▷ **EventHandle** (output_control) . . . . . event ⇝ *handle*
Event synchronization object.

Result

`create_event` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`wait_event`, `signal_event`, `clear_event`

See also

`clear_event`

Module

Foundation

create_message (: : : MessageHandle)

Create a new empty message.

`create_message` creates a new empty message. The output parameter `MessageHandle` is a handle to the newly created message and is used to identify the message in any subsequent operator call using the message.

The message serves as a dictionary-like container that can be passed between the threads of an application using asynchronous message queues. Alternatively, HALCON also provides dictionaries (`create_dict`) to group parameters in programs and procedures.

Messages can store an arbitrary number of entries, each having its unique key (string or integer) and associated value. Each key can refer either to a control parameter tuple or to an iconic object. These data are stored to the message using `set_message_tuple` or `set_message_obj`, respectively, from where they can be retrieved again using `get_message_tuple` or `get_message_obj`.

The control parameter tuples stored in the message are always deep copies of the original data. The original data can thus be reused immediately after the `set_message_tuple` call without affecting the message. Notable exceptions are handles: Storing any handle in the message will copy the handle value, but not the resource behind the handle.

As mentioned above, the messages can be passed between the threads of an application using asynchronous message queues. The data can be appended to the queue by multiple producer threads using `enqueue_message` and retrieved from the queue by multiple receiver threads using `dequeue_message`. All these operations are internally properly synchronized. Therefore, the queue can be safely accessed by all producers and consumers without any explicit locking. All the enqueued messages are copied by the `enqueue_message` operation. The original message(s) can thus be immediately reused after the `enqueue_message` call without affecting the enqueued copy.

Parameters

▷ **MessageHandle** (output_control) . . . . . message ⇝ *handle*
Handle of the newly created message.
Number of elements: MessageHandle == 1
Assertion: MessageHandle != 0

Example

```
MessageHandles := []
for idx := 0 to 4 by 1
  create_message (MessageHandle)
  MessageHandles[idx]:= MessageHandle
endfor
* ...
```

Result

Returns 2 (H_MSG_TRUE) unless a resource allocation error occurs.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

set_message_tuple, set_message_obj

Alternatives

create_dict

See also

clear_message, set_message_tuple, get_message_tuple, set_message_obj, get_message_obj, set_message_param, get_message_param, enqueue_message, dequeue_message

Module

Foundation

create_message_queue (: : : QueueHandle)

Create a new empty message queue.

create_message_queue creates a new empty message queue. The output parameter QueueHandle is a handle to the newly created message queue and is used to identify the queue in any subsequent operator calls using the queue.

The message queues are designed as FIFO pipes delivering arbitrary sets of data safely among different threads. The queue access is internally fully synchronized, no explicit locking is required from the application. The data is traveling through the queue in so called messages (see create_message).

Multiple producer threads can append data simultaneously (enqueue_message) while multiple consumer threads are simultaneously retrieving the data again (dequeue_message). Multiple messages can be enqueued together using a single enqueue_message operation. In such case, those messages will travel together through the queue and will be delivered through a single dequeue_message call.

All the enqueued messages are copied by the enqueue_message operation. The original message(s) can thus be immediately reused after the enqueue_message call without affecting the enqueued copy.

Parameters

▷ **QueueHandle** (output_control) . . . . . message_queue ⇝ *handle*
Handle of the newly created message queue.
Number of elements: QueueHandle == 1
Assertion: QueueHandle != 0

Example

```
create_message_queue (ProducerQueue)
create_message_queue (ResultQueue)
* ...
```

Result

Returns 2 (H_MSG_TRUE) unless a resource allocation error occurs. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Successors

enqueue_message, dequeue_message

See also

clear_message_queue, enqueue_message, dequeue_message, set_message_queue_param, get_message_queue_param, create_message, clear_message, set_message_tuple, get_message_tuple, set_message_obj, get_message_obj

Module

Foundation

create_mutex (: : AttribName, AttribValue : MutexHandle)

Create a mutual exclusion synchronization object.

A mutex is a synchronization object that negotiates mutual exclusion among threads of a single process to avoid simultaneous modifications on common resources such as global variables. A mutex has two possible states: unlocked, i.e., not owned by any thread, and locked, i.e., owned by one certain thread. Threads attempting to lock a mutex that is already owned by another thread, i.e., locked, wait until the owning thread unlocks the mutex first. Unlike events, a thread has to own a mutex to unlock it.

create_mutex allocates and initializes the mutex object according to the attributes specified in AttribName and AttribValue. AttribName specifies the attribute class and AttribValue the kind of the mutex. The description beneath lists all mutex kinds supported by following classes:

'' empty string sets the default attributes.

'type' specifies what happens if a thread attempts to lock a mutex that is already owned:

'sleep' simply suspends the calling thread (default). The behavior is undefined, if the thread is already owner of the mutex.

'spin' a fast mutex implementation that is busy waiting for the signaled mutex. The behavior is undefined, if the thread is already owner of the mutex.

'recursive' suspends the calling thread if the thread is not owner of the mutex. But a recursive mutex can be acquired again by the owning thread. A recursive mutex does not become unlocked until the number of unlock requests equals the number of successful lock requests.

The mutex is unlocked when created.

Parameters

▷ **AttribName** (input_control) . . . . . number(-array) ⇝ *string* / integer / real
Mutex attribute class.
Default: []
List of values: AttribName ∈ {'type'}

▷ **AttribValue** (input_control) . . . . . number(-array) ⇝ *string* / integer / real
Mutex attribute kind.
Number of elements: AttribValue == AttribName
Default: []
List of values: AttribValue ∈ {'spin', 'sleep', 'recursive'}

▷ **MutexHandle** (output_control) . . . . . mutex ⇝ *handle*
Mutex synchronization object.

Result

create_mutex returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

lock_mutex, clear_mutex

See also

clear_mutex

Module

Foundation

dequeue_message (: : QueueHandle, GenParamName, GenParamValue : MessageHandle)

Receive one or more messages from the message queue.

dequeue_message dequeues a message from the message queue denoted by the QueueHandle parameter. The message must have been enqueued by any thread using enqueue_message.

The messages are delivered in FIFO (first-in first-out) order, every message is delivered only once. If the queue is not empty, dequeue_message immediately delivers the oldest message from the queue. This message is removed from the queue and a handle to the message is returned in the MessageHandle output parameter. The message data ownership is transferred (no copy) from the message queue to the newly created message handle.

If the queue is empty, dequeue_message blocks until a message becomes available for delivery (being queued from another thread using enqueue_message).

The data stored in the message can be queried using get_message_tuple, get_message_obj, or get_message_param.

The message handles obtained through dequeue_message can be further reused (modified and/or enqueued to another message queue).

If multiple messages were enqueued using a single enqueue_message call, all those messages will be also retrieved together through a single dequeue_message call, delivering multiple message handles via the MessageHandle tuple.

The queue access is internally fully synchronized, no external locking is required.

During application reconfiguration or cleanup, it might be necessary to wake threads waiting for messages in dequeue_message. This can be achieved using the operator set_message_queue_param with the parameter *'abort_dequeuing'*. In such case the currently blocked dequeue_message calls return immediately with H_ERR_MQCNCL.

Finally, it is possible to adjust dequeue_message behavior through generic parameters within GenParamName and GenParamValue. Currently, just a single generic parameter is supported:

'timeout': Timeout controlling how long the operator will block while waiting for a message if the queue is empty. When expired, the operator returns with H_ERR_TIMEOUT. The timeout can be specified as an integer or double value in seconds or as the string *'infinite'*. When no timeout is specified, infinite timeout is used as default, meaning that the operator would block until a new message is enqueued or until the operation is aborted.

Parameters

▷ **QueueHandle** (input_control) . . . . . message_queue ⇝ *handle*
Message queue handle.
Number of elements: QueueHandle == 1
Restriction: QueueHandle != 0

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Names of optional generic parameters
Number of elements: GenParamName == GenParamValue
Default: 'timeout'
List of values: GenParamName ∈ {'timeout'}

▷ **GenParamValue** (input_control) . . . . . tuple(-array) ⇝ *string* / integer / real
Values of optional generic parameters
Number of elements: GenParamName == GenParamValue
Default: 'infinite'
List of values: GenParamValue ∈ {'infinite'}

▷ **MessageHandle** (output_control) . . . . . message(-array) ⇝ *handle*
Handle(s) of the dequeued message(s).
Number of elements: MessageHandle > 0
Assertion: MessageHandle != 0

Example

```
create_message_queue (Queue)
* ...
dequeue_message (Queue, [], [], Message)
get_message_obj (Image, Message, 'my_image')
```

Result

If the operation succeeds, `dequeue_message` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised. Possible error conditions include invalid parameters, message dequeue wait timeout (H_ERR_TIMEOUT) or message dequeue wait abortion (H_ERR_MQCNCL). Note that in some rare cases, when an error occurs when finishing the operator call (after the message has already been removed from the queue), the operator will attempt to put the message back at the head of the queue to not lose it. This might lead to a different message delivery order if another thread removed another message from the queue in the meanwhile.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`get_message_tuple`, `get_message_obj`, `get_message_param`

See also

`create_message_queue`, `clear_message_queue`, `enqueue_message`, `set_message_queue_param`, `get_message_queue_param`, `create_message`, `clear_message`, `set_message_tuple`, `get_message_tuple`, `set_message_obj`, `get_message_obj`

Module

Foundation

enqueue_message (: : QueueHandle, MessageHandle, GenParamName, GenParamValue :)

Enqueue one or more messages to the message queue.

`enqueue_message` enqueues one or more messages to the message queue denoted by the `QueueHandle` parameter. The enqueued messages can be retrieved from the queue by any thread using `dequeue_message`.

The messages (see `create_message`) to be enqueued are specified in the `MessageHandle` parameter which accepts a single handle as well as tuple of handles. When enqueuing multiple handles together in a single `enqueue_message` call, they are delivered again together through a single `dequeue_message` call.

Multiple producer (enqueuing) threads and multiple consumer (dequeuing) threads can share the same queue at a time. The messages are delivered in FIFO (first-in first-out) order. Every message is delivered only once, even if multiple consumer threads are using the queue.

The queue access is internally fully synchronized, no external locking is required. If the queue is empty and there is at least one consumer thread waiting for the message data in `dequeue_message`, one of those threads is woken up by a successful `enqueue_message` call and the enqueued message data are immediately delivered. Otherwise the message data is appended to the queue asynchronously, to be delivered as soon as a consumer thread is ready to dequeue the message data again.

All the enqueued messages (MessageHandle) are copied by the enqueue_message operation. The original message(s) can thus be immediately reused after the enqueue_message call without affecting the enqueued copy.

Operator parameters GenParamName and GenParamValue are reserved in the operator's interface for future use, currently no generic parameters are supported.

If, after the operation, the queue would hold a larger number of messages than the maximum number specified with parameter value *'max_message_num'* of operator set_message_queue_param, the call would fail with H_ERR_MQOVL.

Parameters

▷ **QueueHandle** (input_control) . . . . . message_queue ⇝ *handle*
Message queue handle.
Number of elements: QueueHandle == 1
Restriction: QueueHandle != 0

▷ **MessageHandle** (input_control) . . . . . message(-array) ⇝ *handle*
Handle(s) of message(s) to be enqueued.
Number of elements: MessageHandle > 0
Restriction: MessageHandle != 0

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Names of optional generic parameters.
Number of elements: GenParamName == GenParamValue

▷ **GenParamValue** (input_control) . . . . . tuple-array ⇝ *string* / integer / real
Values of optional generic parameters.
Number of elements: GenParamName == GenParamValue

Example

```
create_message_queue (Queue)
* ...
create_message (Message)
set_message_tuple (Message, 'mixed_tuple', ['The answer', 42])
enqueue_message (Queue, Message, [], [])
```

Result

If the operation succeeds, enqueue_message returns 2 (H_MSG_TRUE). Otherwise an exception is raised. Possible error conditions include invalid parameters or resource allocation error.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

set_message_tuple, set_message_obj

See also

create_message_queue, clear_message_queue, dequeue_message, set_message_queue_param, get_message_queue_param, create_message, clear_message, set_message_tuple, get_message_tuple, set_message_obj, get_message_obj

Module

Foundation

get_current_hthread_id (: : : HThreadID)

Return the HALCON thread ID of the current thread.

get_current_hthread_id returns a thread ID that represents the currently running thread in HThreadID. This ID is identical to the windows thread ID or the POSIX thread ID on UNIX platforms and is also returned by the par_start operation of the HDevEngine. It can later be used to interrupt operators that are executed in this thread using interrupt_operator.

Parameters

▷ **HThreadID** (output_control) . . . . . integer ⇝ *integer*
ID representing the current thread.

Example

```
global tuple HThreadID
get_current_hthread_id (HThreadID)
* call some slow operator...

* In a different thread
wait_seconds(2)
* Interrupt the long-running operator in the other thread
interrupt_operator (HThreadID, 'break')
```

Result

If the current thread can be determined, get_current_hthread_id returns 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

interrupt_operator

See also

interrupt_operator, set_operator_timeout

Module

Foundation

get_message_obj (: ObjectData : MessageHandle, Key :)

Retrieve an object associated with the key from the message.

get_message_obj retrieves an object associated with the Key from the message denoted by the MessageHandle. The object has to be previously stored to the message using set_message_obj.

The operator returns the data in the parameter ObjectData. The iconic object is copied by the operation. Therefore, clearing or reusing the message object afterwards will not have any side-effect on the returned iconic object, afterwards.

If the given Key is not present in the message or if the data associated with the key is not an object, get_message_obj fails. Presence of keys and information about the data associated with the key can be verified using get_message_param.

Parameters

▷ **ObjectData** (output_object) . . . . . object(-array) ⇝ *object*
Tuple value retrieved from the message.

▷ **MessageHandle** (input_control) . . . . . message ⇝ *handle*
Message handle.
Number of elements: MessageHandle == 1
Restriction: MessageHandle != 0

▷ **Key** (input_control) . . . . . string ⇝ *string* / integer
Key string or integer.
Number of elements: Key == 1

Example

```
* ...
get_message_param (Message,'key_exists',['simple_string','foo','my_image'],\
                   KeysPresence)
get_message_param (Message,'key_data_type',['simple_string','my_image'],\
                   KeysType)
get_message_obj (Image, Message, 'my_image')
```

Result

If the operation succeeds, `get_message_obj` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised. Possible error conditions include invalid parameters (handle or key), the required key not found in the message or other than object data associated with given key.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`dequeue_message`

Alternatives

`get_message_tuple`

See also

`create_message`, `clear_message`, `set_message_tuple`, `get_message_tuple`, `set_message_obj`, `set_message_param`, `get_message_param`, `enqueue_message`, `dequeue_message`

Module

Foundation

`get_message_param` (: : `MessageHandle`, `GenParamName`, `Key` : `GenParamValue`)

Query message parameters or information about the message.

`get_message_param` queries current values of message parameters or other information about the message status.

With each call of `get_message_param`, only a single parameter value can be queried. However, there are two types of parameters/queries:

- Parameters/queries applicable to the entire message. In this case no keys must be specified, the parameter `Key` must be an empty tuple.
- Parameters/queries applicable to the individual keys. In this case a non-empty list of keys must be specified in the parameter `Key`. The keys are processed in the same order as specified in the `Key` parameter.

Key-independent (global) parameter names:

'message_keys': Queries all the keys stored in the message, no matter whether they are associated with tuple or object data. The list of keys is reported as a string tuple via `GenParamValue`. For this query the parameter `Key` must be an empty tuple.

Currently supported key-specific parameter names are:

'key_exists': Reports *1* if the given key is stored in the message, *0* otherwise. The results are reported via GenParamValue, one value for each key.

'key_data_type': Reports *'tuple'* for keys associated with tuple data within the message (the data can be retrieved using get_message_tuple). Reports *'object'* for keys associated with object data (the data can be retrieved using get_message_obj). The results are reported via GenParamValue, one value for each key. This parameter is useful to decide dynamically whether to use get_message_tuple or get_message_obj to get the data of a specific key.

Parameters

▷ **MessageHandle** (input_control) . . . . . message ⇝ *handle*
Message handle.
Number of elements: MessageHandle == 1
Restriction: MessageHandle != 0

▷ **GenParamName** (input_control) . . . . . string ⇝ *string*
Names of the message parameters or info queries.
Number of elements: GenParamName == GenParamValue
Default: 'message_keys'
List of values: GenParamName ∈ {'message_keys', 'key_exists', 'key_data_type'}

▷ **Key** (input_control) . . . . . string(-array) ⇝ *string* / integer
Message keys the parameter/query should be applied to.

▷ **GenParamValue** (output_control) . . . . . tuple(-array) ⇝ *string* / integer / real
Values of the message parameters or info queries.

Example

```
get_message_param (Message, 'message_keys', [], AllKeys)
get_message_param (Message, 'key_data_type', AllKeys, KeysType)
```

Result

If all the operator parameters, and the specified keys are valid, get_message_param returns 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

dequeue_message

See also

create_message, clear_message, set_message_tuple, get_message_tuple, set_message_obj, get_message_obj, set_message_param, enqueue_message, dequeue_message

Module

Foundation

get_message_queue_param (: : QueueHandle, GenParamName : GenParamValue)

Query message queue parameters or information about the queue.

get_message_queue_param queries current values of message queue parameters or other information about the queue status.

Multiple queries can be carried out with a single get_message_queue_param call, passing multiple parameter names to parameter GenParamName. The parameter values are returned in GenParamValue in the same order as the parameter names requested by the caller.

Currently, following parameter names are supported:

'is_empty': Returns *1* if the queue is empty, *0* otherwise.

'message_num': Returns the number of messages currently stored in the queue.

'max_message_num': Returns the current value of the *'max_message_num'* parameter of the message queue, as it was set using `set_message_queue_param`. The default value *-1* stands for no limit.

Parameters

▷ **`QueueHandle`** (input_control) . . . . . message_queue ⇝ *handle*
Message queue handle.
Number of elements: QueueHandle == 1
Restriction: QueueHandle != 0

▷ **`GenParamName`** (input_control) . . . . . string(-array) ⇝ *string*
Names of the queue parameters or info queries.
Number of elements: GenParamName == GenParamValue
Default: 'max_message_num'
List of values: `GenParamName` ∈ {'is_empty', 'message_num', 'max_message_num'}

▷ **`GenParamValue`** (output_control) . . . . . tuple(-array) ⇝ *string* / integer / real
Values of the queue parameters or info queries.
Number of elements: GenParamName == GenParamValue

Example

```
create_message_queue (QueueHandle)
set_message_queue_param ( QueueHandle, 'max_message_num', 10)
* ...
get_message_queue_param( QueueHandle, 'message_num', Num)
```

Result

If all the operator parameters are valid, `get_message_queue_param` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_message_queue`, `enqueue_message`, `dequeue_message`

See also

`create_message_queue`, `clear_message_queue`, `enqueue_message`, `dequeue_message`, `set_message_queue_param`, `create_message`, `clear_message`, `set_message_tuple`, `get_message_tuple`, `set_message_obj`, `get_message_obj`

Module

Foundation

`get_message_tuple` (: : MessageHandle, Key : TupleData)

Retrieve a tuple associated with the key from the message.

`get_message_tuple` retrieves a tuple associated with the `Key` from the message denoted by the `MessageHandle`. The tuple has to be previously stored to the message using `set_message_tuple`.

The operator returns the data in the parameter `TupleData`. The data including strings is copied by the operation, the message can thus be immediately reused.

If the given `Key` is not present in the message or if the data associated with the key is not a tuple, `get_message_tuple` fails. Presence of keys and information about the data associated with the key can be verified using `get_message_param`.

Parameters

▷ **MessageHandle** (input_control) . . . . . message ⇝ *handle*
Message handle.
Number of elements: MessageHandle == 1
Restriction: MessageHandle != 0

▷ **Key** (input_control) . . . . . string ⇝ *string* / integer
Key string or integer.
Number of elements: Key == 1

▷ **TupleData** (output_control) . . . . . tuple(-array) ⇝ *string* / integer / real / handle
Tuple value retrieved from the message.

Example

```
* ...
get_message_param (Message,'key_exists',['simple_string','foo','my_image'],\
                   KeysPresence)
get_message_param (Message,'key_data_type',['simple_string','my_image'],\
                   KeysType)
get_message_tuple (Message,'simple_string',TupleString)
```

Result

If the operation succeeds, `get_message_tuple` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised. Possible error conditions include invalid parameters (handle or key), the required key not found in the message, or other than tuple data associated with given key.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`dequeue_message`

Alternatives

`get_message_obj`

See also

`create_message`, `clear_message`, `set_message_tuple`, `set_message_obj`, `get_message_obj`, `set_message_param`, `get_message_param`, `enqueue_message`, `dequeue_message`

Module

Foundation

get_threading_attrib (: : ThreadingHandle : ThreadingClass, AttribName, AttribValue)

Query the attributes of a threading / synchronization object.

`get_threading_attrib` determines the type of threading object passed to `ThreadingHandle` and the attributes the threading object was created with.

Parameters

▷ **ThreadingHandle** (input_control) . . . . . number ⇝ *integer*
Threading object.

▷ **ThreadingClass** (output_control) . . . . . string-array ⇝ *string*
Class name of threading object.

▷ **AttribName** (output_control) . . . . . string-array ⇝ *string*
Name of an attribute.

▷ **AttribValue** (output_control) . . . . . number-array ⇝ *string* / integer / real
Value of the attribute.

Result

If the threading handle is valid, the operator `get_threading_attrib` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`create_mutex`, `create_event`, `create_condition`, `create_barrier`

See also

`create_mutex`, `create_event`, `create_condition`, `create_barrier`

Module

Foundation

interrupt_operator (: : HThreadID, Mode :)

Attempt to interrupt an operator running in a different thread.

`interrupt_operator` attempts to gracefully interrupt the operator currently running in the HALCON thread represented by `HThreadID`. If that currently running operator supports interruption, it will abort its execution depending on `Mode` (see below). If the thread is currently not executing any operator, or if the currently running operator does not support interruptions, `interrupt_operator` is a no-op. The execution of any future operator in the other thread is not affected.

The ID representing a thread can be obtained with `get_current_hthread_id`.

`interrupt_operator` supports currently only one type of interruption, which can be set in `Mode`. Note that the corresponding type must also be supported by the operator currently running in the other thread. To determine if an operator is interruptible and which kind of interrupts it supports, please see the "Execution Information" section of the operator documentation or query the information using the parameter *'interrupt_mode'* of the operator `get_operator_info`.

Some operators, for which no interruptibility is mentioned in the reference documentation, can still be interrupted at some points of their execution. However, such interruptabilities are not guaranteed and can change without further notice between different versions of HALCON.

'cancel': Abort the execution of the operator. All results computed by the operator are discarded and the operator returns the error code H_ERR_CANCEL (22).

Attention

Note that not all operators support interruption. If a given operator supports interruptions and which modes are supported is described in the execution information section of the reference documentation of the corresponding operator.

Also note that there is no hard guarantee about the granularity of the interruption. The granularity can depend on the operator, its input data and the speed of the device. It is typically finer than 10 ms.

Parameters

▷ **HThreadID** (input_control) . . . . . integer ⇝ *integer*
Thread that runs the operator to interrupt.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Interruption mode.
Default: 'cancel'
List of values: `Mode` ∈ {'cancel'}

Example

```
global tuple HThreadID
get_current_hthread_id (HThreadID)
* call some slow operator...

* In a different thread
wait_seconds(2)
* Interrupt the long-running operator in the other thread
interrupt_operator (HThreadID, 'cancel')
```

Result

If all the operator parameters are valid, `interrupt_operator` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_current_hthread_id`

Alternatives

`set_operator_timeout`

See also

`get_current_hthread_id`, `set_operator_timeout`

Module

Foundation

lock_mutex (: : MutexHandle :)

Lock a mutex synchronization object.

`lock_mutex` locks the mutex given by `MutexHandle`. If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and `lock_mutex` returns immediately. If the mutex is already locked by another thread, the calling thread waits until the mutex is unlocked. The kind of wait is defined by the mutex' attributes set during creation in `create_mutex`.

Parameters

▷ **MutexHandle** (input_control) . . . . . mutex ⇝ *handle*
Mutex synchronization object.

Result

If the mutex handle is valid, the operator `lock_mutex` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_mutex`

Possible Successors

`unlock_mutex`

See also

try_lock_mutex

Module

Foundation

read_message (: : FileName, GenParamName, GenParamValue : MessageHandle)

Read a message from a file.

read_message reads a message that has been stored with write_message. The default HALCON file extension for a message is 'hmsg'.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
File name.
File extension: .hmsg

▷ **GenParamName** (input_control) ..... attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: GenParamName ∈ {'name'}

▷ **GenParamValue** (input_control) ..... attribute.name(-array) ⇝ *string* / integer / real
Value of the generic parameter.
Default: []

▷ **MessageHandle** (output_control) ..... message ⇝ *handle*
Message handle.
Number of elements: MessageHandle == 1
Assertion: MessageHandle != 0

Result

If the parameters are valid, the operator read_message returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

write_message

See also

write_message, serialize_handle, deserialize_handle

Module

Foundation

set_message_obj (ObjectData : : MessageHandle, Key :)

Add a key/object pair to the message.

set_message_obj stores an object associated with a key in the message, which behaves as a dictionary-like data container. The message is denoted by the MessageHandle parameter.

ObjectData is copied by the operation (copying the object data in HALCON's object database, see copy_obj), and can thus be immediately reused. Both an empty object or an object tuple are considered as a valid value that

can be associated with the key. If any data (tuple or object) was already associated with given key (Key), the old data is destroyed by set_message_obj and replaced by ObjectData.

The Key has to be a string or an integer. String keys are treated case sensitive.

The object data for the given key can be retrieved again from the message using get_message_obj.

Parameters

▷ **ObjectData** (input_object) ..... object(-array) ⇝ *object*
Object to be associated with the key.

▷ **MessageHandle** (input_control) ..... message ⇝ *handle*
Message handle.
Number of elements: MessageHandle == 1
Restriction: MessageHandle != 0

▷ **Key** (input_control) ..... string ⇝ *string* / integer
Key string or integer.
Number of elements: Key == 1

Example

```
create_message (Message)
read_image( Image, 'filename')
set_message_obj (Image, Message, 'my_image')
```

Result

If the operation succeeds, set_message_tuple returns 2 (H_MSG_TRUE). Otherwise an exception is raised. Possible error conditions include invalid parameters (handle or key) or resource allocation error.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MessageHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_message

Possible Successors

enqueue_message, set_message_tuple

Alternatives

set_message_tuple

See also

create_message, clear_message, set_message_tuple, get_message_tuple, get_message_obj, set_message_param, get_message_param, enqueue_message, dequeue_message

Module

Foundation

set_message_param (: : MessageHandle, GenParamName, Key, GenParamValue :)

Set message parameter or invoke commands on the message.

set_message_param sets message parameters or invokes action commands on the message.

For each call of set_message_param, only a single parameter can be set. However, there are two types of parameters/commands:

- Parameters/commands applicable to the entire message. In this case no keys must be specified, i.e., the parameter Key must be an empty tuple.
- Parameters/commands applicable to the individual keys. In this case a non-empty list of keys must be specified in parameter Key. The keys are processed in the same order as specified in the Key parameter.

Key-independent parameter names:

'remove_all_keys': Removes all the keys currently stored in the message and releases all the (control or iconic) data associated with those keys. The operation results in an empty message. Note that contents of GenParamValue is ignored for this parameter, while Key must be an empty tuple.

Key-specific parameter names:

'remove_key': Removes the keys specified in the Key parameter and releases all the (tuple or object) data associated with those keys. If an error occurs while processing one or more keys (in particular if the key is invalid), the operator attempts to continue removing as many keys as possible before reporting the failure. Note that the contents of GenParamValue is ignored for this parameter.

Parameters

▷ **MessageHandle** (input_control) . . . . . message ⇝ *handle*
Message handle.
Number of elements: MessageHandle == 1
Restriction: MessageHandle != 0

▷ **GenParamName** (input_control) . . . . . string ⇝ *string*
Names of the message parameters or action commands.
Number of elements: GenParamName == 1
Default: 'remove_key'
List of values: GenParamName ∈ {'remove_key', 'remove_all_keys'}

▷ **Key** (input_control) . . . . . string(-array) ⇝ *string* / integer
Message keys the parameter/command should be applied to.

▷ **GenParamValue** (input_control) . . . . . tuple(-array) ⇝ *string* / integer / real
Values of the message parameters or action commands.

Example

```
* Remove some keys
set_message_param (Message, 'remove_key', ['my_image', 'simple_string'], [])
```

Result

If all the operator parameters, and their values, as well as specified keys are valid, set_message_param returns 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MessageHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

dequeue_message, create_message

Possible Successors

enqueue_message, clear_message

See also

create_message, clear_message, set_message_tuple, get_message_tuple, set_message_obj, get_message_obj, get_message_param, enqueue_message, dequeue_message

Module

Foundation

set_message_queue_param (: : QueueHandle, GenParamName, GenParamValue :)

Set message queue parameters or invoke commands on the queue.

set_message_queue_param sets message queue parameters or invokes action commands on the queue.

Multiple parameters and action commands can be passed through a single set_message_queue_param call, passing multiple parameter names in a tuple GenParamName and corresponding values in GenParamValue. There must be always the same number of names and values and the values must be in the same order as the names. The parameters and action commands are applied in the same order passed in GenParamName. It is important to understand that in particular in case of the action commands, it might not be possible to revert the effect of previous action commands if a later command fails, the caller therefore might need to be careful when setting multiple commands in a single call.

Currently supported parameter names are:

'abort_dequeuing': Aborts any dequeue_message operator waiting in any thread for data to be delivered. Any currently waiting call will unblock and return H_ERR_MQCNCL. Any dequeue_message call invoked in the future will return H_ERR_MQCNCL immediately regardless of the queue status.

This command is typically needed for cleanup when any threads using the message queue and possibly blocked in dequeue_message must be released, because it is not allowed to destroy the queue (clear_message_queue) while any operators might still use it. It might be necessary to wait for the completion of those threads after invoking the *'abort_dequeuing'* command to be sure any operators using the queue are completed.

The message data currently stored in the message queue is not affected by the command. The effect of canceling any further *'abort_dequeuing'* calls can be reset again using the *'restore_queue'* command.

The parameter value must be integer *1*.

'restore_queue': Clears the effect of a previous *'abort_dequeuing'* command. This means the next call to dequeue_message will again attempt to deliver message data.

The parameter value must be integer *1*.

'flush_queue': Flushes the queue. All currently enqueued messages are removed from the queue and all associated resources, in particular the tuple or object data stored in the messages, are released.

The parameter value must be integer *1*.

'max_message_num': Sets the maximum number of messages that can be stored in the queue. The enqueue_message operator call would fail (H_ERR_MQOVL) if the queue would contain more messages after the operation than the defined maximum.

Setting the *'max_message_num'* parameter has no effect on the messages already present in the queue. No messages that are already in the queue can be removed by this operation.

The parameter value can be a positive integer or *-1* if the queue size should be unlimited. The string value *'infinite'* is accepted as well. The queue size is unlimited by default.

Parameters

▷ **QueueHandle** (input_control) . . . . . message_queue ⇝ *handle*
Message queue handle.
Number of elements: QueueHandle == 1
Restriction: QueueHandle != 0

▷ **GenParamName** (input_control) ..... string(-array) ⇝ *string*
Names of the queue parameters or action commands.
Number of elements: GenParamName == GenParamValue
Default: 'max_message_num'
List of values: GenParamName ∈ {'abort_dequeuing', 'restore_queue', 'flush_queue', 'max_message_num'}

▷ **GenParamValue** (input_control) ..... tuple(-array) ⇝ *string* / integer / real
Values of the queue parameters or action commands.
Number of elements: GenParamName == GenParamValue
Default: 1
Suggested values: GenParamValue ∈ {1, -1}

Example

```
* abort waiting dequeue_message operators
set_message_queue_param (Queue, 'abort_dequeuing', 1)
```

Result

If all the operator parameters and their values are valid, set_message_queue_param returns 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_message_queue

Possible Successors

clear_message_queue

See also

create_message_queue, clear_message_queue, enqueue_message, dequeue_message, get_message_queue_param, create_message, clear_message, set_message_tuple, get_message_tuple, set_message_obj, get_message_obj

Module

Foundation

set_message_tuple (: : MessageHandle, Key, TupleData :)

Add a key/tuple pair to the message.

set_message_tuple stores a tuple associated with a key in the message, which behaves as a dictionary-like data container. The message is denoted by the MessageHandle parameter.

TupleData including strings is copied by the operation, and can thus be immediately reused. An empty tuple is considered as a valid value that can be associated with the key. If any data (tuple or object) was already associated with given key (Key), the old data is destroyed by set_message_tuple and replaced by TupleData.

The Key has to be a string or an integer. String keys are treated case sensitive.

The tuple data for the given key can be retrieved again from the message using get_message_tuple.

Attention

Note that if the tuple contains any handles (which are treated as simple integers) only the handle values are copied by the operation, not the resources behind those handles.

Parameters

▷ **MessageHandle** (input_control) . . . . . message ⇝ *handle*
Message handle.
Number of elements: MessageHandle == 1
Restriction: MessageHandle != 0

▷ **Key** (input_control) . . . . . string ⇝ *string* / integer
Key string or integer.
Number of elements: Key == 1

▷ **TupleData** (input_control) . . . . . tuple-array ⇝ *string* / integer / real / handle
Tuple value to be associated with the key.

Example

```
create_message (Message)
set_message_tuple (Message, 'simple_integer', 27)
set_message_tuple (Message, 'simple_string', 'Hello world')
set_message_tuple (Message, 'mixed_tuple', ['The answer', 42])
```

Result

If the operation succeeds, `set_message_tuple` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised. Possible error conditions include invalid parameters (handle or key) or resource allocation error.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- MessageHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_message`

Possible Successors

`enqueue_message`, `set_message_obj`

Alternatives

`set_message_obj`

See also

`create_message`, `clear_message`, `get_message_tuple`, `set_message_obj`, `get_message_obj`, `set_message_param`, `get_message_param`, `enqueue_message`, `dequeue_message`

Module

Foundation

signal_condition (: : ConditionHandle :)

Signal a condition synchronization object.

`signal_condition` restarts one of the threads that are waiting on the condition variable `ConditionHandle`. If no threads are waiting on `ConditionHandle`, nothing happens. If several threads are waiting on `ConditionHandle`, exactly one is restarted, but it is not specified which one.

Parameters

▷ **ConditionHandle** (input_control) . . . . . condition ⇝ *handle*
Condition synchronization object.

Result

If the condition handle is valid, the operator `signal_condition` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_condition`, `wait_condition`

See also

`wait_condition`, `timed_wait_condition`

Module

Foundation

signal_event (: : EventHandle :)

Unlock an event synchronization object.

`signal_event` releases the event given in `EventHandle`. Signaling an event that is already signaled has no effect.

Parameters

▷ **EventHandle** (input_control) . . . . . event ⇝ *handle*
Event synchronization object.

Result

If the event handle is valid, the operator `signal_event` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`wait_event`

Possible Successors

`wait_event`, `clear_event`

Module

Foundation

timed_wait_condition (: : ConditionHandle, MutexHandle, Timeout :)

Bounded wait on the signal of a condition synchronization object.

`timed_wait_condition` atomically unlocks `MutexHandle` and waits on `ConditionHandle`, as `wait_condition` does, but it also bounds the duration of the wait. If `ConditionHandle` has not been

signaled within the amount of time specified by Timeout in micro seconds, the mutex MutexHandle is re-acquired and timed_wait_condition returns the error H_ERR_TIMEOUT.

Using negative values for Timeout means an infinite waiting time.

Parameters

▷ **ConditionHandle** (input_control) ..... condition ⇝ *handle*
Condition synchronization object.

▷ **MutexHandle** (input_control) ..... mutex ⇝ *handle*
Mutex synchronization object.

▷ **Timeout** (input_control) ..... number ⇝ *integer*
Timeout in micro seconds.

Result

timed_wait_condition returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised. If a timeout occurs, the error H_ERR_TIMEOUT is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

signal_condition, clear_condition

Module

Foundation

try_lock_mutex (: : MutexHandle : Busy)

Lock a mutex synchronization object.

try_lock_mutex behaves identically to lock_mutex, except that it does not block the calling thread if the mutex is already locked by another thread but returns immediately. The state of the mutex before trying to lock it is returned in Busy. *1* indicates that the mutex was already locked before calling try_lock_mutex, *0* that the mutex was unlocked (signaled).

Parameters

▷ **MutexHandle** (input_control) ..... mutex ⇝ *handle*
Mutex synchronization object.

▷ **Busy** (output_control) ..... number ⇝ *integer*
Mutex already locked?

Result

If the mutex handle is valid, the operator try_lock_mutex returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_mutex

Possible Successors

unlock_mutex

See also

lock_mutex

Module

Foundation

try_wait_event (: : EventHandle : Busy)

Lock an event synchronization object only if it is unlocked.

try_wait_event behaves identically to wait_event, except that it does not block the calling thread when waiting until the object is signaled but returns immediately. The state of the event before entering try_wait_event is returned in Busy. *1* indicates that the event was non-signaled before calling try_wait_event, *0* that the event was signaled.

Parameters

▷ **EventHandle** (input_control) ..... event ⇝ *handle*
Event synchronization object.

▷ **Busy** (output_control) ..... number ⇝ *integer*
Object already locked?

Result

If the event handle is valid, the operator try_wait_event returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_event

Possible Successors

signal_event

See also

wait_event

Module

Foundation

unlock_mutex (: : MutexHandle :)

Unlock a mutex synchronization object.

unlock_mutex unlocks the mutex object given by MutexHandle. The mutex must have been locked by the calling thread. Attempting to unlock a mutex that is not locked by the calling thread is undefined behavior.

Parameters

▷ **MutexHandle** (input_control) ..... mutex ⇝ *handle*
Mutex synchronization object.

Result

If the mutex handle is valid, the operator unlock_mutex returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

lock_mutex, try_lock_mutex

Possible Successors

lock_mutex, try_lock_mutex, clear_mutex

Module

Foundation

wait_barrier (: : BarrierHandle :)

Wait on the release of a barrier synchronization object.

wait_barrier blocks the thread at a barrier object given by BarrierHandle until a specified number of threads have called the same barrier object. This number is specified by parameter TeamSize during the creation of the barrier object by create_barrier.

Parameters

▷ **BarrierHandle** (input_control) ..... barrier ⇝ *handle*
Barrier synchronization object.

Result

If the barrier handle is valid, the operator wait_barrier returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

clear_barrier

Module

Foundation

wait_condition (: : ConditionHandle, MutexHandle :)

wait on the signal of a condition synchronization object.

wait_condition atomically unlocks the MutexHandle (as per unlock_mutex) and waits for the condition variable ConditionHandle to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The mutex must be locked by the calling thread on entrance to wait_condition. Before returning to the calling thread, wait_condition re-acquires MutexHandle (as per lock_mutex).

Unlocking the mutex and suspending on the condition variable is done atomically. Thus, if all threads always acquire the mutex before signaling the condition, this guarantees that the condition cannot be signaled (and thus ignored) between the time a thread locks the mutex and the time it waits on the condition variable.

Parameters

▷ **ConditionHandle** (input_control) ..... condition ⇝ *handle*
Condition synchronization object.

▷ **MutexHandle** (input_control) ..... mutex ⇝ *handle*
Mutex synchronization object.

Result

`wait_condition` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`signal_condition`, `clear_condition`

Module

Foundation

wait_event (: : EventHandle :)

Lock an event synchronization object.

`wait_event` waits on the event object passed in `EventHandle` until it is signaled. When the event is signaled, `wait_event` succeeds and sets the state automatically to non-signaled again. The kind of wait depends on the event's attributes set during the creation in `create_event`.

Parameters

▷ **EventHandle** (input_control) ..... event ⇝ *handle*
Event synchronization object.

Result

If the event handle is valid, the operator `wait_event` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_event`

Possible Successors

`signal_event`

Module

Foundation

write_message (: : MessageHandle, FileName, GenParamName, GenParamValue :)

Write a message to a file.

`write_message` writes the message denoted by the `MessageHandle` to the file given by `FileName`. The default HALCON file extension for a message is 'hmsg'.

If `MessageHandle` contains a handle that can not be serialized or that has been freed already, an exception is raised per default. This behavior is controlled by `GenParamName` *'raise_error_if_content_not_serializable'*, and the corresponding `GenParamValue` can take the following values:

'true': The default: Errors are raised and the writing process aborted.

'low_level': Only low level errors are raised. Instead of the handle concerned an empty handle is written in `FileName` and the writing process will be continued. The behavior regarding HALCON low level errors is determined by *'do_low_error'* in `set_system`.

'false': The errors are suppressed. Instead of the handle concerned an empty handle is written in `FileName` and the writing process will be continued.

Parameters

▷ **`MessageHandle`** (input_control) . . . . . message ⇝ *handle*
Message handle.
Number of elements: MessageHandle == 1
Restriction: MessageHandle != 0

▷ **`FileName`** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: `.hmsg`

▷ **`GenParamName`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: `GenParamName` ∈ {'raise_error_if_content_not_serializable'}

▷ **`GenParamValue`** (input_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
Value of the generic parameter.
Default: []
Suggested values: `GenParamValue` ∈ {'true', 'false', 'low_level'}

Result

If the parameters are valid, the operator `write_message` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_message`, `deserialize_handle`, `create_message`

Possible Successors

`read_message`

See also

`read_message`, `serialize_handle`, `deserialize_handle`

Module

Foundation

25.9 Operating System

`count_seconds` (: : : Seconds)

Passed Time.

The operator `count_seconds` helps to measure time. Each operator call returns a time value. The difference of the values of two successive calls provides the time interval in seconds. The mode of measuring time can be set with `set_system('clock_mode',...)`.

Attention

The time measurement is not exact and depends on the load of the computer.

Parameters

▷ **Seconds** (output_control) ..... real ⇝ *real*
Process time since the program start.

Example

```
count_seconds(Start)
* program segment to be measured
count_seconds(End)
Seconds := End - Start
```

Result

The operator count_seconds always returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

set_system

Module

Foundation

get_system_time (: : : MSecond, Second, Minute, Hour, Day, YDay, Month, Year)

Read out the system time.

The operator get_system_time reads out the current system time. The system time is read according to the local time zone or as UTC, see *'system_time_base'* in get_system.

Parameters

▷ **MSecond** (output_control) ..... integer ⇝ *integer*
Milliseconds (0..999).

▷ **Second** (output_control) ..... integer ⇝ *integer*
Seconds (0..59).

▷ **Minute** (output_control) ..... integer ⇝ *integer*
Minutes (0..59).

▷ **Hour** (output_control) ..... integer ⇝ *integer*
Hours (0..23).

▷ **Day** (output_control) ..... integer ⇝ *integer*
Day of the month (1..31).

▷ **YDay** (output_control) ..... integer ⇝ *integer*
Day of the year (1..366).

▷ **Month** (output_control) ..... integer ⇝ *integer*
Month (1..12).

▷ **Year** (output_control) ..... integer ⇝ *integer*
Year (xxxx).

Result

get_system_time always returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

get_image_time

Module

Foundation

system_call (: : Command :)

Execute a system command.

The operator system_call executes the system command given in Command.

On Windows operating systems, the system command can be prefixed by *'start /b'* to avoid waiting for the started application. On Unix-like operating systems, an interactive shell ('csh -i') will be started, if Command contains an empty string.

Parameters

▷ **Command** (input_control) . . . . . string ⇝ *string*
Command to be called by the system.
Default: 'ls'

Result

On Windows operating systems,the operator system_call will always return the value 2 (H_MSG_TRUE) unless there was a problem creating a new process. On other operating systems, system_call will always return the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

count_seconds

See also

wait_seconds, count_seconds

Module

Foundation

wait_seconds (: : Seconds :)

Delaying the execution of the program.

The operator wait_seconds delays the execution by the number of seconds indicated in Seconds.

wait_seconds uses the *'performance_counter'* for the measurement of time intervals. The *'performance_counter'* is explained in the description of the parameter *'clock_mode'* of the operator set_system.

Parameters

▷ **Seconds** (input_control) . . . . . real ⇝ *real*
Number of seconds by which the execution of the program will be delayed.
Default: 10
Restriction: Seconds >= 0

Result

The operator wait_seconds always returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

system_call

See also

system_call, count_seconds

Module

Foundation

25.10 Parallelization

get_aop_info (: : OperatorName, IndexName, IndexValue, InfoName : InfoValue)

Return AOP information for operators.

HALCON supports a mechanism to optimize the behavior of automatic operator parallelization (AOP) for a given hardware of a machine. The information for this optimization can be determined by the operator optimize_aop and can also be partly modified by the operator set_aop_info.

To address specific AOP information, the operator's optimization data is indexed within a 3-ary hierarchy that can be obtained by query_aop_info. The three hierarchy stages are indexed by the operator name, passed in OperatorName, the iconic type, and a parameter string value denoting a special mode or method supported by the indexed operator. The latter two indices are passed by IndexName and IndexValue. IndexName holds the dimension identifier whereas IndexValue holds the index value as returned in query_aop_info.

get_aop_info returns the specific information parts of an operator's AOP knowledge in InfoValue if exactly one index per stage was set. The scope of information is specified by InfoName and supports following values:

'max_threads' returns the maximum allowed thread number. If no optimization data is stored, InfoValue contains *-1*. In case the specified operator does not support automatic parallelization, *1* is returned.

'split_level' returns all allowed data split levels of the automatic parallelization for this operator, the specified iconic type and parameter value. Possible levels are *'split_tuple'*, *'split_channel'*, *'split_domain'*, and *'split_partial'*. If no split level is supported or every level was switched off for the indexed operator an empty string *''* is returned. Use get_operator_info to query all split levels generally supported by an operator.

'model' returns the stored model type. Possible values are *'threshold'*, *'linear'*, and *'mlp'* (see also the description of operator optimize_aop). If no optimization data is stored, an empty string is returned.

Parameters

▷ **OperatorName** (input_control) ..... string ⇝ *string* / integer
Operator to get information for
Suggested values: OperatorName ∈ {'mean_image', 'opening_circle', 'find_shape_model'}

▷ **IndexName** (input_control) ..... string-array ⇝ *string*
Further index stages
Default: ['iconic_type','parameter:0']

▷ **IndexValue** (input_control) ..... string-array ⇝ *string*
Further index values
Number of elements: IndexName == IndexValue
Default: ['byte','']

▷ **InfoName** (input_control) ..... string ⇝ *string*
Scope of information
Default: 'max_threads'
Suggested values: InfoName ∈ {'max_threads', 'split_level', 'model'}

▷ **InfoValue** (output_control) ..... string(-array) ⇝ *string*
Value of information

Result

`get_aop_info` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

Module

Foundation

optimize_aop (: : OperatorName, IconicType, FileName, GenParamName, GenParamValue :)

Check hardware regarding its potential for automatic operator parallelization.

HALCON supports automatic operator parallelization (AOP) for various operators. `optimize_aop` is necessary for an efficient automatic parallelization, what HALCON uses to speed up the execution of operators. As the parallelization of operators is done automatically, there is no need for the user to explicitly prepare or change programs for their parallelization. Thus, all HALCON-based programs can be used without adaptations on multiprocessor hardware and users benefit from the potential of parallel hardware. By default, HALCON uses the maximum available number of threads for AOP, up to the number of processors. But, depending on the data size and the parameter set passed to an operator, parallelizing on the maximum thread number might be too excessive and inefficient. `optimize_aop` optimizes the AOP in terms of thread number and checks a given hardware with respect to a parallel processing of HALCON operators. In doing so, it examines every operator, which can be possibly be sped up by an automatic parallelization on tuple, channel, or domain level (the partial level is not considered). Each examined operator is executed several times - both sequentially and parallel - with a changing set of input parameter values/images. The latter helps to evaluate dependencies between an operator's input parameter characteristics (e.g., the size of an input image) and the efficiency of its parallel processing. This may take up to a couple of hours depending on the settings of the operator's parameters. It is essential for a correct optimization not to run any other computation-intensive application simultaneously on the machine, as this would strongly influence the time measurements of the hardware check and thus would lead to wrong results.

Overall, `optimize_aop` performs several test loops and collects a lot of hardware-specific information which enables HALCON to optimize the automatic parallelization for a given hardware. The hardware information can be stored in a binary file given by `FileName` so that it can be used again in future HALCON sessions or even transferred to other machines, identical in hardware. When passing an empty string `''` as file name, `optimize_aop` stores the optimization data to the specific HALCON system file '.aop_info' in the HALCON installation directory (in Linux, see environment variable $HALCONROOT) or within the common application data folder (Windows). This specific file is automatically read when initializing HALCON during the first operator call. Note that the user must have appropriate privileges for read and write access. `optimize_aop` will check the file access before starting the AOP optimization and return an appropriate error when failing. The written file can be read by using the operator `read_aop_knowledge` again.

Thus, it is sufficient, to start `optimize_aop` once on each multiprocessor machine that is used for parallel processing. Of course, it should be started again, if the hardware of the machine changes, for example, by installing a new CPU, new memory, or if the operating system of the machine changes. It is necessary to start `optimize_aop` once for each new processing environment as the time response of operators may differ. A new processing environment is given, if different operating systems, such as Windows and Linux, or different HALCON architectures, different HALCON variants are used, i.e., HALCON versus HALCON XL, or when updating to a new HALCON version or revision. Together with the machine's host name, these dependencies form the knowledge attributes and are stored to the file together with the AOP optimization data. The attributes identify the machine-specific AOP knowledge set and enable the storage of different knowledge sets to the same file.

`optimize_aop` offers a set of parameters to restrict the extensiveness and specify the behavior of the optimization process. A subset of HALCON operators can be passed to `OperatorName`. This will restrict the optimization process to the operators passed by their name. When passing an empty string `''`, all operators will be tested. A subset of iconic types that should be checked can be set by parameter `IconicType`. Again, passing an empty

string '' will test all supported iconic types. Further settings to modify the optimization process can be parameterized by a pair of values passed to GenParamName and GenParamValue. Every entry in GenParamName must have a corresponding entry in GenParamValue, meaning the tuples passed to the parameters must have the same length. GenParamName supports the values in following list, specifying for each possible value for GenParamName all possible applicable values for GenParamValue:

'none' does nothing specific but ignores the parameter GenParamValue.

'system_mode', *'file_mode'* set the way how the information of the system or file, respectively, gets updated.

- *'truncate'* for GenParamValue deletes all the existing information before the new knowledge loaded from file is added.
- *'renew'* overwrites existing knowledge and adds new one (default).
- *'append'* keeps all existing operator information and just adds the knowledge not already contained.
- *'nil'* performs the optimization process but refuses any update of the system or file, respectively.

'parameters' also tests for appropriate operator parameters if the corresponding value of GenParamValue is set to *'true'*. These operator parameters are supposed to show significant influence on the operator's processing time like string parameters picking up a certain operator mode or method or, e.g., parameters setting a filter size. *'false'* dismisses the parameter check in favor of a faster knowledge identification (default).

'model' sets the underlying simulation of the behavior of parallelizing operators on the current hardware. Different models can be selected by GenParamValue differing in hardware adaptability and computation time:

- *'threshold'* determines if it is profitable to run on the maximum thread number or not. This is default on dual processor systems.
- *'linear'* specifies a linear scale model to determine the best thread number for a given data size and parameter set. This is default on multiprocessor systems.
- *'mlp'* is the most complex but also most adaptable model. Note that the optimization process can take a couple of hours depending on the used hardware topology.

'timeout' sets a maximum execution time for a tested operator. If the execution of an operator exceeds the timeout, the test on this operator is aborted. No information about this operator will be stored in this case. The timeout value is set by GenParamValue in seconds. Specifying *'infinite'* prevents any abortion of an operator's optimization process (default).

'split_level' restricts the optimization process on a certain parallelization method. The corresponding value of GenParamValue can contain one of the following values, therefore:

- *'split_domain'* performs the check on all image processing operators supporting data parallelization on domain level.
- *'split_channel'* performs the check on all image processing operators supporting data parallelization on channel level.
- *'split_tuple'* performs the check on all operators supporting data parallelization on tuple level.

Attention

During its test loops optimize_aop has to start every examined operator several times. Thus, the processing time of optimize_aop can be rather long depending on the operator's parameter settings. It is essential for a correct optimization not run any other computation-intensive application simultaneously on the machine, as this would strongly influence the time measurements of the hardware check and thus would lead to wrong results. Note that according to the file location optimize_aop must be called by users with the appropriate privileges for storing the parallelization information. See the operator's description above for more details about these subjects.

Parameters

▷ **OperatorName** (input_control) ..... string(-array) ⇝ *string*
Operators to check
Default: ''

▷ **IconicType** (input_control) ..... string(-array) ⇝ *string*
Iconic object types to check
Default: ''
Suggested values: IconicType ∈ {'', 'byte', 'int1', 'int2', 'uint2', 'int4', 'int8', 'direction', 'cyclic', 'vector_field', 'complex', 'region', 'xld', 'xld_cont', 'xld_poly'}

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string* / integer
Knowledge file name
Default: ”

▷ **GenParamName** (input_control) . . . . . string-array ⇝ *string*
Parameter name
Default: ’none’
Suggested values: GenParamName ∈ {’parameters’, ’model’, ’timeout’, ’file_mode’, ’system_mode’, ’split_level’}

▷ **GenParamValue** (input_control) . . . . . string-array ⇝ *string* / real / integer
Parameter value
Number of elements: GenParamName == GenParamValue
Default: ’none’
Suggested values: GenParamValue ∈ {’true’, ’renew’, ’truncate’, ’threshold’, ’linear’, ’mlp’, -1.0}

Result

optimize_aop returns 2 (H_MSG_TRUE) if all parameters are correct and the file could be read. If necessary, an exception is raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

query_aop_info (: : OperatorName, IndexName, IndexValue : Name, Value)

Query indexing structure of AOP information for operators.

HALCON supports a mechanism to adapt the behavior of automatic parallelization of image processing operators (AOP) to the machine’s hardware capability. The information for this adaption can be obtained by the operator optimize_aop and can be partly read and modified by the operators get_aop_info and set_aop_info, respectively. To address specific AOP information the operator’s optimization data is indexed within a 3-ary hierarchy that can be obtained by query_aop_info. The three hierarchy stages are indexed by the operator name, the iconic type, and a parameter string value denoting a special mode or method supported by the operator.

If an empty string ” is passed to OperatorName, the parameters IndexName and IndexValue are ignored and all operator names in Value are returned, for which optimization data is stored. Name contains the index identifier *’operator’* as many times as operator names are returned. Specifying a single operator name in OperatorName and passing an empty string ” to the parameters IndexName and IndexValue queries all iconic types the operator holds optimization data for. Value contains the type strings, Name the index identifier *’iconic_type’*, respectively. Finally, when specifying a certain operator in OperatorName and a specific iconic type by passing the dimension identifier *’iconic_type’* to IndexName and the specific iconic type to IndexValue, Name contains the parameter index identifier Value holds the parameter string values supported by the operator. The format of the parameter identifier string is compound containing the string *parameter*, followed by a colon separator *:* and a digit, denoting the parameter index (starting with 1). If the operator does not support any mode parameter or the operator’s parameter values were not varied during optimization process of optimize_aop the string *’parameter:0’* is returned for Name and Value contains an empty string. In case no AOP information is stored for the specified index, Name returns an empty string.

Parameters

▷ **OperatorName** (input_control) . . . . . string ⇝ *string* / integer
Operator to get information for
Default: ”

▷ **IndexName** (input_control) ..... string(-array) ⇝ *string* / integer
Further specific index
Default: ''
Suggested values: IndexName ∈ {'', 'iconic_type'}

▷ **IndexValue** (input_control) ..... string(-array) ⇝ *string* / integer
Further specific address
Number of elements: IndexName == IndexValue
Default: ''
Suggested values: IndexValue ∈ {'', 'byte', 'int1', 'int2', 'uint2', 'int4', 'int8', 'direction', 'cyclic', 'vector_field', 'complex', 'region', 'xld', 'xld_cont', 'xld_poly'}

▷ **Name** (output_control) ..... string-array ⇝ *string*
Name of next index stage

▷ **Value** (output_control) ..... string-array ⇝ *string*
Values of next index stage

Result

query_aop_info returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

Module

Foundation

read_aop_knowledge (: : FileName, GenParamName, GenParamValue : Attributes, OperatorNames)

Load knowledge about hardware dependent behavior of automatic operator parallelization.

read_aop_knowledge loads the automatic operator parallelization (AOP) knowledge of HALCON operators from file. The knowledge is used to better utilize multiprocessor hardware in order to speed up the processing of operators. To optimize the automatic operator parallelization, HALCON needs some knowledge about the speed of operators on the used hardware. This hardware-specific knowledge can be obtained and stored by calling the operator optimize_aop and write_aop_knowledge, respectively.

With read_aop_knowledge, it is possible to load this knowledge explicitly from a binary file. In doing so, FileName denotes the name of this file (incl. path and file extension). If an empty string '' is passed to FileName, HALCON reads the knowledge from the specific file .aop_info in the HALCON installation directory (Linux) or within the common application data folder (Windows). This file is read by HALCON automatically during the initialization phase but can easily be re-read by read_aop_knowledge. The read knowledge updates the system's parallelization information and can be controlled by pairs of additional parameter values passed to GenParamName and GenParamValue. Every entry in GenParamName must have one corresponding, specifying entry in GenParamValue, meaning that the tuples passed to the parameters must have the same length. GenParamName supports the values in following list, describing the possible applicable values for GenParamValue:

'ignore_attrib' suppresses to check knowledge attributes. E.g., while reading the file read_aop_knowledge checks whether its content was written for the currently used computer and whether the contained parallelization information regards the currently used HALCON version, revision, and architecture. The attributes to suppress are specified by passing any of the following values to GenParamValue:

'host' the host name of the current machine

'cpu_info' the machine topology information

'variant' the HALCON variant

'version' the HALCON version

'revision' the HALCON revision

'architecture' the HALCON architecture

Multiple suppressions of attribute checks are possible.

'mode' sets the way how the system's information gets updated.

'truncate' for GenParamValue deletes all the existing information before the new knowledge loaded from file is added.

'renew' overwrites existing knowledge and adds new one (default).

'append' keeps all existing operator information and just adds the knowledge not already contained.

'nil' refuses any updates but returns the knowledge attributes in Attributes and all the operators contained in file in OperatorNames.

'operator' denotes that the corresponding index value of GenParamValue contains an operator name. Multiple definitions of operators are possible. By default, information of any contained operator is loaded.

'iconic_type' focuses on AOP information for a specific iconic type, specified by following values of GenParamValue: *'byte'*,*'uint2'*,*'real'*,*'int1'*, *'int2'*,*'int4'*,*'int8'*,*'direction'*, *'vector_field'*,*'cyclic'*,*'vector_field'*, *'complex'*,*'region'*,*'xld'*. Multiple definitions of iconic types are possible.

The names of updated operators are returned in OperatorNames, the attributes of the read knowledge in Attributes. The latter tuple codes the following meaning by index beginning with index 0 in corresponding order: host name, HALCON variant, HALCON version, HALCON revision, HALCON architecture.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
Name of knowledge file
Default: ''

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Parameter name
Default: 'none'
Suggested values: GenParamName ∈ {'none', 'ignore_attrib', 'operator', 'iconic_type', 'mode'}

▷ **GenParamValue** (input_control) . . . . . string(-array) ⇝ *string* / integer / real
Parameter value
Number of elements: GenParamName == GenParamValue
Default: 'none'
Suggested values: GenParamValue ∈ {'none', 'host_name', 'variant', 'architecture', 'version', 'revision', 'byte', 'int1', 'int2', 'uint2', 'int4', 'int8', 'direction', 'cyclic', 'vector_field', 'complex', 'region', 'xld', 'xld_cont', 'xld_poly', 'nil', 'truncate', 'replace', 'renew', 'append'}

▷ **Attributes** (output_control) . . . . . string-array ⇝ *string*
Knowledge attributes

▷ **OperatorNames** (output_control) . . . . . string(-array) ⇝ *string*
Updated Operators

Result

read_aop_knowledge returns 2 (H_MSG_TRUE) if all parameters are correct and the file could be read. If necessary, an exception is raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

set_aop_info (: : OperatorName, IndexName, IndexValue, InfoName, InfoValue :)

Set AOP information for operators.

HALCON supports a mechanism to adapt the behavior of automatic parallelization of image processing operators (AOP) to the machine's hardware capability. The information for this adaption can be obtained by the operator optimize_aop and can also be partly read by the operator set_aop_info.

To address specific AOP information the operator's optimization data is indexed within a 3-ary hierarchy that can be obtained by query_aop_info. The three hierarchy stages are indexed by the operator name, passed in OperatorName, the iconic type, and a parameter string value denoting a special mode or method supported by the indexed operator. The latter two indices are passed by IndexName and IndexValue. IndexName holds the dimension identifier whereas IndexValue addresses the index as returned in query_aop_info. It is also possible to address multiple indices on the same stage to set a specific information value to multiple knowledge sets. Omitting an index name or passing an empty string as index value addresses this stage in all. E.g., passing '' to OperatorName, the tuple *['iconic_type','iconic_type']* to IndexName, and the tuple *['byte','uint2']* to IndexValue addresses all operators processing byte or uint2 images.

set_aop_info sets the specific information parts of an operator's AOP knowledge passed in InfoValue. The scope of information is specified by InfoName and supports the following:

'max_threads' sets the maximum allowed thread number this operator is allowed to run with the specified iconic type and parameter value.

'split_level' sets the allowed data split levels of the automatic parallelization for this operator, the specified iconic type and parameter value. Possible levels are *'split_tuple'*, *'split_channel'*, *'split_domain'*, and *'split_partial'*.

Attention

set_aop_info modifies the AOP system knowledge. These modifications will also be stored by write_aop_knowledge. It is advisable to have a backup knowledge file before modifying the system knowledge.

Parameters

▷ **OperatorName** (input_control) ..... string(-array) ⇝ *string* / integer
Operator to set information to
Default: ''

▷ **IndexName** (input_control) ..... string(-array) ⇝ *string* / integer
Further specific index
Default: ''
Suggested values: IndexName ∈ {'iconic_type', ['iconic_type','parameter:0']}

▷ **IndexValue** (input_control) ..... string(-array) ⇝ *string* / integer
Further specific address
Number of elements: IndexName == IndexValue
Default: ''
Suggested values: IndexValue ∈ {'byte', ['uint2','']}

▷ **InfoName** (input_control) ..... string ⇝ *string*
Scope of information
Default: 'max_threads'
Suggested values: InfoName ∈ {'max_threads', 'model'}

▷ **InfoValue** (input_control) ..... integer(-array) ⇝ *integer*
AOP information value

Result

set_aop_info returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

Module

Foundation

write_aop_knowledge (: : FileName, GenParamName, GenParamValue :)

Write knowledge about hardware dependent behavior of automatic operator parallelization to file.

write_aop_knowledge stores the automatic parallelization knowledge of HALCON operators to file. The knowledge hold by the HALCON system is used to better utilize multiprocessor hardware in order to speed up the processing of operators. To optimize the automatic parallelization of operators, HALCON needs some specific knowledge about the speed of operators on the used hardware. This hardware-specific knowledge can be obtained by calling the operator optimize_aop or read from file by using read_aop_knowledge, respectively.

With write_aop_knowledge it is possible to store this knowledge explicitly in a binary file. At this, FileName denotes the name of this file (incl. path and file extension). If an empty string '' is passed to FileName, HALCON writes the knowledge to the specific file .aop_info in the HALCON installation directory (Linux) or within the common application data folder (Windows). This file is read by HALCON during the initialization phase automatically. write_aop_knowledge adds attributes to the knowledge like the host name and information about HALCON architecture, variant, version, and revision. This enables HALCON to write aop knowledge sets of different machines or HALCON systems into the same file. Therefore, write_aop_knowledge can control the updating of the files contents by a pair of values passed to GenParamName and GenParamValue. Every entry in GenParamName must have one corresponding, specifying entry in GenParamValue, meaning that the tuples passed to the parameters must have the same length. GenParamName supports the values in following list, describing the possible applicable values for GenParamValue:

'mode' sets the way how existing knowledge with same attributes would be updated in the file.

- *'truncate'* for GenParamValue deletes the existing knowledge with matching attributes before the new information is added to file.
- *'renew'* overwrites existing knowledge and adds new one (default).
- *'append'* keeps all existing operator information and just adds the knowledge not already contained.
- *'nil'* however refuses any update of the file but checks the user privileges, i.e., if read write access is available.

'operator' denotes that the corresponding index value of GenParamValue contains an operator name. It delimits the storage on a dedicated operator set. Multiple definitions of operators are possible. By default, information of any operator possessing AOP optimization data is written.

'iconic_type' focuses on AOP information for a specific iconic type, specified by following values of GenParamValue: *'byte'*, *'uint2'*, *'real'*, *'int1'*, *'int2'*, *'int4'*, *'int8'*, *'direction'*, *'vector_field'*, *'cyclic'*, *'complex'*, *'region'*, *'xld'*. Multiple definitions of iconic types are possible.

Parameters

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of knowledge file
Default: ''

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Parameter name
Default: 'none'
Suggested values: GenParamName ∈ {'mode', 'operator', 'iconic_type'}

▷ **GenParamValue** (input_control) . . . . . string(-array) ⇝ *string* / integer / real
Parameter value
Number of elements: GenParamName == GenParamValue
Default: 'none'
Suggested values: GenParamValue ∈ {'truncate', 'renew', 'append'}

Result

write_aop_knowledge returns 2 (H_MSG_TRUE) if all parameters are correct and the file could be read. If necessary, an exception is raised.

Execution Information

- Multithreading type: exclusive (runs in parallel only with independent operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Module

Foundation

25.11 Parameters

get_system (: : Query : Information)

Get current value of HALCON system parameters.

The operator get_system returns information concerning the currently activated HALCON system parameters.

Readable and Writable Parameters

get_system returns information about all parameters that are set with set_system. For more information on read- and writable parameters, please refer to the reference documentation of set_system.

Read-Only Parameters

The following system parameters can be queried:

- **Versions:**

  'halcon_xl': The currently used variant of HALCON: HALCON XL, which allows images larger than 32768 × 32768 (*'true'*) or HALCON, which restricts the maximal image size to 32768 × 32768 (*'false'*).

  'version': HALCON version number.

  'last_update': Creation date of the HALCON library.

  'revision': Revision number of the HALCON library.

  'file_version': File version number of the HALCON library. The number is made up of the HALCON version number, the revision number, and the build number.

- **Upper Limits:**

  'max_inp_obj_par': Maximum number of input parameters.

  'max_outp_obj_par': Maximum number of output parameters.

  'max_inp_ctrl_par': Maximum number of input control parameters.

  'max_outp_ctrl_par': Maximum number of output control parameters.

  'max_window': Maximum number of windows.

- **Graphic:**

  'window_name': Returns the value that may have been set with set_window_attr ('window_title',...). If nothing has been set yet, the string *'default'* is returned.

- **Parallelization:**

  'processor_num': Returns the number of processors that HALCON has found on the hardware it is running on.

  'tsp_used_thread_num': Returns the number of threads the last operator was using for automatic operator parallelization. Only operators that were run by the calling thread and support automatic parallelization are considered. Note that using this parameter in HDevelop works only in run mode, not in single step mode.

  'tsp_used_split_levels' Returns all split levels the last operator was using for automatic parallelization. Only operators that were run by the calling thread and support automatic parallelization are considered. If the last operator was not parallelized, the string *'false'* is returned. Note that using this parameter in HDevelop works only in run mode, not in single step mode.

- **Licensing:**

  'available_license_files' The list of license files HALCON can find.

  'current_license_info' A tuple containing information about the current license. If the license is invalid, the tuple contains a single element with the value *'invalid'*. Otherwise, the tuple contains the following elements in order:

  – The name of the license file used to obtain the current license.

  - The type of license, either *'runtime'*, *'development'*, or *'sasl'*.
  - The highest version of HALCON licensed by the current license.
  - The expiration date of the current license in ISO 8601 format, or *'permanent'* for licenses without an expiration date.
  - The list of HALCON modules licensed by the current license as a single string, separated by ' | '.
  - The number of concurrent licenses, or 0 for unlimited concurrent use.
  - The number of concurrent sessions, or 0 for unlimited concurrent sessions.
  - The host ID used to obtain the current license.
  - The flags of the current license.
  - The contents of the license info field.

  If a particular piece of information is not available, an empty string is returned instead.

'expiration_date' The date the HALCON license will expire, in ISO 8601 format, or *'permanent'* if the license does not expire.

'failed_license_rechecks' The number of failed license rechecks since process start.

'is_license_valid' This parameter triggers an immediate license check and returns *'true'* if a valid license could be found, or *'false'* otherwise.

'licensed_hostid' The host ID used to obtain the current license.

'licensed_modules' A list of all modules licensed by the currently used license.

'licensed_version' The highest HALCON version the current license is valid for.

'licensed_product_edition' The HALCON product edition. Will be one of *'Progress'*, *'Steady'*, *'Steady Deep Learning'*, or *'Student Edition'* depending on the license being used. If there is no valid license, an empty string is returned instead.

'supported_editions' Return an array containing the editions this HALCON binary supports. Possible values are *'steady'* and *'progress'*.

'unlicensed_operators' A list of operators that require a module that is not licensed and thus cannot be used.

- **Other:**

'available_parameters' The list of parameters `get_system` understands.

'hostids': The host IDs of the computer that can be used for licensing HALCON.

'num_proc': Total number of the available HALCON procedures (*'num_sys_proc'* + *'num_user_proc'*).

'num_sys_proc': Number of the system procedures (supported procedures).

'num_user_proc': Number of the user defined procedures (see also "`Extension Package Programmer's Manual`").

'byte_order': Byte order of the processor (*'msb_first'* or *'lsb_first'*).

'operating_system': Name of the operating system of the computer on which the HALCON process is being executed. Note that for all Windows versions, 'Windows NT' is returned. Use the parameter *'operating_system_version'* to query the precise version.

'operating_system_version': Version number of the operating system of the computer on which the HALCON process is being executed. To decode the Windows version numbers, please refer to the website information on operating systems and version numbers provided by Microsoft.

'hostname': Name of the computer on which the HALCON process is being executed. On Windows systems, if the computer is part of a cluster the name of the cluster is returned, not the name of the local computer. If the name cannot be determined, an empty string is returned.

'locale_raw': Locale setting that is used by the runtime environment for encoding system strings (according to the locale category LC_CTYPE). The result corresponds to the result of the system C function setlocale or the result of the shell command 'locale' called in a Linux terminal. In general, it contains data about the language, the country, and the code page or code set - under Windows, for instance, *'German_Germany.1252'* or under Linux *'en_US.utf8'*. Note, that on Linux the names are not clearly defined because they depend on the system and the current configuration. Thus, the code set can be set to 'utf8' or 'UTF-8', or may completely be missing. On systems that do not support locale an empty string is returned.

'locale_codeset': Code set that is used by the runtime environment for encoding system strings (according to the locale category LC_CTYPE). In contrast to *'locale_raw'*, this result contains only the code set or code page and no information about language and country. The code set is returned in a normalized way so that it can easier be used in tests. The result contains only lowercase letters and numbers, with

different results for Windows code pages and Linux code sets, as the code sets used are usually not identical. Under Windows typically the code page number prefixed by 'cp' is returned, e.g., *'cp1252'* (for code page 1252 which is commonly used in the western world and correlates widely with Latin-1) or *'cp932'* (which correlates with ShiftJIS, which is common in Japan). Common results under Linux are *'utf8'* (which is the default locale on most of the recent Linux distributions), *'iso885915'* (Latin-9), *'shiftjis'*, and *'ascii'* (referred to as ANSI_X3.4-1968, US-ASCII, POSIX, C).

'halcon_64': Flag, if the HALCON version is a 64 bit version (*'true'*) or not, i.e., if it is a 32 bit version (*'false'*).

'halcon_arch': Name of the HALCON architecture of the running HALCON process.

'library_fullname': Path to the currently loaded HALCON library including the file name.

'temp_mem': Amount of temporary memory used by the last operator in byte. The return value is only defined if `set_check('memory')` was called before.

'mmx_supported': Flag, if the processor supports MMX operations (*'true'*) or not (*'false'*).

'sse_supported': Flag, if the processor supports SSE operations (*'true'*) or not (*'false'*).

'sse2_supported': Flag, if the processor supports SSE2 operations (*'true'*) or not (*'false'*).

'sse3_supported': Flag, if the processor supports SSE3 operations (*'true'*) or not (*'false'*).

'ssse3_supported': Flag, if the processor supports SSSE3 operations (*'true'*) or not (*'false'*).

'sse41_supported': Flag, if the processor supports SSE41 operations (*'true'*) or not (*'false'*).

'sse42_supported': Flag, if the processor supports SSE42 operations (*'true'*) or not (*'false'*).

'avx_supported': Flag, if the processor supports AVX operations (*'true'*) or not (*'false'*).

'avx2_supported': Flag, if the processor supports AVX2 operations (*'true'*) or not (*'false'*).

'avx2_gather_recommended': Flag, if the usage of AVX2 gather instructions is recommended on the processor (*'true'*) or not (*'false'*).

'avx512f_supported': Flag, if the processor supports AVX-512F operations (*'true'*) or not (*'false'*).

'avx512dq_supported': Flag, if the processor supports AVX-512DQ operations (*'true'*) or not (*'false'*).

'avx512bw_supported': Flag, if the processor supports AVX-512BW operations (*'true'*) or not (*'false'*).

'avx512cd_supported': Flag, if the processor supports AVX-512CD operations (*'true'*) or not (*'false'*).

'avx512vl_supported': Flag, if the processor supports AVX-512VL operations (*'true'*) or not (*'false'*).

'avx512vbmi_supported': Flag, if the processor supports AVX-512VBMI operations (*'true'*) or not (*'false'*).

'avx512ifma_supported': Flag, if the processor supports AVX-512IFMA operations (*'true'*) or not (*'false'*).

'avx512vbmi2_supported': Flag, if the processor supports AVX-512VBMI2 operations (*'true'*) or not (*'false'*).

'avx512vpopcntdq_supported': Flag, if the processor supports AVX-512VPOPCNTDQ operations (*'true'*) or not (*'false'*).

'avx512bitalg_supported': Flag, if the processor supports AVX-512BITALG operations (*'true'*) or not (*'false'*).

'avx512vnni_supported': Flag, if the processor supports AVX-512VNNI operations (*'true'*) or not (*'false'*).

'neon_supported': Flag, if the processor supports NEON operations (*'true'*) or not (*'false'*).

'opengl_hidden_surface_removal_available': Flag, if the graphic card supports the acceleration of the hidden surface removal used in `create_shape_model_3d`, `find_shape_model_3d`, `project_shape_model_3d`, and `project_object_model_3d`. The minimum requirements are OpenGL 2.0 and the extensions GL_EXT_framebuffer_object and GL_ARB_texture_float. Please note that these features are not available via Windows Remote Desktop or X11 forwarding.

'alloctmp_max_used' The maximum amount of temporary memory required so far for a thread, in bytes. The first element of the tuple contains the value for the current thread, while any following elements contain the values of currently idle threads from the thread pool.
The maximum is reset when the temporary memory cache mode is set to *'idle'*.

'tsp_temporary_mem_cache_block_sizes' Returns a tuple containing the block sizes of all temporary memory blocks currently cached in the thread's temporary memory cache, in bytes.

'temporary_mem_reservoir_block_sizes' Returns a tuple containing the block sizes of all temporary memory blocks currently cached in the global temporary memory reservoir, in bytes.

'legacy_handle_mode', *'tsp_legacy_handle_mode'*: Returns a flag if the legacy handle mode is activated.

'memory_allocators_supported': Returns a tuple containing the list of available memory allocators that can be set using parameter *'memory_allocator'* of `set_system`.

- **CUDA support for deep learning:**

  'cuda_loaded': Returns *'true'* if the CUDA library could be loaded.

  'cuda_version': Returns the version of the CUDA library. If the CUDA library could not be loaded, *-1* is returned.

  'cuda_devices': Returns the names of available devices that are compatible with CUDA.

  'cudnn_loaded': Returns *'true'* if the cuDNN library could be loaded.

  'cudnn_version': Returns the version of the cuDNN library. If the cuDNN library could not be loaded, *-1* is returned.

  'cublas_loaded': Returns *'true'* if the cuBLAS library could be loaded.

  'cublas_version': Returns the version of the cuBLAS library. If the cuBLAS library could not be loaded, *-1* is returned.

- **Arm Compute Library support:**

  'arm_compute_loaded': Returns *'true'* if the Arm Compute Library could be loaded, *'false'* otherwise.

  'arm_compute_version': Returns the version of the Arm Compute Library. If the Arm Compute Library could not be loaded, an empty string is returned.

Parameters

▷ **`Query`** (input_control) . . . . . attribute.name(-array) ⇝ *string*

Desired system parameter.
Default: 'init_new_image'
List of values: `Query` ∈ {'3d_model_dir', 'alloctmp_max_blocksize', 'alloctmp_min_blocksize', 'alloctmp_max_used', 'available_license_files', 'available_parameters', 'avx_enable', 'avx_supported', 'avx2_enable', 'avx2_supported', 'avx2_gather_enable', 'avx2_gather_recommended', 'avx512f_enable', 'avx512f_supported', 'avx512dq_enable', 'avx512dq_supported', 'avx512bw_enable', 'avx512bw_supported', 'avx512cd_enable', 'avx512cd_supported', 'avx512vl_enable', 'avx512vl_supported', 'avx512vbmi_enable', 'avx512vbmi_supported', 'avx512ifma_enable', 'avx512ifma_supported', 'avx512vbmi2_enable', 'avx512vbmi2_supported', 'avx512vpopcntdq_enable', 'avx512vpopcntdq_supported', 'avx512bitalg_enable', 'avx512bitalg_supported', 'avx512vnni_enable', 'avx512vnni_supported', 'neon_enable', 'neon_supported', 'backing_store', 'border_shape_models', 'bundle_version', 'byte_order', 'calib_dir', 'cancel_draw_result', 'clip_region', 'clock_mode', 'cuda_loaded', 'cuda_version', 'cuda_devices', 'cudnn_loaded', 'cudnn_version', 'cublas_loaded', 'cublas_version', 'current_license_info', 'current_runlength_number', 'database', 'default_font', 'disabled_operators', 'dl_dir', 'do_low_error', 'empty_region_result', 'example_dir', 'expiration_date', 'extern_alloc_funct', 'extern_free_funct', 'failed_license_rechecks', 'filename_encoding', 'file_version', 'filter_dir', 'flush_file', 'flush_graphic', 'global_mem_cache', 'halcon_arch', 'halcon_dir', 'halcon_xl', 'halcon_64', 'height', 'help_dir', 'hthread_id', 'hostids', 'hostname', 'icon_name', 'image_cache_capacity', 'image_dir', 'image_dpi', 'init_new_image', 'int2_bits', 'int_zooming', 'is_license_valid', 'language', 'last_update', 'legacy_handle_mode', 'library_fullname', 'licensed_hostid', 'licensed_modules', 'licensed_product_edition', 'licensed_version', 'locale_codeset', 'locale_raw', 'lut_dir', 'max_connection', 'max_inp_ctrl_par', 'max_inp_obj_par', 'max_outp_ctrl_par', 'max_outp_obj_par', 'max_window', 'memory_allocator', 'memory_allocators_supported', 'mmx_enable', 'mmx_supported', 'neighborhood', 'no_object_result', 'num_proc', 'num_sys_proc', 'num_user_proc', 'ocr_dir', 'ocr_trainf_version', 'opengl_hidden_surface_removal_available', 'opengl_hidden_surface_removal_enable', 'opengl_compatibility_mode_enable', 'opengl_context_cache_enable', 'operating_system', 'operating_system_version', 'parallelize_operators', 'pregenerate_shape_models', 'processor_num', 'read_halcon_files_encoding_fallback', 'reentrant', 'revision', 'seed_rand', 'sse_enable', 'sse_supported', 'sse2_enable', 'sse2_supported', 'sse3_enable', 'sse3_supported', 'ssse3_enable', 'ssse3_supported', 'sse41_enable', 'sse41_supported', 'sse42_enable', 'sse42_supported', 'store_empty_region', 'supported_editions', 'system_time_base', 'temp_mem', 'temporary_mem_cache', 'temporary_mem_reservoir', 'temporary_mem_reservoir_size', 'thread_num', 'thread_pool', 'timer_mode', 'tuple_string_operator_mode', 'tsp_cancel_draw_result', 'tsp_clip_region', 'tsp_current_runlength_number', 'tsp_empty_region_result', 'tsp_height', 'tsp_init_new_image', 'tsp_legacy_handle_mode', 'tsp_neighborhood', 'tsp_no_object_result', 'tsp_store_empty_region', 'tsp_temporary_mem_cache', 'tsp_temporary_mem_cache_block_sizes', 'tsp_temporary_mem_reservoir', 'tsp_thread_num', 'tsp_tuple_string_operator_mode', 'tsp_used_split_levels', 'tsp_used_thread_num', 'tsp_width', 'unlicensed_operators', 'update_lut', 'use_window_thread', 'version', 'width', 'window_name', 'write_halcon_files_encoding', 'x_package'}

▷ **Information** (output_control) . . . . . attribute.value(-array) ⇝ *integer* / real / string
Current value of the system parameter.

Result

The operator get_system returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

reset_obj_db

Possible Successors

set_system

See also

set_system

Module

Foundation

get_system_info (: : Query : Information)

Get current value of system information without requiring a license.

The operator get_system_info returns information concerning the currently activated HALCON system parameters. Unlike the very similar get_system, get_system_info can be called without a valid license.

get_system_info supports the same parameters supported by get_system. For more information about the parameters, please refer to the reference documentation of get_system and set_system.

Parameters

▷ **Query** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Desired system parameter.
Default: 'available_parameters'
List of values: Query ∈ {'3d_model_dir', 'alloctmp_max_blocksize', 'alloctmp_min_blocksize', 'alloctmp_max_used', 'available_license_files', 'available_parameters', 'avx_enable', 'avx_supported', 'avx2_enable', 'avx2_supported', 'avx2_gather_enable', 'avx2_gather_recommended', 'avx512f_enable', 'avx512f_supported', 'avx512dq_enable', 'avx512dq_supported', 'avx512bw_enable', 'avx512bw_supported', 'avx512cd_enable', 'avx512cd_supported', 'avx512vl_enable', 'avx512vl_supported', 'avx512vbmi_enable', 'avx512vbmi_supported', 'avx512ifma_enable', 'avx512ifma_supported', 'avx512vbmi2_enable', 'avx512vbmi2_supported', 'avx512vpopcntdq_enable', 'avx512vpopcntdq_supported', 'avx512bitalg_enable', 'avx512bitalg_supported', 'avx512vnni_enable', 'avx512vnni_supported', 'neon_enable', 'neon_supported', 'backing_store', 'border_shape_models', 'bundle_version', 'byte_order', 'calib_dir', 'cancel_draw_result', 'clip_region', 'clock_mode', 'cuda_loaded', 'cuda_version', 'cuda_devices', 'cudnn_loaded', 'cudnn_version', 'cublas_loaded', 'cublas_version', 'current_license_info', 'current_runlength_number', 'database', 'default_font', 'disabled_operators', 'dl_dir', 'do_low_error', 'edition', 'empty_region_result', 'example_dir', 'expiration_date', 'extern_alloc_funct', 'extern_free_funct', 'failed_license_rechecks', 'filename_encoding', 'file_version', 'filter_dir', 'flush_file', 'flush_graphic', 'global_mem_cache', 'halcon_arch', 'halcon_dir', 'halcon_xl', 'halcon_64', 'height', 'help_dir', 'hthread_id', 'hostids', 'hostname', 'icon_name', 'image_cache_capacity', 'image_dir', 'image_dpi', 'init_new_image', 'int2_bits', 'int_zooming', 'is_license_valid', 'language', 'last_update', 'legacy_handle_mode', 'library_fullname', 'licensed_hostid', 'licensed_modules', 'licensed_product_edition', 'licensed_version', 'locale_codeset', 'locale_raw', 'lut_dir', 'max_connection', 'max_inp_ctrl_par', 'max_inp_obj_par', 'max_outp_ctrl_par', 'max_outp_obj_par', 'max_window', 'memory_allocator', 'memory_allocators_supported', 'mmx_enable', 'mmx_supported', 'neighborhood', 'no_object_result', 'num_proc', 'num_sys_proc', 'num_user_proc', 'ocr_dir', 'ocr_trainf_version',

'opengl_hidden_surface_removal_available', 'opengl_hidden_surface_removal_enable', 'opengl_compatibility_mode_enable', 'opengl_context_cache_enable', 'operating_system', 'operating_system_version', 'parallelize_operators', 'pregenerate_shape_models', 'processor_num', 'read_halcon_files_encoding_fallback', 'reentrant', 'revision', 'seed_rand', 'sse_enable', 'sse_supported', 'sse2_enable', 'sse2_supported', 'sse3_enable', 'sse3_supported', 'ssse3_enable', 'ssse3_supported', 'sse41_enable', 'sse41_supported', 'sse42_enable', 'sse42_supported', 'store_empty_region', 'system_time_base', 'temp_mem', 'temporary_mem_cache', 'temporary_mem_reservoir', 'temporary_mem_reservoir_size', 'thread_num', 'thread_pool', 'timer_mode', 'tuple_string_operator_mode', 'tsp_cancel_draw_result', 'tsp_clip_region', 'tsp_current_runlength_number', 'tsp_empty_region_result', 'tsp_height', 'tsp_init_new_image', 'tsp_legacy_handle_mode', 'tsp_neighborhood', 'tsp_no_object_result', 'tsp_store_empty_region', 'tsp_temporary_mem_cache', 'tsp_temporary_mem_cache_block_sizes', 'tsp_temporary_mem_reservoir', 'tsp_thread_num', 'tsp_tuple_string_operator_mode', 'tsp_used_split_levels', 'tsp_used_thread_num', 'tsp_width', 'unlicensed_operators', 'update_lut', 'use_window_thread', 'version', 'width', 'window_name', 'write_halcon_files_encoding', 'x_package'}

▷ **Information** (output_control) . . . . . attribute.value(-array) ⇝ *integer* / real / string
Current value of the system parameter.

Result

The operator `get_system_info` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`get_system`

Module

none

set_operator_timeout (: : OperatorName, Timeout, Mode :)

Set a timeout for an operator.

`set_operator_timeout` sets a timeout for the operator `OperatorName`. `Timeout` is given in seconds. Setting `Timeout` to *0* or *'off'* clears the timeout. The timeout is set for the current thread only and is not inherited by child threads.

Multiple operator names can be passed to `OperatorName`. In this case, one can either pass a single timeout in `Timeout`, which is then set for all operators, or one timeout per operator.

Two timeout modes are supported and selected with `Mode`:

'cancel' Cancels the execution of the operator after the timeout expires, raises the exception H_ERR_TIMEOUT (9400). All results computed by the operator are discarded.

'break' Break the execution of the operator after the timeout expires. The operator returns normally. If possible, results computed by the operator up to the point of the timeout are returned.

Attention

Note that not all operators support timeouts. If a given operator supports timeouts and which modes are supported is described in the execution information section of the reference documentation of the corresponding operator.

Also note that there is no hard guarantee about the granularity of the timeout. The granularity can depend on the operator, its input data and the speed of the device. It is typically finer than 10 ms.

Parameters

▷ **OperatorName** (input_control) . . . . . string(-array) ⇝ *string*
Operator for which the timeout shall be set.

▷ **Timeout** (input_control) . . . . . number(-array) ⇝ *real* / integer / string
Timeout in seconds.
Default: 1
Suggested values: Timeout ∈ {1, 0.1, 0.5, 'off', 0}

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Timeout mode to be set.
Default: 'cancel'
Suggested values: Mode ∈ {'cancel', 'break'}

Result

The operator set_operator_timeout returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: local (may only be called from the same thread in which the window, model, or tool instance was created).
- Processed without parallelization.

Alternatives

interrupt_operator

See also

interrupt_operator, get_current_hthread_id

Module

Foundation

set_system (: : SystemParameter, Value :)

Set HALCON system parameters.

The operator set_system allows to change different system parameters.

Parallelization information: Note that some of these parameters are set exclusively only, meaning they block other threads until no other operators are accessing the HALCON library.

- Parameters marked by a *) are globally valid and are set exclusively.
- Parameters without the mark *) and without the prefix *'tsp_'* are also globally valid but are not set exclusively, i.e., the operator call does not block other operator calls.
- Parameters with *'tsp_'* variant are thread specific.
  1. Without *'tsp_'*, they are set for all threads that are started after this call as well as for the current thread. Threads that are already running will not be affected.
  2. Parameters called with *'tsp_'* are only set for the current thread. Other threads that are already running will not be affected.

Available system parameters:

- **Graphic:**
  'backing_store': Determines whether the window content will be refreshed in case of overlapping of the windows. Some implementations of X Windows are faulty; in order to avoid these errors, the storing of contents can be deactivated. In some cases it may be advisable to deactivate the security mechanism, if e.g., performance / memory is critical.
  Value: *'true'*, *'false'*
  Default: *'true'*

'flush_graphic': After each HALCON operation which creates a graphic output, a flush operation will be executed in order to display the data immediately on screen. This is not necessary for all programs, e.g., if everything is done with the help of the mouse. In this case *'flush_graphic'* can be set to *'false'* to improve the runtime. Note that this parameter has no effect on Unix-like systems because there, the window managers flush the display buffer automatically.
Note: Using *'flush_graphic'* is not recommended, since it can lead to unexpected results. Instead, please set the parameter *'flush'* of `set_window_param` to *'false'* and call `flush_buffer` to transfer the buffer to the graphics window.
`Value`: *'true'*, *'false'*
Default: *'true'*

'int2_bits': Number of significant bits of int2 images. This number is used when scaling the gray values. If the value is set to -1 the gray values will be scaled automatically (default).
`Value`: *-1* or *9..16*
Default: *-1*

'icon_name': Name of iconified graphics windows under X Windows. By default the number of the graphics window is displayed.
`Value`: Name of iconified graphics window.
Default: *'default'*

'default_font' ***):** Default font for text output in text and graphic windows. Available fonts can be queried with `query_font`. The syntax of possible fontnames in `Value` is described at `set_font`.
`Value`: Name of Font.
Default: *'fixed'*

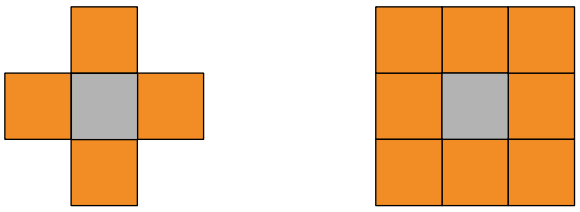
'update_lut': Determines whether the HALCON look-up color tables are adapted according to their environment or not. Such hardware-dependent color tables are no longer in use, so the parameter is obsolete. You can use the operator `query_lut` to query all possible look-up tables and set a specific one using `set_lut`.
`Value`: *'true'*, *'false'*
Default: *'false'*

'use_window_thread': On Microsoft Windows systems, calling any of the HALCON graphics operators from a thread other than the one that created the output window requires that there be an active message loop for that window. Setting this parameter to *'true'* will automatically open all top level HALCON graphics and text windows in a special window thread that will handle the message loop; all graphics operators directed at these windows will also be executed by that thread automatically. This parameter has no effect on non-Windows systems.
`Value`: *'true'*, *'false'*
Default: *'false'*

'x_package': The output of image data via the network may cause errors due to the heavy load on the computer or on the network. In order to avoid this, the data are transmitted in small packages. If the display is used locally (as opposed to using it via network), these units can be enlarged at will. This can lead to a recognizably improved output performance. On X Window systems the maximum request size is queried when the first window is opened.
`Value`: Package size (in bytes).
Default: *131072*

- **Image Processing:**

'neighborhood', *'tsp_neighborhood'*: This parameter checks either four pixels (in horizontal and vertical direction) or all eight neighboring pixels around one pixel. It is used with all operators that examine neighborhood relations: `connection`, `get_region_contour`, `get_region_polygon`, `get_region_thickness`, `boundary`, `paint_region`, `disp_region`, `fill_up`, `contlength`, `shape_histo_all`.
`Value`: *4* or *8*
Default: *8*



(1) (2)

(1) Four neighboring pixels, (2) eight neighboring pixels.

'init_new_image', *'tsp_init_new_image'*: Determines whether new images shall be set to 0 before using filters and thus to make sure that the values returned by a filtering are consistent if the program is executed repeatedly on systems with the same configuration. Note that this is not necessary if always the whole image is filtered or if the data of unfiltered image areas is unimportant.
Value: *'true'*, *'false'*
Default: *'true'*

'no_object_result', *'tsp_no_object_result'*: Determines how operations processing iconic objects shall react if the object tuple is empty (= no objects).
Available values for Value:

- *'true'*: The error will be ignored.
- *'exception'*: An exception is raised.

Default: *'true'*

'empty_region_result', *'tsp_empty_region_result'*: Controls the reaction of operators concerning input objects with empty regions which actually are not useful for such objects (e.g., certain region features, segmentation, etc.).
Available values for Value:

- *'true'*: The error will be ignored if possible.
- *'exception'*: An exception is raised.

Default: *'true'*

'store_empty_region', *'tsp_store_empty_region'*: Quite a number of operations will lead to the creation of objects with an empty region (= no image points) (e.g., intersection,threshold, etc.). This parameter determines whether the object with an empty region will be returned as a result ('true') or whether it will be ignored (*'false'*) that is no result will be returned.
Value: *'true'*, *'false'*
Default: *'true'*

'clip_region', *'tsp_clip_region'*: Determines whether the regions of iconic objects will be clipped to the currently used image size or not. This parameter can be used with operators that have a region as output, for example, gen_circle, gen_rectangle1 or dilation1.
See also: reset_obj_db
Value: *'true'*, *'false'*
Default: *'true'*

'int_zooming': Determines the precision of certain steps of the image processing. By setting *'int_zooming'* to *'true'* integer arithmetic or fast floating point arithmentic is used. High precision floating point arithmetic will be used by setting it to *'false'*. More information how a specific operator is affected by the parameter *'int_zooming'* can be found in the documentation of the directly concerned operator.
Value: *'true'*, *'false'*
Default: *'true'*

'pregenerate_shape_models': This parameter determines whether the shape models are completely pregenerated (*'true'*) or not (*'false'*) if this behavior is not explicitly specified in the shape model operator.
Affected operators:

- create_shape_model
- create_shape_model_xld
- create_scaled_shape_model
- create_scaled_shape_model_xld
- create_aniso_shape_model
- create_aniso_shape_model_xld

Value: *'true'*, *'false'*
Default: *'false'*

'border_shape_models': This parameter determines whether the shape models to be found may lie partially outside the image, i.e., whether they may cross the image border, (*'true'*) or not (*'false'*). Partially means that the model's origin still must be located within the image. A different origin set with set_shape_model_origin is not taken into account.
Affected operators:

- find_generic_shape_model

- find_shape_model
- find_shape_models
- find_scaled_shape_model
- find_scaled_shape_models
- find_aniso_shape_model
- find_aniso_shape_models
- find_planar_uncalib_deformable_model
- find_planar_calib_deformable_model
- find_local_deformable_model
- find_component_model (legacy)

Note, if a value has been set for shape models using set_generic_shape_model_param, the specific value set for the model dominates the system-wide setting. When searching multiple shape models *'border_shape_models'* is treated as *'true'* for all shape models even if *'border_shape_models'* only evaluates to *'true'* for one of the shape models in a search.
Value: *'true'*, *'false'*
Default: *'false'*

'opengl_context_cache_enable': This parameter determines if the graphic card context used in the operator render_object_model_3d is cached for future uses of this operator. The caching speeds up future uses of this operator but requires additional resources. To explicitly delete the context, set *'opengl_context_cache_enable'* to *'false'* and call render_object_model_3d once. *'opengl_context_cache_enable'* can only be set to *'true'* if the graphic card supports the rendering of 3D object models.
Value: *'true'*, *'false'*
Default: *'true'* if supported by the graphics card.

'opengl_hidden_surface_removal_enable': This parameter determines if the graphics card accelerated hidden surface removal is used in create_shape_model_3d, find_shape_model_3d, project_shape_model_3d, and project_object_model_3d. *'opengl_hidden_surface_removal_enable'* can only be set to *'true'* if the graphics card supports the acceleration of the hidden surface removal algorithm. The minimum requirements are OpenGL 2.0 and the extensions GL_EXT_framebuffer_object and GL_ARB_texture_float. Use get_system with the parameter Query set to *'opengl_hidden_surface_removal_available'* to determine if these requirements are fulfilled. Please note that these features are not available via Windows Remote Desktop or X11 forwarding.
Value: *'true'*, *'false'*
Default: *'true'* if supported by the graphics card.

'opengl_compatibility_mode_enable': This parameter determines if the visualization of 3D object models via disp_object_model_3d uses advanced OpenGL features or OpenGL 1.1 features only. If set to *'false'* HALCON determines the capabilities of the graphics card automatically. Set this parameter to *'true'*, if you face visualization problems.
Value: *'true'*, *'false'*
Default: *'false'*.

'image_dpi': This parameter determines the DPI resolution that is stored in image files written with write_image in a format that supports the storing of the DPI resolution.
Value: Resolution in DPI.
Default: *300*

'width', *'tsp_width'*: Determines the internal image width of the system. This size is used for clipping regions or implementing an assumption of memory sizes, if an operator has no further image information. This size is interpreted as maximum width of all HALCON image objects. It will be increased, if images of greater width will subsequently be instantiated. See also reset_obj_db. Note that the default value is *512* within HDevelop (also if you export your code), otherwise it is *128*.
Value: Internal image width.
Default: *128* or *512* (HDevelop)

'height', *'tsp_height'*: Determines the internal image height of the system. This size is used for clipping regions or implementing an assumption of memory sizes, if an operator has no further image information. This size is interpreted as maximum height of all HALCON image objects. It will be increased if images of greater height will subsequently be instantiated. See also reset_obj_db. Note that the default value is *512* within HDevelop (also if you export your code), otherwise it is *128*.

Value: Internal image height.
Default: *128* or *512* (HDevelop)

'current_runlength_number', *'tsp_current_runlength_number'*: Regions will be stored internally in a certain runlength code. This parameter can determine the maximum number of chords which may be used for representing a region. Please note that some operators raise the number automatically if necessary.
The value can be enlarged as well as reduced.
Value: Maximum number of chords.
Default: *50000*

- **Parallelization:**

'parallelize_operators' *): Determines whether HALCON uses an automatic parallelization to speed up the processing of operators on multiprocessor machines. This feature can be switched off by setting *'parallelize_operators'* to *'false'*. Even then, HALCON will remain reentrant (and thread-safe), unless the parameter *'reentrant'* is set to *'false'* via set_system. Changing the value for *'parallelize_operators'* can be helpful, for example, if HALCON operators are called by a multithreaded application that also automatically performs the scheduling and load balancing of operators and data. Then, it may be undesired that HALCON performs additional parallelization steps, which may disturb the application's scheduling and load balancing concepts. For a more detailed control of automatic parallelization, single methods of data parallelization can be switched on and off. *'split_tuple'* enables the tuple parallelization method, *'split_channel'* the parallelization on image channels, *'split_domain'* the parallelization on the image domain, and *'split_partial'* the partial, internal parallelization of an operator. A preceding '~' disables the corresponding method. The method strings can also be passed within a control tuple to switch on or off methods of automatic data parallelization at once. E.g., [*'split_tuple'*,*'split_channel'*, *'split_domain'*, *'split_partial'*] is equivalent to *'true'*.
The methods supported by a specific operator can be viewed in this reference manual or obtained via the operator get_operator_info with the parameter value *'parallel_method'*.
Value: *'true'*, *'false'*, *'split_tuple'*, *'split_channel'*, *'split_domain'*, *'split_partial'*, *' split_tuple'*, *' split_channel'*, *' split_domain'*, *' split_partial'*
Default: *'true'*

'reentrant' *): Determines whether HALCON must be reentrant for being used within a parallel programming environment (e.g., a multithreaded application). If it is set to *'true'*, HALCON internally uses synchronization mechanisms to protect shared data objects from concurrent accesses. Though this is inevitable with any effectively parallel working application, it may cause undesired overhead, if used within an application which works purely sequentially. The latter case can be signaled by setting *'reentrant'* to *'false'*. This switches off all internal synchronization mechanisms and thus reduces overhead. Of course, HALCON is then no longer thread-safe, which causes another side-effect: HALCON will as a consequence no longer use the internal parallelization of operators, because this requires reentrancy. Setting *'reentrant'* to *'true'* resets HALCON to its default state, i.e., it is reentrant (and thread-safe) and it uses the automatic parallelization to speed up the processing of operators on multiprocessor machines.
Value: *'true'*, *'false'*
Default: *'true'*

'thread_num' *), *'tsp_thread_num'*: Sets the number of threads used by the automatic operator parallelization (aop) of HALCON. The number includes the calling thread and is restricted to the number of processors for efficiency reasons. Decreasing the number of threads is helpful if processors are occupied by user worker threads besides the threads of the automatic parallelization. With this, the number of processing threads can be adapted to the number of processors for best efficiency. If a processor affinity was set for the HALCON process, the parameter value *'default'* resets the number of threads to the number of assigned processors. Else, *'default'* sets the number of threads to the number of processors. If the thread-specific variant is used, HALCON reserves the specified number of threads exclusively for the calling thread. If no thread number was set so far for a specific thread, get_system returns value -1 for parameter value *'tsp_thread_num'*. In case the sum of all thread-specific reserved aop threads exceeds the number of threads set by *'thread_num'*, the latter one is increased accordingly. Specifying thread number 1 switches off the automatic parallelization (thread-specific when indicated).
Value: *1* <= Value <= *'processor_num'*, *'default'*
Default: *'default'*

'thread_pool' *): Denotes whether HALCON always creates new threads for automatic parallelization (*'false'*) or uses an existing pool of threads (*'true'*). Using a pool is more efficient for automatic parallelization. When switching off automatic parallelization permanently, deactivating the pool can save resources of the operating system.

Value: *'true'*, *'false'*
Default: *'true'*

- **File:**

'flush_file': This parameter determines whether the output characters of the operator fwrite_string are displayed directly on the output medium. If set to *'false'* the characters will be flushed only after entering the operator fnew_line.
Value: *'true'*, *'false'*
Default: *'true'*

'ocr_trainf_version': This parameter determines the format that is used for writing an OCR training file. The operators write_ocr_trainf, write_ocr_trainf_image and concat_ocr_trainf write training data in ASCII format for version number *1* or in binary format for version number *2* and *3*. Version number *3* stores images of type byte and uint2. The binary version is faster in reading and writing data and stores training files more packed. The ASCII format is compatible to older HALCON releases. Depending on the file version, the OCR training files can be read by the following HALCON releases:

- *1* - All HALCON releases
- *2* - 7.0.2 and higher
- *3* - 7.1 and higher

Value: *1*, *2*, *3*
Default: *3*

'filename_encoding': This parameter only exists for legacy reasons and should not be used anymore. It determines how the HALCON library internally represents strings. The default is UTF-8, which supports all characters of the Unicode standard.
The string encoding of the data code readers, file handles, sockets, tuple operators, the HALCON/C++ interface and the HALCON/C interface is independent of this parameter. For each, the string encoding can individually be changed to local-8-bit encoding. See the respective documentation for details.
The setting of *'filename_encoding'* is a global setting and effects all threads. This means, changing this value in a thread while other threads are processing strings results in an undefined behavior which may lead to crashes. If a legacy application still requires the HALCON library to use local-8-bit encoding, see the Technical Updates for further explanations.
Value: *'utf8'* or *'locale'*
Default: *'utf8'*

'write_halcon_files_encoding': Starting with HALCON 18.05 all proprietary HALCON files which store user defined strings use UTF-8 as the default string encoding when the data is serialized or written into a file. The encoding concerns mainly HALCON tuples as well as OCR and OCV classifiers, training data, and sample identifiers, which all store user defined class or character names. This allows to share these files easily between different operating systems, countries, and locales. If such a file with special characters should be used by an application that was built with an older HALCON version (before HALCON 18.05) and relies on the locale encoding of the strings, this option can be used to force HALCON to write its files with the current locale system encoding instead of UTF-8. Note that the changed encoding concerns only files that store strings with special characters, i.e. characters that are not plain ASCII. In addition, this option does not affect user files which can be opened with open_file.
Value: *'locale'* or *'utf8'*
Default: *'utf8'*

'read_halcon_files_encoding_fallback': Older proprietary HALCON files (before HALCON 18.05) are read by default with the locale system encoding. If an old HALCON file was written with UTF-8 instead, this parameter can be used to change the encoding for reading. This concerns mainly the HALCON tuples as well as OCR and OCV classifiers, training data, and sample identifiers, which all store user defined class or character names. Note that this option has no effect on reading new HALCON files and on files which contain only plain ASCII characters. In addition, this option does not affect user files which can be opened with open_file.
Value: *'locale'* or *'utf8'*
Default: *'locale'*

'tuple_string_operator_mode', *'tsp_tuple_string_operator_mode'*: This parameters determines how UTF-8 strings are processed by tuple string operators, like tuple_ord, tuple_strlen, or tuple_substr, when *'filename_encoding'* is set to *'utf8'*, which is the default now. The standard behavior is that these strings are processed by characters or Unicode code points and not by bytes. I.e.,

the string length (operator tuple_strlen) of all characters that are build from a single Unicode code point is 1, and accessing the n-th element of a string will always return the corresponding Unicode code point, no matter how many bytes are used to represent that code point or the code points before in UTF-8. This behavior can be changed by setting *'tuple_string_operator_mode'* to *'byte'*. In that mode strings are processed byte by byte.
Value: *'codepoint'* or *'byte'*
Default: *'codepoint'*

- **Directories:**

'halcon_dir' ***):** This parameter returns the root directory of the HALCON installation.
Value: Name of the directory.

'example_dir' ***):** HDevelop example programs (e.g., loaded via the Browse HDevelop Program Examples dialog or via the HDevelop examples sections in the reference manual) will be looked for in the directory *'example_dir'*. Note that if you set *'example_dir'* to a directory that does not contain the installed HDevelop example programs, the above mentioned mechanisms to load HDevelop example programs will fail.
Value: Name of the directory.

'calib_dir' ***):** Camera parameter files (e.g., acquired via read_cam_par) will be looked for in the currently used directory and in *'calib_dir'* (if no absolute paths are indicated). More than one directory name can be indicated (search paths), separated by semicolons (Windows) or colons (Unix-like systems).
Value: Name of the directory.

'dl_dir' ***):** Deep learning models (e.g., acquired via read_dl_model) will be looked for in the currently used directory and in *'dl_dir'* (if no absolute paths are indicated). More than one directory name can be indicated (search paths), separated by semicolons (Windows) or colons (Unix-like systems).
Value: Name of the directory.

'filter_dir' ***):** Filter masks (e.g., used via convol_image) will be looked for in the currently used directory and in *'filter_dir'* (if no absolute paths are indicated). More than one directory name can be indicated (search paths), separated by semicolons (Windows) or colons (Unix-like systems).
Value: Name of the directory.

'image_dir' ***):** Image files (e.g., acquired via read_image and read_sequence) will be looked for in the currently used directory and in *'image_dir'* (if no absolute paths are indicated). More than one directory name can be indicated (search paths), separated by semicolons (Windows) or colons (Unix-like systems). The path can also be determined using the environment variable HALCONIMAGES.
Value: Name of the directory.

'3d_model_dir' ***):** 3D object model files (e.g., acquired via read_object_model_3d) will be looked for in the currently used directory and in *'3d_model_dir'* (if no absolute paths are indicated). More than one directory name can be indicated (search paths), separated by semicolons (Windows) or colons (Unix-like systems).
Value: Name of the directory.

'lut_dir' ***):** Color tables (set_lut) which are realized as an ASCII-file will be looked for in the currently used directory and in *'lut_dir'* (if no absolute paths are indicated). As default, HALCON will search the color tables in the sub-directory 'lut'.
Value: Name of the directory.

'help_dir' ***):** The online text files German or English.hlp, .sta, .key .num and .idx will be looked for in the currently used directory or in the path specified by *'help_dir'*. This system parameter is necessary for instance when using the operators get_operator_info and get_param_info. It can also be set by the environment variable HALCONROOT before initializing HALCON. In this case the variable must indicate the directory above the help directories (that is the HALCON home directory).
Value: Name of the directory.

'ocr_dir' ***):** OCR classifiers (e.g., acquired via read_ocr_class_knn) will be looked for in the currently used directory and in *'ocr_dir'* (if no absolute paths are indicated). More than one directory name can be indicated (search paths), separated by semicolons (Windows) or colons (Unix-like systems).
Value: Name of the directory.

- **Other:**

*'disabled_operators'***:** This parameter allows disabling of specific HALCON operators. This can be useful if a program allows running user-created HDevelop scripts, but certain operators should not be allowed for security reasons, e.g., the system_call operator. Any attempt to execute a disabled operator will

raise the exception H_ERR_OP_DISABLED (9055). Note that once an operator is disabled, it cannot be reenabled during the remaining lifetime of the process.
The value for this parameter is a tuple containing the names of all operators to be disabled.

'do_low_error' ***):** Determines the behavior regarding HALCON low level errors.

- If the parameter *'do_low_error'* is set to *'false'*, then no low level errors are thrown. However, the low level error messages still appear in the output console (a window containing a log of the most recent messages) that can be opened via the window menu.
- If the parameter *'do_low_error'* is set to *'disabled'*, the low level errors are suppressed and are not listed in the output console.
- If the parameter is set to *'stderr'*, the corresponding low level error message is printed to standard error.
- If it is set to *'message_box'*, then a message box containing the error text is opened (this functionality is implemented on Windows systems only).
- The parameter value *'callback'* can be used to determine a callback function, which should be called in case of a low level error. The address of this callback function is specified in the second index entry of the parameter `Value`.
  The signature of the callback function is the following:
  `Herror LowErrorCallbackProc(const char* err_text)`
  On Windows 32 bit systems, the __stdcall naming convention is used:
  `Herror (__stdcall LowErrorCallbackProc)(const char* err_text)`
  The parameter value *'callback'* can be used in HDevelop only if the callback procedure is set to 0, in which case the output of low level error messages is omitted.
  The string encoding of the message corresponds to the HALCON library encoding defined by *'filename_encoding'*.
- If low level error messages should be printed to file, then the parameter value *'file'* should be used together with a file handle referring to a corresponding file previously opened via `open_file`.
- The parameter values *'callback'* and *'file'* can be used only in combination with a corresponding procedure and file handle, respectively.
- With the exception of the parameter value *'false'*, the parameter *'do_low_error'* can be assigned multiple values. In case of a low level error, the corresponding actions are executed in the order of the values as passed in the input parameter tuple. If the tuple contains a certain parameter value multiple times, then only the first occurrence of the value in the tuple is taken into account, i.e., it is not possible to execute the same action multiple times in case of a low level error . If *'do_low_error'* is assigned multiple values or either the value *'file'* or *'callback'*, then it cannot be set in combination with other system parameters in the same call of `set_system`.
- Every setting of *'do_low_error'* via `set_system` overrides the previous setting of the parameter.
- The single parameter value *'true'* corresponds to the parameter value *'message_box'* on Windows systems and *'stderr'* on Unix-like systems.

`Value`: *'true'*, *'false'*, *'disabled'*, *'stderr'*, *'message_box'*, *'callback'*, *'file'*
Default: *'false'*

'cancel_draw_result', *'tsp_cancel_draw_result'***:** Many draw operators (like, e.g., `draw_region` or `draw_rectangle1`) can be canceled by ending the drawing with the right mouse button without new objects being drawn or existing objects being modified. The stop button of HDevelop can also be used to abort draw operators. The aborted draw operators return empty objects or empty tuples, respectively. *'cancel_draw_result'* controls the behavior of aborted draw operators.
The following values are available for `Value`:

- *'true'*: The draw operator returns no error.
- *'exception'*: An exception is raised.

Default: *'true'*

*'seed_rand'***:** Sets the seed of the thread specific random generator used in operators like, e.g., `tuple_rand`. If no seed has been set, the random generator is seeded with the current system time (amongst others) on the first call of one of the operators listed above. Afterwards the value used as seed can be queried with `get_system`.
This parameter accepts integer values or the string *'default'* to re-establish the default behavior.

*'cudnn_deterministic'***:** Determines whether the cuDNN library for deep learning uses deterministic algorithms to enforce bit-wise reproducibility "when executed on GPUs with the same architecture and the

same number of SMs." Please note that reproducibility is not guaranteed across different cuDNN versions. Further using deterministic algorithms can slow down the computation on some architectures.
Value: *'true'*, *'false'*
Default: *'false'*

'clock_mode': Determines the mode of the measurement of time intervals with count_seconds.
Value:

- *'performance_counter'* measures the elapsed system time with preferably high precision. HALCON uses the most accurate time measurement method provided by the operating system. If no high precision measurement method is available, a lower resolution method is used. On Windows systems the Performance Counter is used. For more information about the Performance Counter on Windows refer to the Microsoft Developer Network (MSDN) documentation. The Performance Counter provides the highest precision (below one millisecond). However, it may not work correctly on some systems due to issues with energy management and/or multithreading. If the Performance Counter is not supported by the operating system, the setting *'multimedia_timer'* applies. On Unix-like systems (e.g., Linux) the precision is also generally below one millisecond, and it may also be affected by problems due to energy management and/or multithreading. If you encounter such problems you should use the *'elapsed_time'* setting.
- *'elapsed_time'* measures the elapsed system time similar to the setting *'performance_counter'*. The difference is that other measurement methods are used which are less accurate but generally not much affected by problems due to energy management and/or multithreading. The precision is in general one millisecond. This setting should be used, if the measurements using *'performance_counter'* are unreliable and a precision of one millisecond is acceptable.
- *'multimedia_timer'* measures the elapsed system time using Multimedia Timers on Windows. Please refer to the Microsoft Developer Network (MSDN) documentation for more information on Multimedia Timers. This method has in general a resolution of one millisecond. If such a resolution cannot be achieved on your system you may use the Windows functions 'timerBeginPeriod' and 'timerEndPeriod' in order to increase the precision (see MSDN). On Unix-like systems there are no Multimedia Timers and therefor the setting *'performance_counter'* applies.
- *'processor_time'* measures the time the running HALCON process occupies the CPU. This kind of measuring time is independent of the CPU load caused by other processes, but it features a lower resolution on most systems and is therefore less accurate for smaller time intervals. Please note that the runtime of many applications is not only affected by CPU processing time. A lot of applications will be affected by input/output or memory management. For such applications the setting *'performance_counter'* or *'elapsed_time'* should be used instead.

Default: *'performance_counter'*
Please note that the settings *'performance_counter'* and *'elapsed_time'* measure the elapsed system time. As a consequence the measured time intervals are affected by the system load while measuring. If these settings are used to measure the runtime of a given application it should be ensured that other processes do not affect the measurement.

'timer_mode': Determines the mode of measurement used by the operators supporting timeouts.
Available values for Value:

- *'elapsed_time'*: see description in section *'clock_mode'*
- *'multimedia_timer'*: see description in section *'clock_mode'*
- *'performance_counter'*: see description in section *'clock_mode'*

Default: *'multimedia_timer'* on Windows systems, *'elapsed_time'* on Linux systems

'tsp_clear_extended_error': Clears the extended error information of the current thread.

'system_time_base': Determines how the system time is read. The system time can be read using get_system_time.
Value: *'UTC'* (Coordinated Universal Time), *'localtime'* (local time configured on the used system)
Default: *'localtime'*

'max_connection': Determines the maximum number of regions returned by connection. For Value=*0*, all regions are returned.
Value: >=*0*
Default: *0*

'extern_alloc_funct': Pointer to external function for memory allocation of result images. This function should have the following signature: 'void* ExternAllocFunc(size_t)'. If *0* is passed the HALCON allocation function will be used.

Value: Function pointer.
Default: *0*

'extern_free_funct' ***):** Pointer to external function for memory deallocation of result images. This function should have the following signature: 'void ExternFreeFunct(void*)'. If *0* is passed the HALCON deallocation function will be used.
Value: Function pointer.
Default: *0*

'image_cache_capacity' ***):** To speed up allocation of new images, HALCON does not free image memory of image objects but caches it to reuse it. With this parameter, you can set an upper limit in bytes for this HALCON image cache. Freed images are cached as long as the upper limit is not reached. This functionality can be switched off by setting *'image_cache_capacity'* to *0*.
Value: Limit for HALCON image cache.
Default: 16777216 (16MByte)

*'global_mem_cache'***:** Cache mode of global memory, i.e., memory that is visible beyond an operator. It specifies whether unused global memory should be cached for each thread separately (*'exclusive'*) or freed (*'idle'*). When using *'exclusive'* to cache memory, memory allocation and processing are sped up at the cost of memory consumption. With the action parameter *'cleanup'*, cached memory can be freed physically again. For more information on HALCON's caching mechanism, please refer to the technical note `HALCON Memory Management`.
Value: *'idle'*, *'exclusive'*, *'cleanup'*
Default: *'exclusive'*

'temporary_mem_cache' ***),** *'tsp_temporary_mem_cache'***:** This parameter controls the operating mode of the temporary memory cache. The temporary memory cache is used to speed up an application by caching memory used temporarily during the execution of an operator. For most applications the default setting (*'exclusive'*) will produce the best results. For more information on HALCON's caching mechanism, please refer to the technical note `HALCON Memory Management`.
The following modes are supported:

- *'idle'* The temporary memory cache is turned off. This mode will use the least memory, but will also reduce performance compared to the other modes.
- *'shared'* All temporary memory is cached globally in the temporary memory reservoir. This mode will use less memory than *'exclusive'* mode, but will also generally offer less performance.
- *'exclusive'* All temporary memory is cached locally for each thread. This mode will use the most memory, but will generally also offer the best performance.
- *'aggregate'* Temporary memory superblocks that are larger than the threshold set with the *'alloctmp_max_blocksize'* parameter are cached in the global memory reservoir, while all smaller superblocks are aggregated into a single superblock that is cached locally for each thread. If the global memory reservoir is disabled, the large superblocks are freed instead. The aggregated superblock will be sized according to the temporary memory usage the thread has seen so far, but it will not be larger than *'alloctmp_max_blocksize'* (if set) or smaller than *'alloctmp_min_blocksize'* (if set).

Note, the action of setting the temporary memory cache mode to *'idle'* (or *'false'*) itself runs exclusively, see the section "Parallelization information" above. In contrast, the action of setting the temporary memory cache mode to an other mode runs in reentrant mode, meaning without blocking other HALCON operators.
For backward compatibility, the values *'false'* and *'true'* are also accepted; they correspond to *'idle'* and *'exclusive'*, respectively.
Value: *'idle'*, *'shared'*, *'exclusive'*, or *'aggregate'*
Default: *'exclusive'*

*'temporary_mem_reservoir'***,** *'tsp_temporary_mem_reservoir'***:** If set to *'true'*, the global temporary memory reservoir is enabled. If it is set to *'false'*, it is disabled and any memory superblocks put into the reservoir are freed instead.
Any thread that needs new temporary memory will first check if the reservoir has a superblock available and only allocate more memory from the system if it does not. Superblocks are put into the reservoir by threads operating their caches in the *'shared'* or *'aggregate'* modes.
Note that the global setting of this parameter overrides the thread-specific setting, i.e., if *'temporary_mem_reservoir'* is set to *'false'*, *'tsp_temporary_mem_reservoir'* will have no effect.
Value: *'false'* or *'true'*
Default: *'true'*

'temporary_mem_reservoir_size': The maximum amount of memory the global temporary memory reservoir may cache, in bytes. If set to *-1*, the cache is limited only by the amount of available memory.
Value: *-1* or >= *0*
Default: *-1*

'alloctmp_min_blocksize', *'tsp_alloctmp_min_blocksize'*: Minimum size of memory superblocks used by the temporary memory cache, in bytes. (No effect if *'temporary_mem_cache'* == *'idle'*). With the default setting *-1*, HALCON will use a heuristic based on the current image size to determine a sensible superblock size. For most applications this will be the best choice. If the parameter is set to a value greater than or equal to zero, HALCON will use this size instead. Note that if an operator requires temporary memory objects larger than the *'alloctmp_min_blocksize'*, it will ignore this parameter.
Value: *-1* or >= *0*
Default: *-1*

'alloctmp_max_blocksize', *'tsp_alloctmp_max_blocksize'*: Setting this parameter to a value greater than or equal to zero will limit the size of temporary memory cache superblocks determined by a heuristic if *'alloctmp_min_blocksize'* is set to *-1* (no effect if *'temporary_mem_cache'* == *'idle'*). HALCON is still able to allocate superblocks larger than the limit, if necessary. With the default setting *-1*, the superblock size determined by the heuristic is not limited by HALCON. For most applications, this will be the best choice. In *'aggregate'* mode, this parameter additionally limits the size of the aggregated cache superblock, and determines the size threshold for superblocks to be placed into the temporary memory reservoir.
Value: *-1* or >= *0*
Default: *-1*

'database' *): Determines whether instantiated iconic objects should be listed in one of the five relations (gray-value data, region data, XLDs, iconic objects and object tuples) of the HALCON database. The relations can be used for, e.g., debugging purposes. See also count_relation, reset_obj_db. Note that collecting database information is not thread-safe when passing iconic objects between threads, meaning when deleting objects in a different thread than generating them.
Value: *'true'*, *'false'*
Default: *'false'*

'reset_used_modules' *): Set this to any value to reset the used modules reported by get_modules.

'mmx_enable': Flag, if MMX operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'mmx_supported'* == *'false'*, see also operator get_system)
Value: *'true'*, *'false'*
Default: *'true'* if CPU supports MMX, else *'false'*

'sse_enable': Flag, if SSE operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'sse_supported'* == *'false'*, see also operator get_system)
Value: *'true'*, *'false'*
Default: *'true'* if CPU supports SSE, else *'false'*

'sse2_enable': Flag, if SSE2 operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'sse2_supported'* == *'false'*, see also operator get_system)
Value: *'true'*, *'false'*
Default: *'true'* if CPU supports SSE2, else *'false'*

'sse3_enable': Flag, if SSE3 operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'sse3_supported'* == *'false'*, see also operator get_system)
Value: *'true'*, *'false'*
Default: *'true'* if CPU supports SSE3, else *'false'*

'ssse3_enable': Flag, if SSSE3 operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'ssse3_supported'* == *'false'*, see also operator get_system)
Value: *'true'*, *'false'*
Default: *'true'* if CPU supports SSSE3, else *'false'*

'sse41_enable': Flag, if SSE41 operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'sse41_supported'* == *'false'*, see also operator get_system)
Value: *'true'*, *'false'*
Default: *'true'* if CPU supports SSE41, else *'false'*

'sse42_enable': Flag, if SSE42 operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'sse42_supported'* == *'false'*, see also operator get_system)
Value: *'true'*, *'false'*
Default: *'true'* if CPU supports SSE42, else *'false'*

'avx_enable': Flag, if AVX operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX, else *'false'*

'avx2_enable': Flag, if AVX2 operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx2_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX2, else *'false'*

'avx2_gather_enable': Flag, if AVX2 gather operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx2_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: corresponds to *'avx2_gather_recommended'*

'avx512f_enable': Flag, if AVX-512F operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512f_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512F, else *'false'*

'avx512dq_enable': Flag, if AVX-512DQ operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512dq_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512DQ, else *'false'*

'avx512bw_enable': Flag, if AVX-512BW operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512bw_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512BW, else *'false'*

'avx512cd_enable': Flag, if AVX-512CD operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512cd_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512CD, else *'false'*

'avx512vl_enable': Flag, if AVX-512VL operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512vl_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512VL, else *'false'*

'avx512vbmi_enable': Flag, if AVX-512VBMI operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512vbmi_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512VBMI, else *'false'*

'avx512ifma_enable': Flag, if AVX-512IFMA operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512ifma_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512IFMA, else *'false'*

'avx512vbmi2_enable': Flag, if AVX-512VBMI2 operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512vbmi2_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512VBMI2, else *'false'*

'avx512vpopcntdq_enable': Flag, if AVX-512VPOPCNTDQ operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512vpopcntdq_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512VPOPCNTDQ, else *'false'*

'avx512bitalg_enable': Flag, if AVX-512BITALG operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512bitalg_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512BITALG, else *'false'*

'avx512vnni_enable': Flag, if AVX-512VNNI operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'avx512vnni_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports AVX-512VNNI, else *'false'*

'neon_enable': Flag, if NEON operations are used to accelerate selected image processing operators (*'true'*) or not (*'false'*). (No effect, if *'neon_supported'* == *'false'*, see also operator `get_system`)
`Value`: *'true'*, *'false'*
Default: *'true'* if CPU supports NEON, else *'false'*

'language' *): Language used for error messages.
`Value`: *'english'*, *'german'*.
Default: *'english'*

'add_progress_callback', *'remove_progress_callback'*: Adds or removes a callback function for operator progress and messages. The callback function is called by some operators in irregular intervals to signal their progress or a message to the application.
The signature of the callback is

```
void HProgressBarCallback(Hlong id, const char *operator_name,
double progress, const char *message)
```

On Windows 32 bit systems, the __stdcall naming convention is used:

```
void (__stdcall HProgressBarCallback)(Hlong id, const char
*operator_name, double progress, const char *message)
```

A corresponding delegate HalconAPI.HProgressBarCallback is defined for HALCON/.NET. Note that in order to pass a delegate to `set_system` you will need to call Marshal.GetFunctionPointerForDelegate and make sure the delegate is not garbage collected as long as the pointer remains in use.

```
public delegate void HProgressBarCallback(IntPtr id, string
operatorName, double progress, string message)
```

The parameters are:

- *'id'*: Thread-ID from which the operator was originally called. If the automatic operator parallelization is active, the callback might happen in a thread that is different from the one that was originally used to start the operator. Applications which display the progress in a GUI might have to consider this.
- *'operator_name'*: Name of the currently executed operator.
- *'progress'*: The interpretation of this parameter depends on its value. Between 0 and 1: The approximate current progress of the operator. -1: The operator shows only a message and does not set the progress. This parameter should be ignored. -2: The operator completed execution. This is the last callback for this operator and a chance for the application to perform cleanups. Note that this call happens only for operators that have called the callback at least once to set a progress or a message.
- *'message'*: Optional message in english that usually describes the progress of the operator such as the current error when training classifiers. If the operator has no message to show, this parameter can be NULL.

'legacy_handle_mode', *'tsp_legacy_handle_mode'*: Flag to activate or deactivate the legacy handle mode. If enabled, all handles returned by HALCON operators are converted into integers, and all HALCON operators accept integers as handles. Note that if the legacy handle mode is enabled, handles are no longer automatically cleared when they are overwritten. Instead, `clear_handle` or the corresponding clear operator of the handle type must be called.
Also note that the legacy handle mode is solely provided for running legacy code. It is not recommended to activate it in new code. New operators or language features might not work in legacy handle mode.
`Value`: *'true'*, *'false'*
Default: *'false'*

'gs1_syntax_dictionary' Provides a file containing the GS1 Application Identifier Syntax Dictionary to replace the list of GS1 Application Identifiers accepted by HALCON. The latest version can be obtained from the official GS1 website.
GS1 Application Identifiers are prefixes to define the meaning and format of data attributes.

The following values are supported:
- Filename of the file containing the syntax dictionary.
- *'default'*: Factory setting is recovered.

`Value`: Filename, *'default'*.

'memory_allocator' Set the memory allocator used to manage HALCON's heap. The available allocators can be queried with `get_system` using the parameter *'memory_allocators_supported'*. The following allocators are available:
- *'system'*: Use the system's default heap allocator. On Windows this is Win32's HeapAlloc, on other operating systems it is the C runtime's malloc function.
- *'mimalloc'*: Use the mimalloc heap allocator (see the official mimalloc documentation for more details).

Default: *'mimalloc'* on Windows systems, *'system'* otherwise
Please note that this setting does neither replace nor divert the system's standard heap allocator, it only changes which function HALCON calls to allocate memory internally. It also does not affect any of the third-party libraries HALCON uses. As a result, using any setting other than *'system'* will increase overall memory consumption, as the HALCON process will be using multiple separate heap managers. Depending on the operating system and application, this may improve performance.

Attention

Note that despite the information under 'Parallelization' concerning the multithreading type, not all parameters are reentrant. Operators that are followed by '*)' are set exclusively only.

Parameters

▷ **`SystemParameter`** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the system parameter to be changed.
Default: 'init_new_image'
List of values: `SystemParameter` ∈ {'3d_model_dir', 'add_progress_callback', 'alloctmp_min_blocksize', 'alloctmp_max_blocksize', 'avx_enable', 'avx2_enable', 'avx512f_enable', 'avx512dq_enable', 'avx512bw_enable', 'avx512cd_enable', 'avx512vl_enable', 'avx512vbmi_enable', 'avx512ifma_enable', 'avx512vbmi2_enable', 'avx512vpopcntdq_enable', 'avx512bitalg_enable', 'avx512vnni_enable', 'neon_enable', 'backing_store', 'border_shape_models', 'calib_dir', 'cancel_draw_result', 'clip_region', 'clock_mode', 'cudnn_deterministic', 'current_runlength_number', 'database', 'default_font', 'disabled_operators', 'dl_dir', 'do_low_error', 'empty_region_result', 'example_dir', 'extern_alloc_funct', 'extern_free_funct', 'filename_encoding', 'filter_dir', 'flush_file', 'flush_graphic', 'global_mem_cache', 'gs1_syntax_dictionary', 'height', 'help_dir', 'icon_name', 'image_cache_capacity', 'image_dir', 'image_dpi', 'init_new_image', 'int2_bits', 'int_zooming', 'language', 'legacy_handle_mode', 'lut_dir', 'max_connection', 'memory_allocator', 'mmx_enable', 'neighborhood', 'no_object_result', 'opengl_hidden_surface_removal_enable', 'opengl_compatibility_mode_enable', 'opengl_context_cache_enable', 'ocr_dir', 'ocr_trainf_version', 'parallelize_operators', 'pregenerate_shape_models', 'read_halcon_files_encoding_fallback', 'reentrant', 'remove_progress_callback', 'reset_used_modules', 'seed_rand', 'sse_enable', 'sse2_enable', 'sse3_enable', 'ssse3_enable', 'sse41_enable', 'sse42_enable', 'store_empty_region', 'system_time_base', 'temporary_mem_cache', 'temporary_mem_reservoir', 'temporary_mem_reservoir_size', 'thread_num', 'thread_pool', 'timer_mode', 'tuple_string_operator_mode', 'tsp_cancel_draw_result', 'tsp_clear_extended_error', 'tsp_alloctmp_max_blocksize', 'tsp_alloctmp_min_blocksize', 'tsp_clip_region', 'tsp_current_runlength_number', 'tsp_empty_region_result', 'tsp_height', 'tsp_init_new_image', 'tsp_legacy_handle_mode', 'tsp_neighborhood', 'tsp_no_object_result', 'tsp_store_empty_region', 'tsp_temporary_mem_cache', 'tsp_temporary_mem_reservoir', 'tsp_thread_num', 'tsp_tuple_string_operator_mode', 'tsp_width', 'update_lut', 'use_window_thread', 'width', 'write_halcon_files_encoding', 'x_package'}

▷ **`Value`** (input_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
New value of the system parameter.
Default: 'true'
Suggested values: `Value` ∈ {'true', 'false', 0, 4, 8}

Result

The operator `set_system` returns the value `2` (`H_MSG_TRUE`) if the parameters are correct. Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

reset_obj_db, get_system, set_check

See also

get_system, set_check, count_seconds

Module

Foundation

25.12 Serial

clear_serial (: : SerialHandle, Channel :)

Clear the buffer of a serial connection.

clear_serial discards data written to the serial device referred to by SerialHandle, but not transmitted (Channel = *'output'*), or data received, but not read (Channel = *'input'*), or performs both these operations at once (Channel = *'in_out'*).

Parameters

▷ **SerialHandle** (input_control) . . . . . serial ⇝ *handle*
Serial interface handle.

▷ **Channel** (input_control) . . . . . string ⇝ *string*
Buffer to be cleared.
Default: 'input'
List of values: Channel ∈ {'input', 'output', 'in_out'}

Result

If the parameters are correct and the buffers of the serial device could be cleared, the operator clear_serial returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SerialHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

open_serial

Possible Successors

read_serial, write_serial

See also

read_serial

Module

Foundation

close_serial (: : SerialHandle :)

Close a serial device.

close_serial closes a serial device that was opened with open_serial.

Parameters

▷ **SerialHandle** (input_control) . . . . . serial ⇝ *handle*
Serial interface handle.

Result

If the parameters are correct and the device could be closed, the operator close_serial returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SerialHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

open_serial

See also

open_serial, close_file

Module

Foundation

get_serial_param (: : SerialHandle : BaudRate, DataBits, FlowControl, Parity, StopBits, TotalTimeOut, InterCharTimeOut)

Get the parameters of a serial device.

get_serial_param returns the current parameter settings of the serial device passed in SerialHandle. For a description of the parameters of a serial device, see set_serial_param.

Parameters

▷ **SerialHandle** (input_control) . . . . . serial ⇝ *handle*
Serial interface handle.

▷ **BaudRate** (output_control) . . . . . integer ⇝ *integer*
Speed of the serial interface.

▷ **DataBits** (output_control) . . . . . integer ⇝ *integer*
Number of data bits of the serial interface.

▷ **FlowControl** (output_control) . . . . . string ⇝ *string*
Type of flow control of the serial interface.

▷ **Parity** (output_control) . . . . . string ⇝ *string*
Parity of the serial interface.

▷ **StopBits** (output_control) . . . . . integer ⇝ *integer*
Number of stop bits of the serial interface.

▷ **TotalTimeOut** (output_control) . . . . . integer ⇝ *integer*
Total timeout of the serial interface in ms.

▷ **InterCharTimeOut** (output_control) . . . . . integer ⇝ *integer*
Inter-character timeout of the serial interface in ms.

Result

If the parameters are correct and the parameters of the device could be read, the operator `get_serial_param` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SerialHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`open_serial`

Possible Successors

`read_serial`, `write_serial`

See also

`set_serial_param`

Module

Foundation

open_serial (: : PortName : SerialHandle)

Open a serial device.

`open_serial` opens a serial device. The name of the device is determined by the parameter `PortName` and is operating system specific. On Windows machines, *'COM1'-'COM4'* is typically used, while on Unix-like systems the serial devices usually are named *'/dev/tty*'*. The parameters of the serial device, e.g., its speed or number of data bits, are set to the system default values for the respective device after the device has been opened. They can be set or changed by calling `set_serial_param`.

Parameters

▷ **PortName** (input_control) . . . . . filename ⇝ *string*
Name of the serial port.
Default: 'COM1'
Suggested values: `PortName` ∈ {'COM1', 'COM2', 'COM3', 'COM4', '/dev/ttya', '/dev/ttyb', '/dev/tty00', '/dev/tty01', '/dev/ttyd1', '/dev/ttyd2', '/dev/cua0', '/dev/cua1'}

▷ **SerialHandle** (output_control) . . . . . serial ⇝ *handle*
Serial interface handle.

Result

If the parameters are correct and the device could be opened, the operator `open_serial` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_serial_param, read_serial, write_serial, close_serial

See also

set_serial_param, get_serial_param, open_file

Module

Foundation

read_serial (: : SerialHandle, NumCharacters : Data)

Read from a serial device.

read_serial tries to read NumCharacters from the serial device given in SerialHandle. The read characters are returned in Data as a tuple of integers. This allows to read NUL characters, which would otherwise be interpreted as the end of a string. If the timeout of the serial device has been set to a value greater than 0 with set_serial_param, read_serial waits at most as long for the arrival of the first character as indicated by the timeout. Otherwise, the operator returns immediately. In any case, the number of characters available at the time of return are passed back to the caller, i.e., fewer characters than requested can be returned. This can be checked by the length of the tuple Data.

Parameters

▷ **SerialHandle** (input_control) ..... serial ⇝ *handle*
Serial interface handle.

▷ **NumCharacters** (input_control) ..... integer ⇝ *integer*
Number of characters to read.
Default: 1
Suggested values: NumCharacters ∈ {1, 2, 3, 4, 5, 10, 20, 40, 100}

▷ **Data** (output_control) ..... integer(-array) ⇝ *integer*
Read characters (as tuple of integers).

Result

If the parameters are correct and the read from the device was successful, the operator read_serial returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_serial

See also

write_serial

Module

Foundation

set_serial_param (: : SerialHandle, BaudRate, DataBits, FlowControl, Parity, StopBits, TotalTimeOut, InterCharTimeOut :)

Set the parameters of a serial device.

set_serial_param can be used to set the parameters of a serial device. The parameter BaudRate determines the input and output speed of the device. It should be noted that not all devices support all possible speeds. The number of sent and received data bits is set with DataBits. The parameter FlowControl determines if and what kind of data flow control should be used. Software control (*'xon_xoff'*) and hardware control (*'cts_rts'* and *'dtr_dsr'*) can be used. Multiple values can be set separated by a space within one string (e.g., *'cts_rts dtr_dsr'*).

If and what kind of parity check of the transmitted data should be performed can be determined by `Parity`. The number of stop bits sent is set with `StopBits`. Finally, two timeout for reading from the serial device can be set. The parameter `TotalTimeOut` determines the maximum time, which may pass in `read_serial` until the first character arrives, independent of the actual number of characters requested. The parameter `InterCharTimeOut` determines the time which may pass between the reading of individual characters, if multiple characters are requested with `read_serial`. If one of the timeouts is set to *-1*, the system waits an arbitrary amount of time for the arrival of characters. If both timeouts are set to *0* the system doesn't wait and returns the available or none characters. Thus, on Windows systems, a total timeout of `TotalTimeOut` + n`InterCharTimeOut` results if n characters are to be read. On Unix-like systems, only one of the two timeouts can be set. Thus, if both timeouts are passed larger than -1, only the total timeout is used. The unit of both timeouts is milliseconds. It should be noted, however, that the timeout is specified in increments of one tenths of a second on Unix-like systems, i.e., the minimum timeout that has any effect is *100*. For each parameter, the current values can be left in effect by passing *'unchanged'*.

Parameters

▷ **`SerialHandle`** (input_control) . . . . . serial ⇝ *handle*
Serial interface handle.

▷ **`BaudRate`** (input_control) . . . . . integer ⇝ *integer* / string
Speed of the serial interface.
Default: 'unchanged'
List of values: `BaudRate` ∈ {50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 76800, 115200, 153600, 230400, 307200, 460800, 'unchanged'}

▷ **`DataBits`** (input_control) . . . . . integer ⇝ *integer* / string
Number of data bits of the serial interface.
Default: 'unchanged'
List of values: `DataBits` ∈ {5, 6, 7, 8, 'unchanged'}

▷ **`FlowControl`** (input_control) . . . . . string ⇝ *string*
Type of flow control of the serial interface.
Default: 'unchanged'
List of values: `FlowControl` ∈ {'none', 'cts_rts', 'dtr_dsr', 'xon_xoff', 'dtr_dsr xon_xoff', 'cts_rts xon_xoff', 'cts_rts dtr_dsr', 'cts_rts dtr_dsr xon_xoff', 'unchanged'}

▷ **`Parity`** (input_control) . . . . . string ⇝ *string*
Parity of the serial interface.
Default: 'unchanged'
List of values: `Parity` ∈ {'none', 'odd', 'even', 'unchanged'}

▷ **`StopBits`** (input_control) . . . . . integer ⇝ *integer* / string
Number of stop bits of the serial interface.
Default: 'unchanged'
List of values: `StopBits` ∈ {1, 2, 'unchanged'}

▷ **`TotalTimeOut`** (input_control) . . . . . integer ⇝ *integer* / string
Total timeout of the serial interface in ms.
Default: 'unchanged'
Suggested values: `TotalTimeOut` ∈ {-1, 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 'unchanged'}

▷ **`InterCharTimeOut`** (input_control) . . . . . integer ⇝ *integer* / string
Inter-character timeout of the serial interface in ms.
Default: 'unchanged'
Suggested values: `InterCharTimeOut` ∈ {-1, 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 'unchanged'}

Result

If the parameters are correct and the parameters of the device could be set, the operator `set_serial_param` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SerialHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

open_serial, get_serial_param

Possible Successors

read_serial, write_serial

See also

get_serial_param

Module

Foundation

write_serial (: : SerialHandle, Data :)

Write to a serial connection.

write_serial writes the characters given in Data to the serial device given by SerialHandle. The data to be written is passed as a tuple of integers. This allows to write NUL characters, which would otherwise be interpreted as the end of a string. write_serial always waits until all data has been transmitted, i.e., a timeout for writing cannot be set.

Parameters

▷ **SerialHandle** (input_control) . . . . . serial ⇝ *handle*
Serial interface handle.

▷ **Data** (input_control) . . . . . integer(-array) ⇝ *integer*
Characters to write (as tuple of integers).

Result

If the parameters are correct and the write to the device was successful, the operator write_serial returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_serial

See also

read_serial

Module

Foundation

25.13 Serialized Item

clear_serialized_item (: : SerializedItemHandle :)

Delete a serialized item.

clear_serialized_item deletes a serialized item, which is passed by the handle SerializedItemHandle (see fwrite_serialized_item for an introduction of the basics of serialization). If a serialized item is created only by a data pointer, it will not be deleted. Such a data pointer, that

points to the beginning of serialized item can be created by the operator get_serialized_item_ptr. After calling clear_serialized_item, the handle of the serialized item becomes invalid.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item(-array) ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator clear_serialized_item returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- SerializedItemHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

fwrite_serialized_item, receive_serialized_item

Module

Foundation

create_serialized_item_ptr (: : Pointer, Size,
Copy : SerializedItemHandle)

Create a serialized item.

It is not recommended to use the operator create_serialized_item_ptr in HDevelop.

create_serialized_item_ptr creates a serialized item and returns the handle SerializedItemHandle (see fwrite_serialized_item for an introduction of the basics of serialization). The data pointer, i.e., the beginning of the serialized item, is passed by the parameter Pointer. The size of the serialized item in bytes is passed by parameter Size. The behavior of the operator is controlled by the parameter Copy. If Copy is set to *'true'*, a new serialized item is created and the values of the existing serialized item are copied. If Copy is set to *'false'*, only the data pointer of the input serialized item is stored, i.e., the actual data are not copied, which leads to shorter execution times.

Attention

If the parameter Copy is set to *'false'*, the data pointer of the serialized item must not be deleted during using the new serialized item.

Parameters

▷ **Pointer** (input_control) . . . . . pointer ⇝ *integer*
Data pointer of the serialized item.

▷ **Size** (input_control) . . . . . integer ⇝ *integer*
Size of the serialized item.

▷ **Copy** (input_control) . . . . . string ⇝ *string*
Copy mode of the serialized item.
Default: 'true'
List of values: Copy ∈ {'true', 'false'}

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `create_serialized_item_ptr` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

`get_serialized_item_ptr`

Module

Foundation

decrypt_serialized_item (: : EncryptedItemHandle, DecryptionParam : SerializedItemHandle)

Decrypt an encrypted item.

The operator `decrypt_serialized_item` decrypts the encrypted item `EncryptedItemHandle` to the serialized item `SerializedItemHandle` using the secret in the dictionary `DecryptionParam`. The secret must be passed in the form of a password string in the dictionary key 'password'.

Parameters

▷ **EncryptedItemHandle** (input_control) ..... encrypted_item ⇝ *handle*
Encrypted item handle.

▷ **DecryptionParam** (input_control) ..... dict ⇝ *handle*
Parameters for the decryption.
Default: []

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Serialized item handle.

Result

The operator `decrypt_serialized_item` returns the value `2` (`H_MSG_TRUE`) if the passed handles and the parameters are valid. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

`encrypt_serialized_item`, `read_encrypted_item`

Possible Successors

`deserialize_handle`

Module

Foundation

encrypt_serialized_item (: : SerializedItemHandle, EncryptionParam : EncryptedItemHandle)

Encrypt a serialized item.

The operator encrypt_serialized_item encrypts the serialized item SerializedItemHandle to the encrypted item EncryptedItemHandle using the secret in the dictionary EncryptionParam. The secret must be passed in the form of a password string in the dictionary key 'password'.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Serialized item handle.

▷ **EncryptionParam** (input_control) ..... dict ⇝ *handle*
Parameters for the encryption.
Default: []

▷ **EncryptedItemHandle** (output_control) ..... encrypted_item ⇝ *handle*
Encrypted item handle.

Result

The operator encrypt_serialized_item returns the value 2 (H_MSG_TRUE) if the passed handles and the parameters are valid. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

serialize_handle

Possible Successors

write_encrypted_item, decrypt_serialized_item

Module

Foundation

fread_serialized_item (: : FileHandle : SerializedItemHandle)

Read a serialized item from a file.

fread_serialized_item reads a serialized item, that was written by fwrite_serialized_item from the current input file with the file handle FileHandle (see fwrite_serialized_item for an introduction of the basics of serialization). For this, a serialized item is created and the values read from file are stored in the serialized item. The operator returns the handle of the serialized item in the parameter SerializedItemHandle. The file can be opened by the operator open_file. Note that the file must be open in binary format. For reading more than on item from one file the operator fread_serialized_item must be called several times.

Parameters

▷ **FileHandle** (input_control) ..... file ⇝ *handle*
File handle.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator fread_serialized_item returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised. The error code 9581 indicates that the end of file is reached before a serialized item could be read.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

open_file, fwrite_serialized_item

Possible Successors

close_file, get_serialized_item_ptr, deserialize_shape_model, deserialize_ncc_model

Module

Foundation

fwrite_serialized_item (: : FileHandle, SerializedItemHandle :)

Write a serialized item to a file.

fwrite_serialized_item writes a serialized item to the output file with the handle FileHandle. The serialized item is defined by the handle SerializedItemHandle. The file can be opened by the operator open_file. Note that the file must be open in binary format. For writing more than one item in one file, the operator fwrite_serialized_item must be called several times. A serialized item can be read with fread_serialized_item.

Basics of the serialization and deserialization

For writing or sending iconic objects, data, or models to a file or to another HALCON process over a socket connection, first, the iconic objects, data, or models must be converted to serialized items. For this purpose, a lot of operators are available. E.g., the operator serialize_shape_model performs the serialization for the shape model . After this, a serialized item can be written by fwrite_serialized_item in a file or can be transferred by send_serialized_item to another HALCON process over a socket connection.

To deserialize a serialized item, first a serialized item is read from a file (see fread_serialized_item) or is received over the socket connection from another HALCON process (see receive_serialized_item). There are also operators to convert the serialized item to the original format, i.e., to the iconic object, the data, or the model. These operators deserialize the iconic object, the data, or the model. E.g., the operator deserialize_shape_model deserializes the item for a serialized shape model .

Parameters

▷ **FileHandle** (input_control) ..... file ⇝ *handle*
File handle.

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator fwrite_serialized_item returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_file, serialize_shape_model, serialize_ncc_model

Possible Successors

close_file, fread_serialized_item

Module

Foundation

get_serialized_item_ptr (: : SerializedItemHandle : Pointer, Size)

Access the data pointer of a serialized item.

It is not recommended to use the operator get_serialized_item_ptr in HDevelop.

get_serialized_item_ptr returns the data pointer to the serialized item which is passed by the handle SerializedItemHandle (see fwrite_serialized_item for an introduction of the basics of serialization). The data pointer, i.e., the beginning of the serialized item is returned by the parameter Pointer. The size of the serialized item in bytes is passed by the parameter Size.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **Pointer** (output_control) . . . . . pointer ⇝ *integer*
Data pointer of the serialized item.

▷ **Size** (output_control) . . . . . integer ⇝ *integer*
Size of the serialized item.

Result

If the parameters are valid, the operator get_serialized_item_ptr returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_serialized_item_ptr

Module

Foundation

25.14 Sockets

close_socket (: : Socket :)

Close a socket.

close_socket closes a socket that was previously opened with open_socket_accept, open_socket_connect, or socket_accept_connect. For a detailed example, see open_socket_accept.

Parameters

▷ **Socket** (input_control) . . . . . socket ⇝ *handle*
Socket number.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- Socket

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

open_socket_accept, open_socket_connect, socket_accept_connect

Module

Foundation

get_next_socket_data_type (: : Socket : DataType)

Determine the HALCON data type of the next socket data.

get_next_socket_data_type returns the data type of the next data that are present on the socket Socket and returns it in DataType. The possible values for DataType are:

'no_data': No data are present.

'no_halcon_data': Some data are present, but they are not HALCON data.

'tuple': The next data is a tuple.

'region': The next data is a region object.

'image': The next data is an image object.

'xld_cont': The next data is an XLD contour.

'xld_poly': The next data is an XLD polygon.

'xld_para': The next data is an XLD parallel.

'xld_mod_para': The next data is a modified XLD parallel.

'xld_ext_para': The next data is an extended XLD parallel.

'serialized_item': The next data is a serialized item.

Parameters

▷ **Socket** (input_control) ..... socket ⇝ *handle*
Socket number.

▷ **DataType** (output_control) ..... string ⇝ *string*
Data type of next HALCON data.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

send_image, receive_image, send_region, receive_region, send_tuple, receive_tuple, send_serialized_item, receive_serialized_item

Module

Foundation

get_socket_descriptor (: : Socket : SocketDescriptor)

Get the socket descriptor of a socket used by the operating system.

get_socket_descriptor returns the socket descriptor used by the operating system for the socket connection that is passed in Socket. The socket descriptor can be used in operating system calls such as select, read, write, recv, or send.

Parameters

▷ **Socket** (input_control) . . . . . socket ⇝ *handle*
Socket number.

▷ **SocketDescriptor** (output_control) . . . . . integer ⇝ *integer*
Socket descriptor used by the operating system.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_accept, open_socket_connect, socket_accept_connect

See also

set_socket_param

Module

Foundation

get_socket_param (: : Socket, GenParamName : GenParamValue)

Get the value of a socket parameter.

get_socket_param reads the GenParamValue of the GenParamName for the specified socket connection. Available parameters are *'timeout'*, *'address_info'*, *'SO_SNDBUF'*, *'SO_RCVBUF'*, *'SO_BROADCAST'*, and *'TCP_NODELAY'*.

The parameter *'address_info'* returns the IP address and port of the local and the remote side of the specified socket connection.

See set_socket_param for a description of the other values.

Parameters

▷ **Socket** (input_control) . . . . . socket ⇝ *handle*
Socket number.

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Name of the socket parameter.
List of values: GenParamName ∈ {'timeout', 'address_info', 'SO_SNDBUF', 'SO_RCVBUF', 'SO_BROADCAST', 'TCP_NODELAY'}

▷ **GenParamValue** (output_control) . . . . . string ⇝ *string* / real / integer
Value of the socket parameter.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect

Possible Successors

set_socket_param

See also

set_socket_param

Module

Foundation

open_socket_accept (: : Port, GenParamName, GenParamValue : AcceptingSocket)

Open a socket that accepts connection requests.

open_socket_accept opens a socket that will accept incoming connection requests. This operator is the necessary first step in the establishment of a communication channel between two HALCON processes. The socket listens for incoming connection requests on the port number given by Port.

It is possible to specify a Port of 0, triggering implementation-defined behavior. On most operating systems, using a port number of 0 instructs the operating system to choose a free port randomly.

The following parameters can be set for GenParamName.

'protocol': Specifies the protocol to be used. The *'HALCON'* protocol without a specific address family, will use IPv4 or IPv6 automatically depending on the network configuration of the computer. To use a specific address family a '4' or '6' (for IPv4 or IPv6, respectively) must be appended to the name of the protocol. For example, *'HALCON4'* designates a HALCON connection over IPv4. Possible values for a generic socket communication are *'UDP'* and *'TCP'* which also support appended '4' and '6'. Both communication partners must use the same protocol. To exchange data using generic sockets use send_data and receive_data only.

Default: *'HALCON'*

'address': Instructs the socket to accept only connection requests addressed to a specific address. This address is specified as either an IPv4 or IPv6 address in numerical form, or a hostname. For example, *'localhost'* would instruct the socket to accept only connections addressed to the address *'localhost'*, which normally maps to the local loopback network interface *'127.0.0.1'*.

'timeout': Sets a timeout for this socket. The timeout is given in seconds or as the string *'infinite'*. It is used especially as timeout and waiting mode in a following socket_accept_connect call with the parameter Wait set to *'auto'*.

'string_encoding': Sets the string encoding that is expected when sending and receiving strings with this socket. It is used for send_tuple and receive_tuple as well as send_data and receive_data. When the protocol *'HALCON'* is used, the setting only has effect on connections to HALCON versions prior to 18.11. If the connection partner is detected to have HALCON version 18.11 or newer, UTF-8 is used to encode tuples.

'reuseaddr': Controls the setting of the socket option SO_REUSEADDR. When *'reuseaddr'* is set to *'false'*, SO_REUSEADDR will not be set. On Windows Systems, conflicting access on a single port will only be detected, when *'reuseaddr'* is set to *'false'*.

Default: *'true'*

'tls_enable': Controls the use of Transport Layer Security (TLS) for the new Socket. When *'tls_enable'* is set to *'true'*, the created socket will accept TLS secured connections only. Furthermore, both generic parameters *'tls_private_key'* and *'tls_certificate'* must be set, each with a path to the private certificate key and the path to the public server certificate. Private key and server certificate must be provided in PEM-format.

Default: *'false'*

The accepting socket is returned in AcceptingSocket. open_socket_accept returns immediately without waiting for a request from another process, which must be initiated by calling open_socket_connect. This allows multiple other processes to connect to the particular HALCON process that calls open_socket_accept. To accept an incoming *'HALCON'* or *'TCP'* connection request, socket_accept_connect must be called to obtain a socket for the final communication.

Parameters

▷ **Port** (input_control) ..... number ⇝ *integer*
Port number.
Default: 3000
Suggested values: Port ∈ {3000, 4570}
Value range: 1024 ≤ Port ≤ 65535
Minimum increment: 1
Recommended increment: 1

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that can be adjusted for the socket.
Default: []
List of values: GenParamName ∈ {'address', 'protocol', 'timeout', 'string_encoding', 'tls_enable', 'tls_private_key', 'tls_certificate'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer
Values of the generic parameters that can be adjusted for the socket.
Default: []
Suggested values: GenParamValue ∈ {0, 3.0, 'infinite', 'HALCON', 'UDP', 'TCP', 'HALCON4', 'UDP4', 'TCP4', 'HALCON6', 'UDP6', 'TCP6', 'utf8', 'locale', 'ignore', 'true', 'false'}

▷ **AcceptingSocket** (output_control) . . . . . socket ⇝ *handle*
Socket number.

Example

```
* Process 1
dev_set_colored (12)
open_socket_accept (3000, [], [], AcceptingSocket)
* Busy wait for an incoming connection
dev_error_var (Error, 1)
dev_set_check ('~give_error')
OpenStatus := 5
while (OpenStatus != 2)
    socket_accept_connect (AcceptingSocket, 'false', Socket)
    OpenStatus := Error
    wait_seconds (0.2)
endwhile
dev_set_check ('give_error')
* Connection established
receive_image (Image, Socket)
threshold (Image, Region, 0, 63)
send_region (Region, Socket)
receive_region (ConnectedRegions, Socket)
area_center (ConnectedRegions, Area, Row, Column)
send_tuple (Socket, Area)
send_tuple (Socket, Row)
send_tuple (Socket, Column)
close_socket (Socket)
close_socket (AcceptingSocket)

* Process 2
dev_set_colored (12)
open_socket_connect ('localhost', 3000, [], [], Socket)
read_image (Image, 'fabrik')
send_image (Image, Socket)
receive_region (Region, Socket)
connection (Region, ConnectedRegions)
send_region (ConnectedRegions, Socket)
receive_tuple (Socket, Area)
receive_tuple (Socket, Row)
receive_tuple (Socket, Column)
close_socket (Socket)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

socket_accept_connect

See also

open_socket_connect, close_socket, get_socket_param, set_socket_param, send_image, receive_image, send_region, receive_region, send_tuple, receive_tuple, send_data, receive_data

Module

Foundation

open_socket_connect (: : HostName, Port, GenParamName, GenParamValue : Socket)

Open a socket and connect it to an accepting socket.

open_socket_connect opens a connecting socket to an accepting socket on the computer HostName, which listens on port Port.

The following parameters can be set for GenParamName.

'protocol': Specifies the protocol to be used. The *'HALCON'* protocol without a specific address family, will use IPv4 or IPv6 automatically depending on the network configuration of the computer. To use a specific address family a '4' or '6' (for IPv4 or IPv6, respectively) must be appended to the name of the protocol. For example, *'HALCON4'* designates a HALCON connection over IPv4. Possible values for a generic socket communication are *'UDP'* and *'TCP'* which also support appended '4' and '6'. Alternatively, the usage of addresses for IPv4 (e.g., *'127.0.0.1'*) or IPv6 (e.g., *'::1'*) for HostName determines the address family to be used. Both communication partners must use the same protocol. To exchange data using generic sockets use send_data and receive_data only.

Default: *'HALCON'*

'timeout': Sets a timeout for this operation. The timeout is given in seconds or as the string *'infinite'*.

'tls_enable': Controls the use of Transport Layer Security (TLS) for the new socket connection. When *'tls_enable'* is set to *'true'*, a connection attempt using TLS is undertaken. This requires TLS support on the server side. The certificate to optionally check against is specified with the generic parameter *'tls_certificate'*. Without a certificate, the connection will still be encrypted, but the authenticity of the server can not be verified.

Default: *'false'*

'tls_certificate': Specify the path to a certificate to check the authenticity of the server of connection using Transport Layer Security (TLS). The certificate must be provided in PEM-format. This parameter is ignored if *'tls_enable'* is not set to *'true'*.

'tsl_sni': Set the "server name indication" to use for the TLS connection. This tells the server for which domain the certificate should be valid, which is important if multiple domains are hosted on the same server. This parameter is ignored if *'tls_enable'* is not set to *'true'*.

'string_encoding': Sets the string encoding that is expected when sending and receiving strings with this socket. It is used for send_tuple and receive_tuple as well as send_data and receive_data. When the protocol *'HALCON'* is used, the setting only has effect on connections to HALCON versions prior to 18.11. If the connection partner is detected to have HALCON version 18.11 or newer, UTF-8 is used to encode tuples.

For the *'HALCON'* protocol the listening socket must have been created earlier with the operator open_socket_accept in another HALCON process. To establish the connection, the HALCON process, in which the accepting socket resides, must call socket_accept_connect. For a detailed example, see open_socket_accept.

For generic sockets the socket to connect to can be any socket from the same protocol type.

Parameters

▷ **HostName** (input_control) . . . . . string ⇝ *string*
Hostname of the computer to connect to.
Default: 'localhost'

▷ **Port** (input_control) . . . . . number ⇝ *integer*
Port number.
Suggested values: Port ∈ {3000, 4570}
Value range: 1024 ≤ Port ≤ 65535
Minimum increment: 1
Recommended increment: 1

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters that can be adjusted for the socket.
Default: []
List of values: GenParamName ∈ {'timeout', 'protocol', 'string_encoding', 'tls_enable', 'tls_certificate', 'tls_sni'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real / integer
Values of the generic parameters that can be adjusted for the socket.
Default: []
Suggested values: GenParamValue ∈ {0, 3.0, 'infinite', 'HALCON', 'UDP', 'TCP', 'HALCON4', 'UDP4', 'TCP4', 'HALCON6', 'UDP6', 'TCP6', 'utf8', 'locale', 'ignore', 'true', 'false'}

▷ **Socket** (output_control) . . . . . socket ⇝ *handle*
Socket number.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

send_image, receive_image, send_region, receive_region, send_tuple, receive_tuple, send_data, receive_data

See also

open_socket_accept, socket_accept_connect, get_socket_param, set_socket_param, close_socket

Module

Foundation

receive_data (: : Socket, Format : Data, From)

Receive arbitrary data from external devices or applications using a generic socket connection.

receive_data receives arbitrary data over a generic socket connection. The received data is converted from a binary network packet to a value (or a tuple of values) using the parameter Format as specification and is well-suited to communicate with external devices or applications. This operator does not support the standard *'HALCON'* protocol, but is intended for arbitrary data transfer.

The received data is converted to a value or tuple of values using the parameter Format. It is possible to specify multiple formats. In this case the From parameter will contain a 3rd value which tells you which format has been used to convert the data. To decide which format to use the size of the necessary data for each format is calculated initially. When data is received, the first format string with the matching size is used to convert the data to values.

The parameter From contains the IP address or hostname and port of the communication partner. For UDP connections it can be used in the send_data operator to send a response.

Please see send_data for a detailed description of the format.

Parameters

▷ **Socket** (input_control) . . . . . socket ⇝ *handle*
Socket number.

▷ **Format** (input_control) . . . . . string(-array) ⇝ *string*
Specification how to convert the data to tuples.
Default: 'z'

▷ **Data** (output_control) . . . . . string ⇝ *string* / real / integer / handle
Value (or tuple of values) holding the received and converted data.

▷ **From** (output_control) . . . . . string ⇝ *string* / integer
IP address or hostname and network port of the communication partner.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect, get_socket_param, set_socket_param

Possible Successors

close_socket

See also

send_data

Module

Foundation

receive_image (: Image : Socket :)

Receive an image over a socket connection.

receive_image reads an image object that was sent over the socket connection determined by Socket by another HALCON process using the operator send_image. If no image has been sent, the HALCON process calling receive_image blocks until enough data arrives. For a detailed example, see open_socket_accept.

Attention

'int8' images can be received by 64 bit systems only!

Parameters

▷ **Image** (output_object) . . . . . image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Received image.

▷ **Socket** (input_control) . . . . . socket ⇝ *handle*
Socket number.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect, get_socket_param, set_socket_param

See also

send_image, send_region, receive_region, send_tuple, receive_tuple, get_next_socket_data_type

Module

Foundation

receive_region (: Region : Socket :)

Receive regions over a socket connection.

receive_region reads a region object that was sent over the socket connection determined by Socket by another HALCON process using the operator send_region. If no regions have been sent, the HALCON process calling receive_region blocks until enough data arrives. For a detailed example, see open_socket_accept.

Parameters

▷ **Region** (output_object) ..... region(-array) ⇝ *object*
Received regions.

▷ **Socket** (input_control) ..... socket ⇝ *handle*
Socket number.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect, get_socket_param, set_socket_param

See also

send_region, send_image, receive_image, send_tuple, receive_tuple, get_next_socket_data_type

Module

Foundation

receive_serialized_item (: : Socket : SerializedItemHandle)

Receive a serialized item over a socket connection.

receive_serialized_item receives a serialized item that was sent over the socket connection determined by Socket of another HALCON process using the operator send_serialized_item (see fwrite_serialized_item for an introduction of the basics of serialization). If no serialized item has been sent, the HALCON process calling receive_serialized_item blocks until enough data arrives. The data is stored in a new created serialized item. The operator returns the handle of the serialized item in parameter SerializedItemHandle. For a detailed example showing the use of a socket connection, see open_socket_accept.

Parameters

▷ **Socket** (input_control) ..... socket ⇝ *handle*
Socket number.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator receive_serialized_item returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

send_serialized_item

Possible Successors

get_serialized_item_ptr, deserialize_matrix, deserialize_metrology_model

Module

Foundation

receive_tuple (: : Socket : Tuple)

Receive a tuple over a socket connection.

receive_tuple reads a Tuple that was sent over the socket connection determined by Socket by another HALCON process using the operator send_tuple. If no tuple has been sent, the HALCON process calling receive_tuple blocks until enough data arrives. For a detailed example, see open_socket_accept.

Attention

receive_tuple does not allow receiving tuples containing HALCON handles.

Parameters

▷ **Socket** (input_control) . . . . . socket ⇝ *handle*
Socket number.

▷ **Tuple** (output_control) . . . . . string ⇝ *string* / real / integer
Received tuple.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect, get_socket_param, set_socket_param

See also

send_tuple, send_image, receive_image, send_region, receive_region, get_next_socket_data_type

Module

Foundation

receive_xld (: XLD : Socket :)

Receive an XLD object over a socket connection.

receive_xld reads an XLD object that was sent over the socket connection determined by Socket by another HALCON process using the operator send_xld. If no XLD object has been sent, the HALCON process calling receive_xld blocks until enough data arrives. For a detailed example, see send_xld.

Parameters

▷ **XLD** (output_object) . . . . . xld(-array) ⇝ *object*
Received XLD object.

▷ **Socket** (input_control) . . . . . socket ⇝ *handle*
Socket number.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect, get_socket_param, set_socket_param

See also

send_xld, send_image, receive_image, send_region, receive_region, send_tuple, receive_tuple, get_next_socket_data_type

Module

Foundation

send_data (: : Socket, Format, Data, To :)

Send arbitrary data to external devices or applications using a generic socket communication.

send_data sends arbitrary data over a socket connection. The sent data is converted to a binary network packet from a value (or a tuple of values) using the parameter Format as specification and is well-suited to communicate with external devices or applications. This operator does not support the standard *'HALCON'* protocol, but is intended for arbitrary data transfer.

The parameter Format specifies how to convert the given tuples to a binary packet. It uses one or multiple qualifier characters each followed by an optional modifier and repeat count. Most qualifiers require a single value in the Data parameter which will be converted.

The following characters are allowed in this format string:

Integer values:

'c': One byte = 8 bit, signed.

'C': Same as 'c' but unsigned.

's': Two bytes = 16 bit, signed.

'S': Same as 's' but unsigned.

'i': Four bytes = 32 bit, signed.

'I': Same as 'i' but unsigned.

'q': Sight bytes = 64 bit, signed (only available on 64bit architectures).

'Q': Same as 'q' but unsigned.

Float values:

'f': Float, 4 bytes = 32 bit.

'd': Double, 8 bytes = 64 bit.

String values:

'A': String (default length 1024 bytes), padded with spaces.

'a': String with arbitrary binary data. When sending data, the whole memory block will be transmitted and the size will be determined automatically. When receiving, the length argument must always be provided explicitly.

'Z': String (default length 1024 bytes), padded with NULL-Bytes and will be NULL terminated when sending.

'z': String with variable length, the length modifier specifies the maximum length (default length 1024 bytes).

Special characters which do not require a value as Data parameter:

'-': A single byte is written as binary NULL or when reading, a NULL-Byte is skipped.

'_': A single byte is written as space (binary 0x20) or when reading, a space byte is skipped.

' ': Ignored, can be used to enhance readability of the format string.

Modifiers which can be used after one of the qualifiers above:

'n': Convert the integer or float value when writing to or when reading from network byte order (big endian) to host byte order.

'N': Convert the integer or float value when writing to or when reading from Intel byte order (little endian) to host byte order.

'0-n': Specify a repeat count for the preceding qualifier, e.g., 'c5' means the same as 'ccccc' (and requires therefore a tuple of 5 values) but 'A10' means a string with a size of 10 bytes (and requires only one value).

The modifiers 'n' and 'N' can also be used as first character in the format string and set the default byte order. 'n', which means network byte order, is the default byte order when nothing else is specified.

For UDP connections the binary data must be transferred in one network packet so that the size of the binary data must not be bigger than one network packet. Usually this means it should be smaller than the MTU (maximum transfer unit) of the interface, which is usually about 1500 bytes, but only 576 bytes are guaranteed (1280 bytes for IPv6).

The parameter `To` should be left empty for socket connections that are already bound (all TCP connections and bound UDP connections), but in case of an unbound UDP connection it must be used to specify the IP address or host name and port number of the communication partner.

Parameters

▷ **`Socket`** (input_control) . . . . . socket ⇝ *handle*
Socket number.

▷ **`Format`** (input_control) . . . . . string ⇝ *string*
Specification how to convert the data.
Default: 'z'

▷ **`Data`** (input_control) . . . . . string(-array) ⇝ *string* / real / integer / handle
Value (or tuple of values) holding the data to send.

▷ **`To`** (input_control) . . . . . string(-array) ⇝ *string* / integer
IP address or hostname and network port of the communication partner.
Default: []
List of values: `To` ∈ {[], ['localhost',3000]}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`open_socket_connect`, `socket_accept_connect`, `get_socket_param`, `set_socket_param`

Possible Successors

`close_socket`

See also

`receive_data`

Module

Foundation

`send_image` (Image : : Socket :)

Send an image over a socket connection.

`send_image` sends an image object over the socket connection determined by `Socket`. The receiving HALCON process must call `receive_image` to read the image from the socket. For a detailed example, see `open_socket_accept`.

Parameters

▷ **Image** (input_object) ..... (multichannel-)image(-array) ⇝ *object :* byte / direction / cyclic / int1 / int2 / uint2 / int4 / int8 / real / complex / vector_field
Image to be sent.

▷ **Socket** (input_control) ..... socket ⇝ *handle*
Socket number.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect

See also

receive_image, send_region, receive_region, send_tuple, receive_tuple, get_next_socket_data_type

Module

Foundation

send_region (Region : : Socket :)

Send regions over a socket connection.

send_region sends a region object over the socket connection determined by Socket. The receiving HALCON process must call receive_region to read the regions from the socket. For a detailed example, see open_socket_accept.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Regions to be sent.

▷ **Socket** (input_control) ..... socket ⇝ *handle*
Socket number.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect

See also

receive_region, send_image, receive_image, send_tuple, receive_tuple, get_next_socket_data_type

Module

Foundation

send_serialized_item (: : Socket, SerializedItemHandle :)

Send a serialized item over a socket connection.

send_serialized_item sends a serialized item over a socket connection determined by Socket (see fwrite_serialized_item for an introduction of the basics of serialization). The receiving HALCON process must call receive_serialized_item to read the serialized item from the socket. The serialized item is returned by the handle SerializedItemHandle. For a detailed example showing the use of a socket connection, see open_socket_accept.

Parameters

▷ **Socket** (input_control) ..... socket ⇝ *handle*
Socket number.

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator send_serialized_item returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_serialized_item_ptr, serialize_matrix, serialize_metrology_model

Possible Successors

receive_serialized_item

Module

Foundation

send_tuple (: : Socket, Tuple :)

Send a tuple over a socket connection.

send_tuple sends a tuple Tuple over the socket connection determined by Socket. The receiving HALCON process must call receive_tuple to read the tuple from the socket. For a detailed example, see open_socket_accept.

Attention

send_tuple does not allow sending tuples containing HALCON handles.

Parameters

▷ **Socket** (input_control) ..... socket ⇝ *handle*
Socket number.

▷ **Tuple** (input_control) ..... string ⇝ *string* / real / integer
Tuple to be sent.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect

See also

receive_tuple, send_image, receive_image, send_region, receive_region, get_next_socket_data_type

Module

Foundation

send_xld (XLD : : Socket :)

Send an XLD object over a socket connection.

send_xld sends an XLD object over the socket connection determined by Socket. The receiving HALCON process must call receive_xld to read the XLD object from the socket.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
XLD object to be sent.

▷ **Socket** (input_control) ..... socket ⇝ *handle*
Socket number.

Example

```
* Process 1
dev_set_colored (12)
open_socket_accept (3000, [], [], AcceptingSocket)
socket_accept_connect (AcceptingSocket, 'true', Socket)
receive_image (Image, Socket)
edges_sub_pix (Image, Edges, 'canny', 1.5, 20, 40)
send_xld (Edges, Socket)
receive_xld (Polygons, Socket)
split_contours_xld (Polygons, Contours, 'polygon', 1, 5)
gen_parallels_xld (Polygons, Parallels, 10, 30, 0.15, 'true')
send_xld (Parallels, Socket)
receive_xld (ModParallels, Socket)
receive_xld (ExtParallels, Socket)
stop ()
close_socket (Socket)
close_socket (AcceptingSocket)

* Process 2
dev_set_colored (12)
open_socket_connect ('localhost', 3000, [], [], Socket)
read_image (Image, 'mreut')
send_image (Image, Socket)
receive_xld (Edges, Socket)
gen_polygons_xld (Edges, Polygons, 'ramer', 2)
send_xld (Polygons, Socket)
split_contours_xld (Polygons, Contours, 'polygon', 1, 5)
receive_xld (Parallels, Socket)
mod_parallels_xld (Parallels, Image, ModParallels, ExtParallels, \
                   0.4, 160, 220, 10)
send_xld (ModParallels, Socket)
send_xld (ExtParallels, Socket)
stop ()
close_socket (Socket)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect

See also

receive_xld, send_image, receive_image, send_region, receive_region, send_tuple, receive_tuple, get_next_socket_data_type

Module

Foundation

set_socket_param (: : Socket, GenParamName, GenParamValue :)

Set a socket parameter.

set_socket_param sets the GenParamName according to GenParamValue for the specified socket connection. Available parameters are *'timeout'*, *'SO_SNDBUF'*, *'SO_RCVBUF'*, *'SO_BROADCAST'*, and *'TCP_NODELAY'*.

The parameter *'timeout'* can be used to set a timeout for this socket. The timeout is given in seconds as a floating point number or as the string *'infinite'*.

The parameters starting with 'SO_' or 'TCP_' set the corresponding socket options. *'SO_SNDBUF'* and *'SO_RCVBUF'* specify the size of the send respectively receive buffer of the operating system for this socket. Please be aware that this does not mean the size of one network packet but the size of the buffers where all packets are temporarily stored by your operating system. *'SO_BROADCAST'* can only be used together with UDP connections and enables broadcasting of network packets. *'TCP_NODELAY'* controls the Nagle algorithm, which optimizes the flow of small TCP packets and can only be used with TCP connections.

Parameters

▷ **Socket** (input_control) . . . . . socket ⇝ *handle*
Socket number.

▷ **GenParamName** (input_control) . . . . . string(-array) ⇝ *string*
Name of the socket parameter.
List of values: GenParamName ∈ {'timeout', 'SO_SNDBUF', 'SO_RCVBUF', 'SO_BROADCAST', 'TCP_NODELAY'}

▷ **GenParamValue** (input_control) . . . . . string(-array) ⇝ *string* / real / integer
Value of the socket parameter.
Default: 'on'
Suggested values: GenParamValue ∈ {'on', 'off', 0, 1, 3.0, 'infinite', 530, 1460}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_connect, socket_accept_connect

Possible Successors

send_data, receive_data

See also

get_socket_param

Module

Foundation

socket_accept_connect (: : AcceptingSocket, Wait : Socket)

Accept a connection request on a listening socket of the protocol type 'HALCON' or 'TCP'/'TCP4'/'TCP6'.

socket_accept_connect accepts an incoming connection request, generated by open_socket_connect in another HALCON process or from an external application, on the

listening socket AcceptingSocket. The listening socket must have been created earlier with open_socket_accept. Its timeout determines the timeout of the socket_accept_connect call. If Wait='true', socket_accept_connect waits until a connection request from another HALCON process arrives. If Wait='false', socket_accept_connect returns with the error 5 (H_MSG_FAIL), if currently there are no connection requests from other HALCON processes. The value 'auto' for Wait automatically waits if the timeout of the accepting socket is not equal 0. The result of socket_accept_connect is another socket Socket, which is used for a two-way communication with another process. After this connection has been established, data can be exchanged between the two processes by calling the appropriate send or receive operators. For a detailed example, see open_socket_accept.

For the data transfer with generic sockets only the operators send_data and receive_data are available.

Parameters

▷ **AcceptingSocket** (input_control) ..... socket ⇝ *handle*
Socket number of the accepting socket.

▷ **Wait** (input_control) ..... string ⇝ *string*
Should the operator wait until a connection request arrives?
Default: 'auto'
List of values: Wait ∈ {'auto', 'true', 'false'}

▷ **Socket** (output_control) ..... socket ⇝ *handle*
Socket number.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

open_socket_accept

Possible Successors

send_image, receive_image, send_region, receive_region, send_tuple, receive_tuple, send_data, receive_data

See also

open_socket_connect, close_socket, get_socket_param, set_socket_param

Module

Foundation

Chapter 26

Tools

26.1 Background Estimator

close_bg_esti (: : BgEstiHandle :)

Delete the background estimation data set.

close_bg_esti deletes the background estimation data set and releases all used memory.

Parameters

▷ **BgEstiHandle** (input_control) ..... bg_estimation ⇝ *handle*
ID of the BgEsti data set.

Example

```
* Read image for initialization:
read_image (InitImage, 'xing/init')
* Initialize BgEsti dataset with
* fixed gains and threshold adaption:
create_bg_esti(InitImage,0.7,0.7,'fixed',0.002,0.02, \
               'on',7.0,10,3.25,15.0,BgEstiHandle)
* Read the next image in sequence:
read_image(Image0, 'xing/xing001')
* Estimate the background:
run_bg_esti(Image0,ForegroundRegion1,BgEstiHandle)
* Display the foreground region:
dev_display (ForegroundRegion1)
* Read the next image in sequence:
read_image(Image1, 'xing/xing002')
* Estimate the background:
run_bg_esti(Image1,ForegroundRegion2,BgEstiHandle)
* Display the foreground region:
dev_display (ForegroundRegion2)
* etc.
* - End of background estimation -
* Close the dataset:
close_bg_esti(BgEstiHandle)
```

Result

close_bg_esti returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- BgEstiHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

run_bg_esti

See also

create_bg_esti

Module

Foundation

create_bg_esti (InitializeImage : : Syspar1, Syspar2, GainMode, Gain1, Gain2, AdaptMode, MinDiff, StatNum, ConfidenceC, TimeC : BgEstiHandle)

Generate and initialize a data set for the background estimation.

create_bg_esti creates a new data set for the background estimation and initializes it with the appropriate parameters. The estimated background image is part of this data set. The newly created set automatically becomes the current set.

InitializeImage is used as an initial prediction for the background image. For a good prediction an image of the observed scene without moving objects should be passed in InitializeImage. That way the foreground adaptation rate can be held low. If there is no empty scene image available, a homogeneous gray image can be used instead. In that case the adaptation rate for the foreground image must be raised, because initially most of the image will be detected as foreground. The initialization image must to be of type byte or real. Because of processing single-channel images, data sets must be created for every channel. Size and region of InitializeImage determines size and region for all background estimations (run_bg_esti) that are performed with this data set.

Syspar1 and Syspar2 are the parameters of the Kalman system matrix. The system matrix describes the system of the gray value changes according to Kalman filter theory. The background estimator implements a different system for each pixel.

GainMode defines whether a fixed Kalman gain should be used for the estimation or whether the gain should adapt itself depending on the difference between estimation and actual value. If GainMode is set to *'fixed'*, then Gain1 is used as Kalman gain for pixels predicted as foreground and Gain2 as gain for pixels predicted as background. Gain1 should be smaller than Gain2, because adaptation of the foreground should be slower than adaptation of the background. Both Gain1 and Gain2 should be smaller than *1.0*.

If GainMode is set to *'frame'*, then tables for foreground and background estimation are computed containing Kalman gains for all the 256 possible gray value changes. Gain1 and Gain2 then denote the number of frames necessary to adapt the difference between estimated value and actual value. So with a fixed time for adaptation (i.e. number of frames) the needed Kalman gain grows with the gray value difference. Gain1 should therefore be larger than Gain2. Different gains for different gray value differences are useful if the background estimator is used for generating an 'empty' scene assuming that there are always moving objects in the observed area. In that case the adaptation time for foreground adaptation (Gain1) must not be too big. Gain1 and Gain2 should be bigger than *1.0*.

AdaptMode denotes, whether the foreground/background decision threshold applied to the gray value difference between estimation and actual value is fixed or whether it adapts itself depending on the gray value deviation of the background pixels.

If AdaptMode is set to *'off'*, the parameter MinDiff denotes a fixed threshold. The parameters StatNum, ConfidenceC and TimeC are meaningless in this case.

If AdaptMode is set to *'on'*, then MinDiff is interpreted as a base threshold. For each pixel an offset is added to this threshold depending on the statistical evaluation of the pixel value over time. StatNum holds the number

of data sets (past frames) that are used for computing the gray value variance (FIR-Filter). ConfidenceC is used to determine the confidence interval.

The confidence interval determines the values of the background statistics if background pixels are hidden by a foreground object and thus are detected as foreground. According to the student t-distribution the confidence constant is *4.30* (*3.25*, *2.82*, *2.26*) for a confidence interval of *99,8%* (*99,0%*, *98,0%*, *95,0%*). TimeC holds a time constant for the exp-function that raises the threshold in case of a foreground estimation of the pixel. That means, the threshold is raised in regions where movement is detected in the foreground. That way larger changes in illumination are tolerated if the background becomes visible again. The main reason for increasing this tolerance is the impossibility for a prediction of illumination changes while the background is hidden. Therefore no adaptation of the estimated background image is possible.

Attention

If GainMode was set to *'frame'*, the run-time can be extremely long for large values of Gain1 or Gain2, because the values for the gains' table are determined by a simple binary search.

Parameters

▷ **InitializeImage** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / real
initialization image.

▷ **Syspar1** (input_control) . . . . . real ⇝ *real*
1. system matrix parameter.
Default: 0.7
Suggested values: Syspar1 ∈ {0.65, 0.7, 0.75}
Value range: 0.05 ≤ Syspar1 ≤ 1.0
Recommended increment: 0.05

▷ **Syspar2** (input_control) . . . . . real ⇝ *real*
2. system matrix parameter.
Default: 0.7
Suggested values: Syspar2 ∈ {0.65, 0.7, 0.75}
Value range: 0.05 ≤ Syspar2 ≤ 1.0
Recommended increment: 0.05

▷ **GainMode** (input_control) . . . . . string ⇝ *string*
Gain type.
Default: 'fixed'
List of values: GainMode ∈ {'fixed', 'frame'}

▷ **Gain1** (input_control) . . . . . real ⇝ *real*
Kalman gain / foreground adaptation time.
Default: 0.002
Suggested values: Gain1 ∈ {10.0, 20.0, 50.0, 0.1, 0.05, 0.01, 0.005, 0.001}
Restriction: 0.0 <= Gain1

▷ **Gain2** (input_control) . . . . . real ⇝ *real*
Kalman gain / background adaptation time.
Default: 0.02
Suggested values: Gain2 ∈ {2.0, 4.0, 8.0, 0.5, 0.1, 0.05, 0.01}
Restriction: 0.0 <= Gain2

▷ **AdaptMode** (input_control) . . . . . string ⇝ *string*
Threshold adaptation.
Default: 'on'
List of values: AdaptMode ∈ {'on', 'off'}

▷ **MinDiff** (input_control) . . . . . real ⇝ *real*
Foreground/background threshold.
Default: 7.0
Suggested values: MinDiff ∈ {3.0, 5.0, 7.0, 9.0, 11.0}
Recommended increment: 0.2

▷ **StatNum** (input_control) . . . . . integer ⇝ *integer*
Number of statistic data sets.
Default: 10
Suggested values: StatNum ∈ {5, 10, 20, 30}
Value range: 1 ≤ StatNum
Recommended increment: 5

▷ **ConfidenceC** (input_control) . . . . . real ⇝ *real*
Confidence constant.
Default: 3.25
Suggested values: ConfidenceC ∈ {4.30, 3.25, 2.82, 2.62}
Recommended increment: 0.01
Restriction: 0.0 < ConfidenceC

▷ **TimeC** (input_control) . . . . . real ⇝ *real*
Constant for decay time.
Default: 15.0
Suggested values: TimeC ∈ {10.0, 15.0, 20.0}
Recommended increment: 5.0
Restriction: 0.0 < TimeC

▷ **BgEstiHandle** (output_control) . . . . . bg_estimation ⇝ *handle*
ID of the BgEsti data set.

Example

```
* read Init-Image:
read_image (InitImage, 'xing/init')
* initialize 1. BgEsti-Dataset with
* fixed gains and threshold adaption:
create_bg_esti(InitImage,0.7,0.7,'fixed',0.002,0.02, \
               'on',7.0,10,3.25,15.0,BgEstiHandle1)
* initialize 2. BgEsti-Dataset with
* frame orientated gains and fixed threshold
create_bg_esti(InitImage,0.7,0.7,'frame',30.0,4.0, \
               'off',9.0,10,3.25,15.0,BgEstiHandle2)
```

Result

create_bg_esti returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

run_bg_esti

See also

set_bg_esti_params, close_bg_esti

Module

Foundation

get_bg_esti_params (: : BgEstiHandle : Syspar1, Syspar2, GainMode, Gain1, Gain2, AdaptMode, MinDiff, StatNum, ConfidenceC, TimeC)

Return the parameters of the data set.

get_bg_esti_params returns the parameters of the data set. The returned parameters are the same as in create_bg_esti and set_bg_esti_params (see these for an explanation).

Parameters

- ▷ **BgEstiHandle** (input_control) ..... bg_estimation ⇝ *handle*
  ID of the BgEsti data set.
- ▷ **Syspar1** (output_control) ..... real ⇝ *real*
  1. system matrix parameter.
- ▷ **Syspar2** (output_control) ..... real ⇝ *real*
  2. system matrix parameter.
- ▷ **GainMode** (output_control) ..... string ⇝ *string*
  Gain type.
- ▷ **Gain1** (output_control) ..... real ⇝ *real*
  Kalman gain / foreground adaptation time.
- ▷ **Gain2** (output_control) ..... real ⇝ *real*
  Kalman gain / background adaptation time.
- ▷ **AdaptMode** (output_control) ..... string ⇝ *string*
  Threshold adaptation.
- ▷ **MinDiff** (output_control) ..... real ⇝ *real*
  Foreground / background threshold.
- ▷ **StatNum** (output_control) ..... integer ⇝ *integer*
  Number of statistic data sets.
- ▷ **ConfidenceC** (output_control) ..... real ⇝ *real*
  Confidence constant.
- ▷ **TimeC** (output_control) ..... real ⇝ *real*
  Constant for decay time.

Example

```
* Read image for initialization:
read_image(InitImage,'xing/init')
* Initialize BgEsti dataset with
* fixed gains and threshold adaption:
create_bg_esti(InitImage,0.7,0.7,'fixed',0.002,0.02, \
               'on',7.0,10,3.25,15.0,BgEstiHandle)
* Read the next image in sequence:
read_image(Image0,'xing/xing000')
* Estimate the background:
run_bg_esti(Image0,ForegroundRegion1,BgEstiHandle)
* Display the foreground region:
dev_display (ForegroundRegion1)
* Read the next image in sequence:
read_image(Image1,'xing/xing001')
* Estimate the background:
run_bg_esti(Image1,ForegroundRegion2,BgEstiHandle)
* Display the foreground region:
dev_display (ForegroundRegion2)
* etc.
* Change only the gain parameter in dataset:
get_bg_esti_params(BgEstiHandle, Syspar1, Syspar2, \
                   GainMode, Gain1, Gain2, AdaptMode,\
                   MinDiff, StatNum, ConfidenceC, TimeC)
set_bg_esti_params(BgEstiHandle, Syspar1, Syspar2, \
                   GainMode, 0.004, 0.08, AdaptMode,\
                   MinDiff, StatNum, ConfidenceC, TimeC)
* Read the next image in sequence:
read_image(Image3,'xing/xing003')
* Estimate the background:
run_bg_esti(Image3,ForegroundRegion3,BgEstiHandle)
* Display the foreground region:
```

```
dev_display(ForegroundRegion3)
* etc
```

Result

`get_bg_esti_params` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_bg_esti`

Possible Successors

`run_bg_esti`

See also

`set_bg_esti_params`

Module

Foundation

give_bg_esti (: BackgroundImage : BgEstiHandle :)

Return the estimated background image.

`give_bg_esti` returns the estimated background image of the current BgEsti data set. The background image has the same type and size as the initialization image passed in `create_bg_esti`.

Parameters

▷ **BackgroundImage** (output_object) ..... image ⇝ *object :* byte / real
Estimated background image of the current data set.

▷ **BgEstiHandle** (input_control) ..... bg_estimation ⇝ *handle*
ID of the BgEsti data set.

Example

```
* Read image for initialization:
read_image(InitImage,'xing/init')
* Initialize BgEsti dataset with
* fixed gains and threshold adaption:
create_bg_esti(InitImage,0.7,0.7,'fixed',0.002,0.02, \
               'on',7.0,10,3.25,15.0,BgEstiHandle)
* Read the next image in sequence:
read_image(Image0,'xing/xing000')
* Estimate the background:
run_bg_esti(Image0,ForegroundRegion1,BgEstiHandle)
* Give the background image from the aktive dataset:
give_bg_esti(BgImage,BgEstiHandle)
* Display the background image:
dev_display(BgImage)
```

Result

`give_bg_esti` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

run_bg_esti

Possible Successors

run_bg_esti, create_bg_esti, update_bg_esti

See also

run_bg_esti, update_bg_esti, create_bg_esti

Module

Foundation

run_bg_esti (PresentImage : ForegroundRegion : BgEstiHandle :)

Estimate the background and return the foreground region.

run_bg_esti adapts the background image stored in the BgEsti data set using a Kalman filter on each pixel and returns a region of the foreground (detected moving objects).

For every pixel an estimation of its gray value is computed using the values of the current data set and its stored background image and the current image (PresentImage). By comparison to the threshold (fixed or adaptive, see create_bg_esti) the pixels are classified as either foreground or background.

The background estimation processes only single-channel images. Therefore the background has to be adapted separately for every channel.

The background estimation should be used on half- or even quarter-sized images. For this, the input images (and the initialization image!) has to be reduced using zoom_image_factor. The advantage is a shorter run-time on one hand and a low-band filtering on the other. The filtering eliminates high frequency noise and results in a more reliable estimation. As a result the threshold (see create_bg_esti) can be lowered. The foreground region returned by run_bg_esti then has to be enlarged again for further processing.

Attention

The passed image (PresentImage) must have the same type and size as the background image of the current data set (initialized with create_bg_esti).

Parameters

- ▷ **PresentImage** (input_object) ..... singlechannelimage ⇝ *object :* byte / real
  Current image.
- ▷ **ForegroundRegion** (output_object) ..... region ⇝ *object*
  Region of the detected foreground.
- ▷ **BgEstiHandle** (input_control) ..... bg_estimation ⇝ *handle*
  ID of the BgEsti data set.

Example

```
* Read image for initialization:
read_image(InitImage,'xing/init')
* Initialize BgEsti dataset with
* fixed gains and threshold adaption:
create_bg_esti(InitImage,0.7,0.7,'fixed',0.002,0.02, \
               'on',7.0,10,3.25,15.0,BgEstiHandle)
* Read the next image in sequence:
read_image(Image0,'xing/xing000')
* Estimate the background:
run_bg_esti(Image0,ForegroundRegion1,BgEstiHandle)
* Display the foreground region:
dev_display (ForegroundRegion1)
* Read the next image in sequence:
read_image(Image1,'xing/xing001')
```

```
* Estimate the background:
run_bg_esti(Image1,ForegroundRegion2,BgEstiHandle)
* Display the foreground region:
dev_display (ForegroundRegion2)
* etc.
```

Result

`run_bg_esti` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_bg_esti`, `update_bg_esti`

Possible Successors

`give_bg_esti`, `update_bg_esti`

See also

`set_bg_esti_params`, `create_bg_esti`, `update_bg_esti`, `give_bg_esti`

Module

Foundation

set_bg_esti_params (: : BgEstiHandle, Syspar1, Syspar2, GainMode, Gain1, Gain2, AdaptMode, MinDiff, StatNum, ConfidenceC, TimeC :)

Change the parameters of the data set.

`set_bg_esti_params` is used to change the parameters of the data set. The parameters passed by `set_bg_esti_params` are the same as in `create_bg_esti` (see there for an explanation).

The image format cannot be changed! To do this, a new data set with an initialization image of the appropriate format has to be created.

To exchange the background image completely, use `update_bg_esti`. The current image then has to be passed for both the input image and the update region.

Attention

If `GainMode` was set to *'frame'*, the run-time can be extremely long for large values of `Gain1` or `Gain2`, because the values for the gains' table are determined by a simple binary search.

Parameters

▷ **BgEstiHandle** (input_control) . . . . . bg_estimation ⇝ *handle*
ID of the BgEsti data set.

▷ **Syspar1** (input_control) . . . . . real ⇝ *real*
1. system matrix parameter.
Default: 0.7
Suggested values: `Syspar1` ∈ {0.65, 0.7, 0.75}
Value range: 0.05 ≤ `Syspar1` ≤ 1.0
Recommended increment: 0.05

▷ **Syspar2** (input_control) . . . . . real ⇝ *real*
2. system matrix parameter.
Default: 0.7
Suggested values: `Syspar2` ∈ {0.65, 0.7, 0.75}
Value range: 0.05 ≤ `Syspar2` ≤ 1.0
Recommended increment: 0.05

▷ **GainMode** (input_control) . . . . . string ⇝ *string*
Gain type.
Default: 'fixed'
List of values: GainMode ∈ {'fixed', 'frame'}

▷ **Gain1** (input_control) . . . . . real ⇝ *real*
Kalman gain / foreground adaptation time.
Default: 0.002
Suggested values: Gain1 ∈ {10.0, 20.0, 50.0, 0.1, 0.05, 0.01, 0.005, 0.001}
Restriction: 0.0 <= Gain1

▷ **Gain2** (input_control) . . . . . real ⇝ *real*
Kalman gain / background adaptation time.
Default: 0.02
Suggested values: Gain2 ∈ {2.0, 4.0, 8.0, 0.5, 0.1, 0.05, 0.01}
Restriction: 0.0 <= Gain2

▷ **AdaptMode** (input_control) . . . . . string ⇝ *string*
Threshold adaptation.
Default: 'on'
List of values: AdaptMode ∈ {'on', 'off'}

▷ **MinDiff** (input_control) . . . . . real ⇝ *real*
Foreground/background threshold.
Default: 7.0
Suggested values: MinDiff ∈ {3.0, 5.0, 7.0, 9.0, 11.0}
Recommended increment: 0.2

▷ **StatNum** (input_control) . . . . . integer ⇝ *integer*
Number of statistic data sets.
Default: 10
Suggested values: StatNum ∈ {5, 10, 20, 30}
Value range: 1 ≤ StatNum
Recommended increment: 5

▷ **ConfidenceC** (input_control) . . . . . real ⇝ *real*
Confidence constant.
Default: 3.25
Suggested values: ConfidenceC ∈ {4.30, 3.25, 2.82, 2.62}
Recommended increment: 0.01
Restriction: 0.0 < ConfidenceC

▷ **TimeC** (input_control) . . . . . real ⇝ *real*
Constant for decay time.
Default: 15.0
Suggested values: TimeC ∈ {10.0, 15.0, 20.0}
Recommended increment: 5.0
Restriction: 0.0 < TimeC

Example

```
* Read image for initialization:
read_image(InitImage,'xing/init')
* Initialize BgEsti dataset with
* fixed gains and threshold adaption:
create_bg_esti(InitImage,0.7,0.7,'fixed',0.002,0.02, \
               'on',7.0,10,3.25,15.0,BgEstiHandle)
* Read the next image in sequence:
read_image(Image0,'xing/xing000')
* Estimate the background:
run_bg_esti(Image0,ForegroundRegion1,BgEstiHandle)
* Display the foreground region:
dev_display (ForegroundRegion1)
* Read the next image in sequence:
read_image(Image1,'xing/xing001')
* Estimate the background:
```

```
run_bg_esti(Image1,ForegroundRegion2,BgEstiHandle)
* Display the foreground region:
dev_display (ForegroundRegion2)
* etc.
* Change parameter in dataset:
set_bg_esti_params(BgEstiHandle,0.7,0.7,'fixed', \
                   0.004,0.08,'on',9.0,10,3.25,20.0)
* Read the next image in sequence:
read_image(Image2,'xing/xing002')
* Estimate the background:
run_bg_esti(Image2,ForegroundRegion3,BgEstiHandle)
* Display the foreground region:
dev_display(ForegroundRegion3)
* etc.
```

Result

`set_bg_esti_params` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_bg_esti`

Possible Successors

`run_bg_esti`

See also

`update_bg_esti`

Module

Foundation

update_bg_esti (PresentImage, UpDateRegion : : BgEstiHandle :)

Change the estimated background image.

`update_bg_esti` overwrites the image stored in the current BgEsti data set with the gray values of `PresentImage` within the bounds of `UpDateRegion`. This can be used for a "hard" adaptation: Image regions with a sudden change in (known) background can be adapted very fast this way.

Attention

The passed image (`PresentImage`) must have the same type and size as the background image of the current data set (initialized with `create_bg_esti`).

Parameters

▷ **PresentImage** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / real
Current image.

▷ **UpDateRegion** (input_object) . . . . . region ⇝ *object*
Region describing areas to change.

▷ **BgEstiHandle** (input_control) . . . . . bg_estimation ⇝ *handle*
ID of the BgEsti data set.

Example

```
* read Init-Image:
read_image(InitImage,'xing/init')
```

```
* initialize BgEsti-Dataset with
* fixed gains and threshold adaption
create_bg_esti(InitImage,0.7,0.7,'fixed',0.002,0.02, \
               'on',7,10,3.25,15.0,BgEstiHandle)
* read the next image in sequence:
read_image(Image0,'xing/xing000')
* estimate the Background:
run_bg_esti(Image0,Region1,BgEstiHandle)
* use the Region and the information of a knowledge base
* to calculate the UpDateRegion
update_bg_esti(Image0,UpdateRegion,BgEstiHandle)
* then read the next image in sequence:
read_image(Image1,'xing/xing001')
* estimate the Background:
run_bg_esti(Image1,Region2,BgEstiHandle)
* etc.
```

Result

`update_bg_esti` returns `2` (`H_MSG_TRUE`) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`run_bg_esti`

Possible Successors

`run_bg_esti`

See also

`run_bg_esti`, `give_bg_esti`

Module

Foundation

26.2 Function

`abs_funct_1d` (: : `Function` : `FunctionAbsolute`)

Absolute value of the y values.

`abs_funct_1d` calculates the absolute values of all y values of `Function`.

Parameters

▷ **`Function`** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function.

▷ **`FunctionAbsolute`** (output_control) . . . . . function_1d ⇝ *real* / integer
Function with the absolute values of the y values.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array

Module

Foundation

compose_funct_1d (: : Function1, Function2, Border : ComposedFunction)

Compose two functions.

compose_funct_1d composes two functions, i.e., calculates

$$\texttt{ComposedFunction}(x) = \texttt{Function2}(\texttt{Function1}(x)) \ .$$

ComposedFunction has the same domain (x-range) as Function1. If the range (y value range) of Function1 is larger than the domain of Function2, the parameter Border determines the border treatment of Function2. For Border='*zero*' values outside the domain of Function2 are set to 0, for Border='*constant*' they are set to the corresponding value at the border, for Border='*mirror*' they are mirrored at the border, and for Border='*cyclic*' they are continued cyclically. To obtain y values, Function2 is interpolated linearly.

Parameters

- ▷ **Function1** (input_control) ..... function_1d ⇝ *real* / integer
  Input function 1.
- ▷ **Function2** (input_control) ..... function_1d ⇝ *real* / integer
  Input function 2.
- ▷ **Border** (input_control) ..... string ⇝ *string*
  Border treatment for the input functions.
  Default: 'constant'
  List of values: Border ∈ {'zero', 'constant', 'mirror', 'cyclic'}
- ▷ **ComposedFunction** (output_control) ..... function_1d ⇝ *real* / integer
  Composed function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array

Module

Foundation

create_funct_1d_array (: : YValues : Function)

Create a function from a sequence of y values.

create_funct_1d_array creates a one-dimensional function from a set of y values YValues. The resulting function can then be processed and analyzed with the operators for 1d functions. YValues is interpreted as follows: the first value of YValues is the function value at zero, the second value is the function value at one, etc. Thus, the values define a function at equidistant x values (with distance $\Delta x = 1$), starting at $x_1 = 0$.

The created function is composed like this:

Function $= [0, \Delta x, x_1, y_1, y_2, ..., y_n]$

where

Function[0] = 0, denotes the function type as an equidistant function,

Δx is the equidistance of the x values,

x_1 is the starting x value (is always 0), and

y_i are the y values passed in YValues with $i \in [1, n]$.

Alternatively, the operator create_funct_1d_pairs can be used to create a function. create_funct_1d_pairs also allows to define a function with non-equidistant x values by specifying them explicitly. Thus to get the same definition as with create_funct_1d_array, one would pass a tuple of x values to create_funct_1d_pairs that has the same length as YValues and contains values starting at 0 and increasing by 1 in each position. Note, however, that create_funct_1d_pairs leads to a different internal representation of the function which needs more storage (because all (x,y) pairs are stored) and sometimes cannot be processed as efficiently as functions created by create_funct_1d_array.

Parameters

▷ **YValues** (input_control) ..... number(-array) ⇝ *real* / integer
X value for function points.

▷ **Function** (output_control) ..... function_1d ⇝ *real* / integer
Created function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

write_funct_1d, y_range_funct_1d, get_pair_funct_1d, transform_funct_1d

Alternatives

create_funct_1d_pairs, read_funct_1d

See also

funct_1d_to_pairs

Module

Foundation

create_funct_1d_pairs (: : XValues, YValues : Function)

Create a function from a set of (x,y) pairs.

create_funct_1d_pairs creates a one-dimensional function from a set of pairs of (x,y) values. The XValues of the functions have to be passed in ascending order. The resulting function can then be processed and analyzed with the operators for 1d functions.

The created function is composed like this:

Function $= [1, x_1, y_1, x_2, y_2, ..., x_n, y_n]$

where

Function[1] = 1 denotes the function type as a one-dimensional function from a set of pairs and

$[x_i, y_i]$ are pairs of (x,y) values passed in XValues and YValues with $i \in [1, n]$.

Alternatively, functions can be created with the operator create_funct_1d_array. In contrast to this operator, x values with arbitrary positions can be specified with create_funct_1d_pairs. Hence, it is the more general operator. It should be noted, however, that because of this generality the processing of a function created with create_funct_1d_pairs cannot be carried out as efficiently as for equidistant functions. In particular, not all operators accept such functions. If necessary, a function can be transformed into an equidistant function with the operator sample_funct_1d.

Attention

create_funct_1d_pairs examines whether the x values of type 'double' are ascending. Some other operators apply the same check but use the type 'float' instead of 'double'. If such an operator is called as successor, it

might happen that two consecutive x values seem to be equal for the 'float' variant, although they were ascending in the 'double' variant. If this happens, an error is thrown.

Parameters

▷ **XValues** (input_control) . . . . . number(-array) ⇝ *real* / integer
X value for function points.

▷ **YValues** (input_control) . . . . . number(-array) ⇝ *real* / integer
Y value for function points.

▷ **Function** (output_control) . . . . . function_1d ⇝ *real* / integer
Created function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`write_funct_1d`, `y_range_funct_1d`, `get_pair_funct_1d`

Alternatives

`create_funct_1d_array`, `read_funct_1d`

See also

`funct_1d_to_pairs`

Module

Foundation

derive_funct_1d (: : Function, Mode : Derivative)

Calculate the derivatives of a function.

`derivate_funct_1d` calculates the derivatives of the function `Function` up to the second degree. It uses a finite difference approximation of order $O(h^2)$. The derivative is also a function with the same sampling points as `Function`. With the parameter `Mode`, *'first'* and *'second'* derivatives can be selected.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Type of derivative
Default: 'first'
List of values: `Mode` ∈ {'first', 'second'}

▷ **Derivative** (output_control) . . . . . function_1d ⇝ *real* / integer
Derivative of the input function

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_funct_1d_pairs`, `create_funct_1d_array`, `smooth_funct_1d_gauss`, `smooth_funct_1d_mean`

Module

Foundation

funct_1d_to_pairs (: : Function : XValues, YValues)

Access to the x/y values of a function.

`funct_1d_to_pairs` splits the input function `Function` into tuples for the x and y values.

Parameters

▷ **Function** (input_control) ..... function_1d ⇝ *real* / integer
Input function.

▷ **XValues** (output_control) ..... number-array ⇝ *real* / integer
X values of the function.

▷ **YValues** (output_control) ..... number-array ⇝ *real* / integer
Y values of the function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_funct_1d_pairs`, `create_funct_1d_array`

Module

Foundation

get_pair_funct_1d (: : Function, Index : X, Y)

Access a function value using the index of the control points.

`get_pair_funct_1d` accesses a function value of `Function`. This is done by specifying the index of one or more control points of the function.

Parameters

▷ **Function** (input_control) ..... function_1d ⇝ *real* / integer
Input function.

▷ **Index** (input_control) ..... integer(-array) ⇝ *integer*
Index of the control points.

▷ **X** (output_control) ..... number(-array) ⇝ *real*
X value at the given control points.

▷ **Y** (output_control) ..... number(-array) ⇝ *real*
Y value at the given control points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_funct_1d_pairs`, `create_funct_1d_array`

Module

Foundation

get_y_value_funct_1d (: : Function, X, Border : Y)

Return the value of a function at an arbitrary position.

get_y_value_funct_1d returns the y value of the function Function at the x coordinates specified by X. To obtain the y values, the input function is interpolated linearly. The parameter Border determines the values of the function Function outside of its domain. For Border='zero' these values are set to 0, for Border='constant' they are set to the corresponding value at the border, for Border='mirror' they are mirrored at the border, for Border='cyclic' they are continued cyclically, and for Border='error' an exception is raised.

Parameters

- ▷ **Function** (input_control) . . . function_1d ⇝ *real* / integer
  Input function.
- ▷ **X** (input_control) . . . number(-array) ⇝ *real*
  X coordinate at which the function should be evaluated.
- ▷ **Border** (input_control) . . . string ⇝ *string*
  Border treatment for the input function.
  Default: 'constant'
  List of values: Border ∈ {'zero', 'constant', 'mirror', 'cyclic', 'error'}
- ▷ **Y** (output_control) . . . number(-array) ⇝ *real*
  Y value at the given x value.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array

Module

Foundation

integrate_funct_1d (: : Function : Positive, Negative)

Compute the positive and negative areas of a function.

integrate_funct_1d integrates the function Function (see create_funct_1d_array and create_funct_1d_pairs) and returns the integral of the positive and negative parts of the function in Positive and Negative, respectively. Hence, the integral of the function is the difference Positive - Negative. The integration is done on the interval on which the function is defined. For the integration, the function is interpolated linearly.

Parameters

- ▷ **Function** (input_control) . . . function_1d ⇝ *real*
  Input function.
- ▷ **Positive** (output_control) . . . number ⇝ *real*
  Area under the positive part of the function.
- ▷ **Negative** (output_control) . . . number-array ⇝ *real*
  Area under the negative part of the function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array

See also

create_funct_1d_array, create_funct_1d_pairs

Module

Foundation

invert_funct_1d (: : Function : InverseFunction)

Calculate the inverse of a function.

invert_funct_1d calculates the inverse function of the input function Function and returns it in InverseFunction. The function Function must be monotonic. If this is not the case an error message is returned.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function.

▷ **InverseFunction** (output_control) . . . . . function_1d ⇝ *real* / integer
Inverse of the input function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array

Module

Foundation

local_min_max_funct_1d (: : Function, Mode, Interpolation : Min, Max)

Calculate the local minimum and maximum points of a function.

local_min_max_funct_1d searches for the local minima Min and maxima Max of the function Function.

Since the function values are only known at discrete sampling points, the function can interpolated by parabolas between these points. Setting the parameter Interpolation to *'true'*, enables this feature. If Interpolation is *'false'*, extrema are always sampling points.

If Mode is set to *'strict_min_max'*, extrema are only calculated close to points with a function value that is strictly smaller or strictly greater than the values of its direct neighbors.

If Mode is set to *'plateaus_center'*, areas with a function value that is constant throughout several sampling points are also considered. If such an area is identified as being a flat extremum, its center coordinate is returned.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Handling of plateaus
Default: 'strict_min_max'
List of values: Mode ∈ {'strict_min_max', 'plateaus_center'}

▷ **Interpolation** (input_control) . . . . . string ⇝ *string*
Interpolation of the input function
Default: 'true'
List of values: Interpolation ∈ {'true', 'false'}

▷ **Min** (output_control) . . . . . real-array ⇝ *real*
Minimum points of the input function

▷ **Max** (output_control) . . . . . real-array ⇝ *real*
Maximum points of the input function

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array, smooth_funct_1d_gauss, smooth_funct_1d_mean

Module

Foundation

match_funct_1d_trans (: : Function1, Function2, Border, ParamsConst, UseParams : Params, ChiSquare, Covar)

Calculate transformation parameters between two functions.

match_funct_1d_trans calculates the transformation parameters between two functions given as the tuples Function1 and Function2 (see create_funct_1d_array and create_funct_1d_pairs). The following model is used for the transformation between the two functions:

$$y_1(x) = a_1 y_2(a_3 x + a_4) + a_2 \ .$$

The transformation parameters are determined by a least-squares minimization of the following function:

$$\sum_{i=0}^{n-1} \left(y_1(x_i) - a_1 y_2(a_3 x_i + a_4) + a_2\right)^2 \ .$$

The values of the function y_2 are obtained by linear interpolation. The parameter Border determines the values of the function Function2 outside of its domain. For Border='*zero*' these values are set to 0, for Border='*constant*' they are set to the corresponding value at the border, for Border='*mirror*' they are mirrored at the border, and for Border='*cyclic*' they are continued cyclically. The calculated transformation parameters are returned as a 4-tuple $[a_1, a_2, a_3, a_4]$ in Params. If some of the parameter values are known, the respective parameters can be excluded from the least-squares adjustment by setting the corresponding value in the tuple UseParams to the value '*false*'. In this case, the tuple ParamsConst must contain the known value of the respective parameter. If a parameter is used for the adjustment (UseParams = '*true*'), the corresponding parameter in ParamsConst is ignored. On output, match_funct_1d_trans additionally returns the sum of the squared errors ChiSquare of the resulting function, i.e., the function obtained by transforming the input function with the transformation parameters, as well as the covariance matrix Covar of the transformation parameters Params. These parameters can be used to decide whether a successful matching of the functions was possible.

Note that in case that there is no unique solution for the transformation parameters, match_funct_1d_trans either returns one selected single solution or returns the error 9205 (Matrix is singular).

Parameters

▷ **Function1** (input_control) . . . . . function_1d ⇝ *real* / integer
Function 1.

▷ **Function2** (input_control) . . . . . function_1d ⇝ *real* / integer
Function 2.

▷ **Border** (input_control) . . . . . string ⇝ *string*
Border treatment for function 2.
Default: 'constant'
List of values: Border ∈ {'zero', 'constant', 'mirror', 'cyclic'}

▷ **ParamsConst** (input_control) . . . . . number-array ⇝ *real*
Values of the parameters to remain constant.
Number of elements: 4
Default: [1.0,0.0,1.0,0.0]

▷ **UseParams** (input_control) . . . . . string-array ⇝ *string*
Should a parameter be adapted for it?
Number of elements: 4
Default: ['true','true','true','true']
List of values: `UseParams` ∈ {'true', 'false'}

▷ **Params** (output_control) . . . . . number-array ⇝ *real*
Transformation parameters between the functions.
Number of elements: 4

▷ **ChiSquare** (output_control) . . . . . number ⇝ *real*
Quadratic error of the output function.

▷ **Covar** (output_control) . . . . . number-array ⇝ *real*
Covariance Matrix of the transformation parameters.
Number of elements: 16

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_funct_1d_array`, `create_funct_1d_pairs`

See also

`gray_projections`

Module

Foundation

negate_funct_1d (: : Function : FunctionInverted)

Negation of the y values.

`negate_funct_1d` negates all y values of `Function`.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function.

▷ **FunctionInverted** (output_control) . . . . . function_1d ⇝ *real* / integer
Function with the negated y values.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_funct_1d_pairs`, `create_funct_1d_array`

Module

Foundation

num_points_funct_1d (: : Function : Length)

Number of control points of the function.

num_points_funct_1d calculates the number of control points of Function.

Parameters

▷ **Function** (input_control) ..... function_1d ⇝ *real / integer*
Input function.

▷ **Length** (output_control) ..... integer ⇝ *integer*
Number of control points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array

Module

Foundation

read_funct_1d (: : FileName : Function)

Read a function from a file.

The operator read_funct_1d reads the contents of FileName and converts it into the function Function. The file has be generated by write_funct_1d.

Parameters

▷ **FileName** (input_control) ..... filename.read ⇝ *string*
Name of the file to be read.

▷ **Function** (output_control) ..... function_1d ⇝ *real / integer*
Function from the file.

Result

If the parameters are correct the operator read_funct_1d returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

fread_string, read_tuple

See also

write_funct_1d, write_image, write_region, open_file

Module

Foundation

sample_funct_1d (: : Function, XMin, XMax, XDist, Border : SampledFunction)

Sample a function equidistantly in an interval.

sample_funct_1d samples the input function Function in the interval [XMin,XMax] at equidistant points with the distance XDist. The last point lies in the interval if XMax-XMin is not an integer multiple of XDist. To obtain the samples, the input function is interpolated linearly. The parameter Border determines the values of the function Function outside of its domain. For Border='zero' these values are set to 0, for Border='constant' they are set to the corresponding value at the border, for Border='mirror' they are mirrored at the border, and for Border='cyclic' they are continued cyclically.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function.

▷ **XMin** (input_control) . . . . . number ⇝ *real* / integer
Minimum x value of the output function.

▷ **XMax** (input_control) . . . . . number ⇝ *real* / integer
Maximum x value of the output function.
Restriction: XMax > XMin

▷ **XDist** (input_control) . . . . . number ⇝ *real* / integer
Distance of the samples.
Restriction: XDist > 0

▷ **Border** (input_control) . . . . . string ⇝ *string*
Border treatment for the input function.
Default: 'constant'
List of values: Border ∈ {'zero', 'constant', 'mirror', 'cyclic'}

▷ **SampledFunction** (output_control) . . . . . function_1d ⇝ *real* / integer
Sampled function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

transform_funct_1d, create_funct_1d_array, create_funct_1d_pairs

Module

Foundation

scale_y_funct_1d (: : Function, Mult, Add : FunctionScaled)

Multiplication and addition of the y values.

scale_y_funct_1d multiplies and adds the y values of Function with the parameters Mult and Add.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function.

▷ **Mult** (input_control) . . . . . number ⇝ *real*
Factor for scaling of the y values.
Default: 2.0
Suggested values: Mult ∈ {0.1, 0.3, 0.5, 1.0, 2.0, 5.0, 10.0}

▷ **Add** (input_control) . . . . . number ⇝ *real*
Constant which is added to the y values.
Default: 0.0
Suggested values: Add ∈ {-10.0, -5.0, 1.0, 0.0, 5.0, 10.0}

▷ **FunctionScaled** (output_control) . . . . . function_1d ⇝ *real* / integer
Transformed function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array

Module

Foundation

smooth_funct_1d_gauss (: : Function, Sigma : SmoothedFunction)

Smooth an equidistant 1D function with a Gaussian function.

The operator smooth_funct_1d_gauss smooths a one-dimensional function with a Gaussian function. The function must be equidistant, i.e., created with create_funct_1d_array, sample_funct_1d or similar. At the function borders the function values are mirrored.

Note that the smoothing parameter Sigma must not be larger than (Length-2) / 7.8 * Function[1], with Length-2 being the number of control points of Function and Function[1] specifying the equidistance of the x values. The value of Length can, e.g., be determined with num_points_funct_1d.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Function to be smoothed.

▷ **Sigma** (input_control) . . . . . number ⇝ *real*
Sigma of the Gaussian function for the smoothing.
Default: 2.0
Suggested values: Sigma ∈ {0.5, 1.0, 2.0, 3.0, 4.0, 5.0}
Value range: 0.1 ≤ Sigma ≤ 50.0 (lin)
Minimum increment: 0.01
Recommended increment: 0.2

▷ **SmoothedFunction** (output_control) . . . . . function_1d ⇝ *real* / integer
Smoothed function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_array, sample_funct_1d

Possible Successors

match_funct_1d_trans, distance_funct_1d

Alternatives

smooth_funct_1d_mean

Module

Foundation

smooth_funct_1d_mean (: : Function, SmoothSize,
Iterations : SmoothedFunction)

Smooth an equidistant 1D function by averaging its values.

The operator smooth_funct_1d_mean smooths a one dimensional function by applying an average (mean) filter multiple times. The function must be equidistant, i.e., created with create_funct_1d_array, sample_funct_1d or similar. At the function borders the function values are mirrored.

Attention

If an even value instead of an odd value is given for SmoothSize, the routine uses the next larger odd value instead (this way the center of the filter mask is always explicitly determined).

Parameters

- ▷ **Function** (input_control) . . . . . function_1d ⇝ *integer* / real
  1D function.
- ▷ **SmoothSize** (input_control) . . . . . integer ⇝ *integer*
  Size of the averaging mask.
  Default: 9
  Suggested values: SmoothSize ∈ {1, 3, 5, 7, 9, 11, 13, 15, 21, 31, 51}
  Value range: 0 ≤ SmoothSize ≤ 1001 (lin)
  Minimum increment: 1
  Recommended increment: 2
- ▷ **Iterations** (input_control) . . . . . integer ⇝ *integer*
  Number of iterations for the smoothing.
  Default: 3
  Suggested values: Iterations ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9}
  Value range: 1 ≤ Iterations ≤ 100 (lin)
  Minimum increment: 1
  Recommended increment: 1
- ▷ **SmoothedFunction** (output_control) . . . . . function_1d ⇝ *real* / integer
  Smoothed function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_array, sample_funct_1d

Alternatives

smooth_funct_1d_gauss

Module

Foundation

transform_funct_1d (: : Function, Params : TransformedFunction)

Transform a function using given transformation parameters.

transform_funct_1d transforms the input function Function using the transformation parameters given in Params. The function Function is passed as a tuple (see create_funct_1d_array and create_funct_1d_pairs). The following model is used for the transformation between the two functions (see match_funct_1d_trans):

$$y_t(x) = a_1 y(a_3 x + a_4) + a_2 \ .$$

The output function TransformedFunction is obtained by transforming the x and y values of the input function separately with the above formula, i.e., the output function is not sampled again. Therefore, the parameter a_3 is restricted to $a_3 \neq 0.0$. To resample a function, the operator sample_funct_1d can be used.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function.
▷ **Params** (input_control) . . . . . number-array ⇝ *real*
Transformation parameters between the functions.
Number of elements: 4
▷ **TransformedFunction** (output_control) . . . . . function_1d ⇝ *real* / integer
Transformed function.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_funct_1d_pairs`, `create_funct_1d_array`, `match_funct_1d_trans`

Module

Foundation

write_funct_1d (: : Function, FileName :)

Write a function to a file.

The operator `write_funct_1d` writes the contents of `Function` to a file. The data is written in an ASCII format. Therefore, the file can be exchanged between different architectures (see also Tuple / String Operations). The data can be read by the operator `read_funct_1d`. There is no specific extension for this kind of file.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Function to be written.
▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
Name of the file to be written.

Result

If the parameters are correct the operator `write_funct_1d` returns the value `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_funct_1d_pairs`, `create_funct_1d_array`

Alternatives

`write_tuple`, `fwrite_string`

See also

`read_funct_1d`, `write_image`, `write_region`, `open_file`

Module

Foundation

x_range_funct_1d (: : Function : XMin, XMax)

Smallest and largest x value of the function.

x_range_funct_1d calculates the smallest and the largest x value of Function.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function.

▷ **XMin** (output_control) . . . . . number ⇝ *real*
Smallest x value.

▷ **XMax** (output_control) . . . . . number ⇝ *real*
Largest x value.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array

Module

Foundation

y_range_funct_1d (: : Function : YMin, YMax)

Smallest and largest y value of the function.

y_range_funct_1d calculates the smallest and the largest y value of Function.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function.

▷ **YMin** (output_control) . . . . . number ⇝ *real*
Smallest y value.

▷ **YMax** (output_control) . . . . . number ⇝ *real*
Largest y value.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array

Module

Foundation

zero_crossings_funct_1d (: : Function : ZeroCrossings)

Calculate the zero crossings of a function.

zero_crossings_funct_1d calculates the zero crossings ZeroCrossings of the function Function. A linear interpolation is applied to the function between its sampling points so that the coordinates of the zero crossing can be calculated exactly. If an entire line segment between two sampling points has a value of *0*, only the end points of its supporting interval are returned.

Parameters

▷ **Function** (input_control) . . . . . function_1d ⇝ *real* / integer
Input function

▷ **ZeroCrossings** (output_control) . . . . . real-array ⇝ *real*
Zero crossings of the input function

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_funct_1d_pairs, create_funct_1d_array, smooth_funct_1d_gauss, smooth_funct_1d_mean

Module

Foundation

26.3 Geometry

angle_ll (: : RowA1, ColumnA1, RowA2, ColumnA2, RowB1, ColumnB1, RowB2, ColumnB2 : Angle)

Calculate the angle between two lines.

The operator angle_ll calculates the angle between two lines. As input the coordinates of two points on the first line (RowA1,ColumnA1, RowA2,ColumnA2) and on the second line (RowB1,ColumnB1, RowB2,ColumnB2) are expected. The calculation is performed as follows: We interpret the lines as vectors with starting points RowA1,ColumnA1 and RowB1,ColumnB1 and end points RowA2,ColumnA2 and RowB2,ColumnB2, respectively. Rotating the vector A counter clockwise onto the vector B (the center of rotation is the intersection point of the two lines) yields the angle. The result depends on the order of the points and on the order of the lines. The parameter Angle returns the angle in radians, ranging from $-\pi \leq \text{Angle} \leq \pi$.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed angles is always *N*.

Parameters

▷ **RowA1** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the first point of the first line.

▷ **ColumnA1** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the first line.

▷ **RowA2** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the first line.

▷ **ColumnA2** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the first line.

▷ **RowB1** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the first point of the second line.

▷ **ColumnB1** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the second line.

▷ **RowB2** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the second line.

▷ **ColumnB2** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the second line.

▷ **Angle** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Angle between the lines [rad].

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
RowA1 := 255
ColumnA1 := 10
RowA2 := 255
ColumnA2 := 501
gen_contour_polygon_xld (Contour, [RowA1,RowA2], [ColumnA1,ColumnA2])
RowB1 := 255
ColumnB1 := 255
for I := 5 to 360 by 5
  RowB2 := 255 - sin(rad(I)) * 200
  ColumnB2 := 255 + cos(rad(I)) * 200
  gen_contour_polygon_xld (Contour, [RowB1,RowB2], [ColumnB1,ColumnB2])
  angle_ll (RowA1, ColumnA1, RowA2, ColumnA2,\
            RowB1, ColumnB1, RowB2, ColumnB2, Angle)
  AngleDeg := deg(Angle)
endfor
```

Result

`angle_ll` returns 2 (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

`angle_lx`

Module

Foundation

angle_lx (: : Row1, Column1, Row2, Column2 : Angle)

Calculate the angle between one line and the horizontal axis.

The operator `angle_lx` calculates the angle between one line and the horizontal axis. As input the coordinates of two points on the line (`Row1`,`Column1`, `Row2`,`Column2`) are expected. The calculation is performed as follows: We interpret the line as a vector with starting point `Row1`,`Column1` and end point `Row2`,`Column2`. The starting point is on the horizontal axis and defines the center of rotation in the following consideration. If the end point is above the horizontal axis, the angle (with positive sign) results from the rotation of the horizontal axis in counter clockwise direction onto the vector. If the end point is below the horizontal axis, the angle (with negative sign) results from the rotation of the horizontal axis in clockwise direction onto the vector. The result depends on the order of the two points defining the line. The parameter `Angle` returns the angle in radians, ranging from $-\pi \leq$ `Angle` $< \pi$.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed angles is always *N*.

Parameters

▷ **Row1** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate the first point of the line.

▷ **Column1** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the line.

▷ **Row2** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the line.

▷ **Column2** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the line.

▷ **Angle** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Angle between the line and the horizontal axis [rad].

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
RowA1 := 255
ColumnA1 := 10
RowA2 := 255
ColumnA2 := 501
gen_contour_polygon_xld (Contour, [RowA1,RowA2], [ColumnA1,ColumnA2])
Row1 := 255
Column1 := 255
for I := 5 to 360 by 5
  Row2 := 255 - sin(rad(I)) * 200
  Column2 := 255 + cos(rad(I)) * 200
  gen_contour_polygon_xld (Contour, [Row1,Row2], [Column1,Column2])
  angle_lx (Row1, Column1, Row2, Column2, Angle)
  AngleDeg := deg(Angle)
endfor
```

Result

`angle_lx` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

`angle_ll`

Module

Foundation

apply_distance_transform_xld (Contour : ContourOut : DistanceTransformID :)

Determine the pointwise distance of two contours using an XLD distance transform.

The operator `apply_distance_transform_xld` determines for each point in `Contour` the minimal distance to the reference contour using its XLD distance transform `DistanceTransformID`. The returned contour `ContourOut` consists of `Contour` with the attribute *'distance'* containing the calculated distances. They can be accessed by querying the attribute *'distance'* with `get_contour_attrib_xld`. See the operator reference of `get_contour_attrib_xld` for further information about contour attributes.

Note that the distances depend on the parameters of `create_distance_transform_xld`: The distances are clipped to the maximum distance specified by the parameter MaxDistance. The parameter Mode determines whether the distances are calculated *'point_to_point'* or *'point_to_segment'*. For further details please refer to the documentation of `create_distance_transform_xld`.

Parameters

▷ **Contour** (input_object) . . . . . xld_cont(-array) ⇝ *object*
Contour(s) for whose points the distances are calculated.

▷ **ContourOut** (output_object) . . . . . xld_cont(-array) ⇝ *object*
Copy of Contour containing the distances as an attribute.

▷ **DistanceTransformID** (input_control) . . . . . xld_dist_trans ⇝ *handle*
Handle of the XLD distance transform of the reference contour.

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_distance_transform_xld, read_distance_transform_xld, deserialize_distance_transform_xld, set_distance_transform_xld_param

Possible Successors

get_contour_attrib_xld, segment_contour_attrib_xld

Alternatives

distance_contours_xld

See also

get_contour_attrib_xld, set_distance_transform_xld_param, write_distance_transform_xld, serialize_distance_transform_xld, clear_distance_transform_xld

Module

Foundation

area_intersection_rectangle2 (: : Rect1Row, Rect1Column, Rect1Phi, Rect1Length1, Rect1Length2, Rect2Row, Rect2Column, Rect2Phi, Rect2Length1, Rect2Length2 : AreaIntersection)

Calculate the intersection area of oriented rectangles.

area_intersection_rectangle2 calculates the intersection area of two oriented rectangles (i.e., of type rectangle2), which are defined by their parameters (center (Rect1Row, Rect1Column), orientation Rect1Phi, and the half edge lengths Rect1Length1 and Rect1Length2) and (center (Rect2Row, Rect2Column), orientation Rect2Phi, and the half edge lengths Rect2Length1 and Rect2Length2), respectively. The intersection area is returned in AreaIntersection.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of results is always *N*.

Parameters

▷ **Rect1Row** (input_control) . . . . . rectangle2.center.y(-array) ⇝ *real* / integer
Center row coordinate of the first rectangle.

▷ **Rect1Column** (input_control) . . . . . rectangle2.center.x(-array) ⇝ *real* / integer
Center column coordinate of the first rectangle.

▷ **Rect1Phi** (input_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real* / integer
Angle between the positive horizontal axis and the first edge of the first rectangle (in radians).

▷ **Rect1Length1** (input_control) . . . . . rectangle2.hwidth(-array) ⇝ *real* / integer
Half length of the first edge of the first rectangle.

▷ **Rect1Length2** (input_control) . . . . . rectangle2.hheight(-array) ⇝ *real* / integer
Half length of the second edge of the first rectangle.

▷ **Rect2Row** (input_control) . . . . . rectangle2.center.y(-array) ⇝ *real* / integer
Center row coordinate of the second rectangle.

▷ **Rect2Column** (input_control) . . . . . rectangle2.center.x(-array) ⇝ *real* / integer
Center column coordinate of the second rectangle.

▷ **Rect2Phi** (input_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real* / integer
Angle between the positive horizontal axis and the first edge of the second rectangle (in radians).

▷ **Rect2Length1** (input_control) . . . . . rectangle2.hwidth(-array) ⇝ *real* / integer
Half length of the first edge of the second rectangle.

▷ **Rect2Length2** (input_control) . . . . . rectangle2.hheight(-array) ⇝ *real* / integer
Half length of the second edge of the second rectangle.

▷ **AreaIntersection** (output_control) . . . . . real(-array) ⇝ *real*
Intersection area of the first rectangle with the second rectangle.

Result

If the parameters are valid, the operator `area_intersection_rectangle2` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`gen_rectangle2`, `intersection_segments`, `intersection_segment_line`, `intersection_segment_contour_xld`, `intersection_line_contour_xld`, `intersection_contours_xld`

Module

Foundation

clear_distance_transform_xld (: : DistanceTransformID :)

Clear a XLD distance transform.

`clear_distance_transform_xld` clears the XLD distance transform `DistanceTransformID` that was previously created by `create_distance_transform_xld`.

Parameters

▷ **DistanceTransformID** (input_control) . . . . . xld_dist_trans ⇝ *handle*
Handle of the XLD distance transform.

Result

If all parameters are correct, the operator returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DistanceTransformID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_distance_transform_xld, write_distance_transform_xld, serialize_distance_transform_xld

See also

apply_distance_transform_xld, get_distance_transform_xld_contour, get_distance_transform_xld_param, set_distance_transform_xld_param, read_distance_transform_xld, deserialize_distance_transform_xld

Module

Foundation

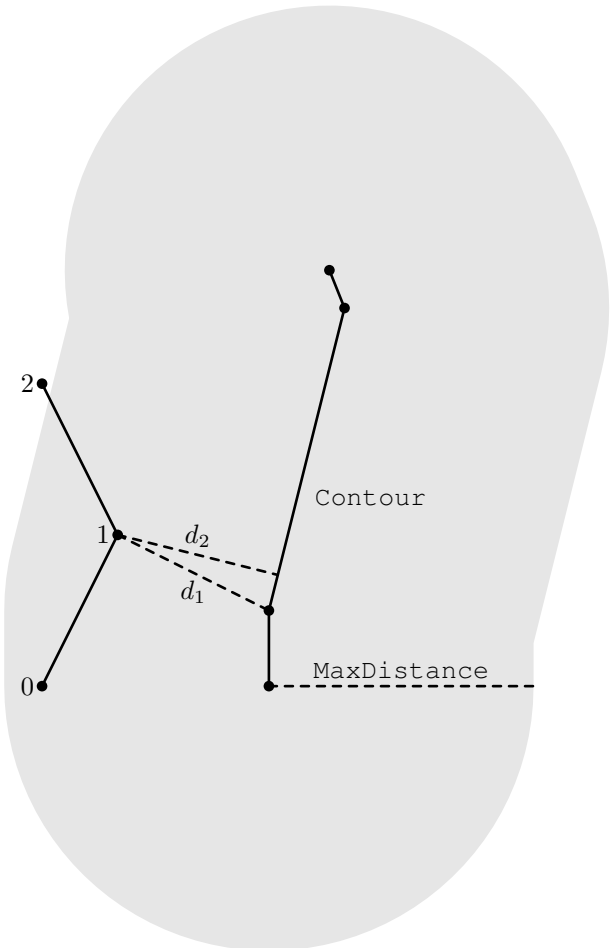
create_distance_transform_xld (Contour : : Mode, MaxDistance : DistanceTransformID)

Create the XLD distance transform.

create_distance_transform_xld creates the XLD distance transform of the reference contour Contour and returns the resulting handle in DistanceTransformID.

Once the XLD distance transform has been created, the operator apply_distance_transform_xld calculates pointwise distances from test contours to the reference contour Contour. More precisely, for each point of a test contour its minimal distance to the contours in Contour is calculated.

The parameter Mode determines which distances apply_distance_transform_xld calculates: *'point_to_point'* calculates the minimum distance to the base points of the Contour. In contrast, *'point_to_segment'* calculates the minimum distance to the contour segments in Contour (see the figure below).



The dashed lines illustrate the calculated distances starting from point 1 of the test contour on the left. The distance d_1 corresponds to the *'point_to_point'* mode, whereas the distance d_2 corresponds to the *'point_to_segment'* mode. The gray area illustrates the area within MaxDistance around the reference contour for which distances are calculated. The distance of point 2 of the test contour would be set to MaxDistance because it is outside of this area.

The parameter MaxDistance specifies a maximum distance from the reference contour Contour which is of interest to the caller. If the distance from a point of a test contour to the reference contour exceeds MaxDistance, the output distance of apply_distance_transform_xld is set to MaxDistance.

The operators create_distance_transform_xld and apply_distance_transform_xld are an alternative to distance_contours_xld, if the reference contour is repeatedly used. create_distance_transform_xld stores for each pixel in a relevant area around Contour its nearest points or segments (depending on Mode) of the reference contour Contour. This allows apply_distance_transform_xld to calculate the distances rather fast, nearly independent of the number of points or segments of the reference contour, the Mode and the position of the points of the test contour. However, the preparation of the XLD distance transform can take several seconds or minutes, depending on the number of points or segments of the reference contour and the relevant area around Contour which can be influenced by MaxDistance. Furthermore, create_distance_transform_xld is faster, if Mode is set to 'point_to_point'.

get_distance_transform_xld_contour and get_distance_transform_xld_param can be used to get back the reference contours and the parameters of the XLD distance transform DistanceTransformID.

Parameters

- ▷ **Contour** (input_object) ..... xld_cont(-array) ⇝ *object*
  Reference contour(s).
- ▷ **Mode** (input_control) ..... string ⇝ *string*
  Compute the distance to points ('point_to_point') or entire segments ('point_to_segment').
  Default: 'point_to_point'
  List of values: Mode ∈ {'point_to_point', 'point_to_segment'}
- ▷ **MaxDistance** (input_control) ..... real ⇝ *real* / integer
  Maximum distance of interest.
  Default: 20.0
- ▷ **DistanceTransformID** (output_control) ..... xld_dist_trans ⇝ *handle*
  Handle of the XLD distance transform.

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

apply_distance_transform_xld, write_distance_transform_xld, serialize_distance_transform_xld, clear_distance_transform_xld

See also

distance_contours_xld, get_distance_transform_xld_contour, read_distance_transform_xld, deserialize_distance_transform_xld, get_distance_transform_xld_param, set_distance_transform_xld_param

Module

Foundation

deserialize_distance_transform_xld (: : SerializedItemHandle : DistanceTransformID)

Deserialize an XLD distance transform.

deserialize_distance_transform_xld deserializes an XLD distance transform. The serialized XLD distance transform is defined by the handle SerializedItemHandle. The deserialized XLD distance transform is returned in DistanceTransformID.

Note that the previous values of DistanceTransformID are overwritten, if the handle already exists.

get_distance_transform_xld_contour can be used to get the reference contour that was used to build the XLD distance transform DistanceTransformID.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized XLD distance transform.

▷ **DistanceTransformID** (output_control) . . . . . xld_dist_trans ⇝ *handle*
Handle of the deserialized XLD distance transform.

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

apply_distance_transform_xld, get_distance_transform_xld_contour, get_distance_transform_xld_param

See also

serialize_distance_transform_xld, write_distance_transform_xld, read_distance_transform_xld, create_distance_transform_xld, set_distance_transform_xld_param, clear_distance_transform_xld

Module

Foundation

distance_cc (Contour1, Contour2 : : Mode : DistanceMin, DistanceMax)

Calculate the distance between two contours.

The operator distance_cc calculates the minimum and maximum distance between the base points of two contours (Contour1 and Contour2). The parameters DistanceMin and DistanceMax contain the resulting distance.

The parameter Mode sets the type of computing the distance: *'point_to_point'* only determines the minimum and maximum distance between the base points of the contours. This results in faster algorithm, but may lead to inaccurate minimum distances. In contrast, *'point_to_segment'* determines the actual minimum distance between the contour segments.

In both cases, the search algorithm has a quadratic complexity (n*n). If only the minimum distance is required, the operator distance_cc_min can be used alternatively since it offers algorithms with a complexity of n*log(n).

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed distances is always *N*.

Parameters

▷ **Contour1** (input_object) . . . . . xld_cont(-array) ⇝ *object*
First input contour.

▷ **Contour2** (input_object) . . . . . xld_cont(-array) ⇝ *object*
Second input contour.

▷ **Mode** (input_control) . . . . . string(-array) ⇝ *string*
Distance calculation mode.
Default: 'point_to_point'
List of values: Mode ∈ {'point_to_point', 'point_to_segment'}

▷ **DistanceMin** (output_control) . . . . . real(-array) ⇝ *real*
Minimum distance between both contours.

▷ **DistanceMax** (output_control) . . . . . real(-array) ⇝ *real*
Maximum distance between both contours.

Example

```
gen_contour_polygon_rounded_xld(Cont1, [0,100,100,0,0], [0,0,100,100,0],
                                [50,50,50,50,50], 0.5);
gen_contour_polygon_rounded_xld(Cont2, [41,91,91,41,41], [41,41,91,91,41],
                                [25,25,25,25,25], 0.5);
distance_cc(Cont1, Cont2, 'point_to_point', &distance_min, &distance_max);
```

Result

distance_cc returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

distance_sc, distance_pc, distance_cc_min, distance_contours_xld

See also

distance_sr, distance_pr

Module

Foundation

distance_cc_min (Contour1, Contour2 : : Mode : DistanceMin)

Calculate the minimum distance between two contours.

distance_cc_min calculates the minimum distance between two contours Contour1 and Contour2. The minimum distance is returned in DistanceMin.

The parameter Mode sets the type of computing the distance. *'point_to_point'* determines the distance between the closest contour points, *'fast_point_to_segment'* calculates the distance between the line segments adjacent to these points, and *'point_to_segment'* determines the actual minimum distance between the contour segments.

While *'point_to_point'* and *'fast_point_to_segment'* are efficient algorithms with a complexity of n*log(n), *'point_to_segment'* has quadratic complexity and thus takes a longer time to execute, especially for contours with many line segments.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed distances is always *N*.

Parameters

▷ **Contour1** (input_object) ..... xld_cont(-array) ⇝ *object*
First input contour.

▷ **Contour2** (input_object) ..... xld_cont(-array) ⇝ *object*
Second input contour.

▷ **Mode** (input_control) ..... string(-array) ⇝ *string*
Distance calculation mode.
Default: 'fast_point_to_segment'
List of values: Mode ∈ {'point_to_point', 'point_to_segment', 'fast_point_to_segment'}

▷ **DistanceMin** (output_control) ..... number(-array) ⇝ *real*
Minimum distance between the two contours.

Example

```
gen_contour_polygon_rounded_xld(Cont1, [0,100,100,0,0], [0,0,100,100,0],
                                [50,50,50,50,50], 0.5);
gen_contour_polygon_rounded_xld(Cont2, [41,91,91,41,41], [41,41,91,91,41],
                                [25,25,25,25,25], 0.5);
distance_cc_min(Cont1, Cont2, "fast_point_to_segment", &distance_min);
```

Result

distance_cc_min returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

distance_sc, distance_pc, distance_cc, distance_contours_xld

See also

distance_sr, distance_pr

Module

Foundation

distance_cc_min_points (Contour1, Contour2 : : Mode : DistanceMin, Row1, Column1, Row2, Column2)

Calculate the minimum distance between two contours and the points used for the calculation.

distance_cc_min_points calculates the minimum distance between Contour1 and Contour2. The minimum distance is returned in DistanceMin. In comparison to distance_cc_min, this operator also returns the points on the contours that provide the minimum distance. The point on Contour1 is returned in Row1 and Column1; the point on Contour2 is returned in Row2 and Column2.

The parameter Mode sets the type of computing the distance. *'fast_point_to_segment'* calculates the distance between the line segments adjacent to the closest contour points, and *'point_to_segment'* determines the actual minimum distance between the contour segments.

While *'fast_point_to_segment'* is an efficient algorithm with a complexity of n*log(n), *'point_to_segment'* has quadratic complexity and thus takes a longer time to execute, especially for contours with many line segments.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed distances is always *N*.

Parameters

▷ **Contour1** (input_object) ..... xld_cont(-array) ⇝ *object*
First input contour.

▷ **Contour2** (input_object) ..... xld_cont(-array) ⇝ *object*
Second input contour.

▷ **Mode** (input_control) ..... string(-array) ⇝ *string*
Distance calculation mode.
Default: 'fast_point_to_segment'
List of values: Mode ∈ {'point_to_segment', 'fast_point_to_segment'}

▷ **DistanceMin** (output_control) ..... number(-array) ⇝ *real*
Minimum distance between the two contours.

▷ **Row1** (output_control) ..... point.y(-array) ⇝ *real*
Row coordinate of the point on Contour1.

▷ **Column1** (output_control) ..... point.x(-array) ⇝ *real*
Column coordinate of the point on Contour1.

▷ **Row2** (output_control) ..... point.y(-array) ⇝ *real*
Row coordinate of the point on Contour2.

▷ **Column2** (output_control) ..... point.x(-array) ⇝ *real*
Column coordinate of the point on Contour2.

Example

```
gen_contour_polygon_rounded_xld(Cont1, [0,100,100,0,0], [0,0,100,100,0],
                                [50,50,50,50,50], 0.5);
gen_contour_polygon_rounded_xld(Cont2, [41,91,91,41,41], [41,41,91,91,41],
                                [25,25,25,25,25], 0.5);
distance_cc_min_points(Cont1, Cont2, "fast_point_to_segment", &distance_min,
                       &Row1, &Column1, &Row2, &Column2);
```

Result

distance_cc_min_points returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

distance_sc, distance_pc, distance_cc, distance_contours_xld, distance_cc_min

See also

distance_sr, distance_pr

Module

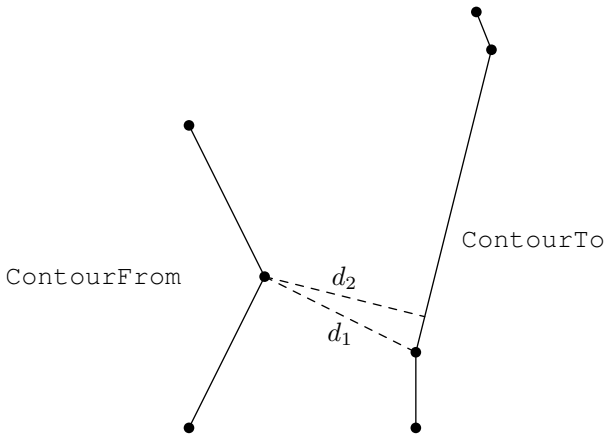
Foundation

distance_contours_xld (ContourFrom, ContourTo : ContourOut : Mode :)

Calculate the pointwise distance from one contour to another.

The operator distance_contours_xld calculates for each point in ContourFrom the minimal distance to the contours in ContourTo. The operator returns the contour ContourOut which consists of ContourFrom containing the computed distances in the attribute *'distance'*. The distances can be accessed by querying the attribute *'distance'* with the operator get_contour_attrib_xld. See the operator reference of get_contour_attrib_xld for further information about contour attributes.

The parameter Mode determines which distances are calculated for each point in ContourFrom: *'point_to_point'* calculates the minimal distance to the base points of ContourTo. In contrast, *'point_to_segment'* calculates the minimum distance to the contour segments in ContourTo (see the figure below).



The dashed lines illustrate the calculated distances starting from a point of ContourFrom on the left. The distance d_1 corresponds to the *'point_to_point'* mode, whereas the distance d_2 corresponds to the *'point_to_segment'* mode.

Note that in many applications the mode *'point_to_segment'* suggests itself for an accurate result, whereas *'point_to_point'* is considerably faster. If m is the number of points in ContourFrom, and n the number of points in ContourTo, then distance_contours_xld has complexity O(mlog(n)) for *'point_to_point'*, and O(mn) for *'point_to_segment'*. In case the template contour ContourTo is repeatedly used, the operators create_distance_transform_xld and apply_distance_transform_xld can be used as an alternative to distance_contours_xld. For further details, please refer to the documentation of create_distance_transform_xld.

Parameters

▷ **ContourFrom** (input_object) ..... xld_cont(-array) ⇝ *object*
Contours for whose points the distances are calculated.

▷ **ContourTo** (input_object) ..... xld_cont(-array) ⇝ *object*
Contours to which the distances are calculated to.

▷ **ContourOut** (output_object) ..... xld_cont(-array) ⇝ *object*
Copy of ContourFrom containing the distances as an attribute.

▷ **Mode** (input_control) ..... string ⇝ *string*
Compute the distance to points ('point_to_point') or to entire segments ('point_to_segment').
Default: 'point_to_point'
List of values: Mode ∈ {'point_to_point', 'point_to_segment'}

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

get_contour_attrib_xld, segment_contour_attrib_xld

Alternatives

apply_distance_transform_xld, distance_cc, distance_cc_min

Module

Foundation

distance_lc (Contour : : Row1, Column1, Row2, Column2 : DistanceMin, DistanceMax)

Calculate the distance between a line and one contour.

The operator distance_lc calculates the orthogonal distance between a line and the segments of a contour. As input the coordinates of two points on a line (Row1,Column1, Row2,Column2) and a contour (Contour) are expected. The parameters DistanceMin and DistanceMax return the result of the calculation.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of distances is always *N*.

Parameters

- ▷ **Contour** (input_object) ..... xld_cont(-array) ⇝ *object*
  Input contour.
- ▷ **Row1** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the first point of the line.
- ▷ **Column1** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the first point of the line.
- ▷ **Row2** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the second point of the line.
- ▷ **Column2** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the second point of the line.
- ▷ **DistanceMin** (output_control) ..... real(-array) ⇝ *real*
  Minimum distance between the line and the contour.
- ▷ **DistanceMax** (output_control) ..... real(-array) ⇝ *real*
  Maximum distance between the line and the contour.

Result

distance_lc returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

distance_pc, distance_sc, distance_cc, distance_cc_min

See also

distance_lr, distance_pr, distance_sr

Module

Foundation

distance_lr (Region : : Row1, Column1, Row2, Column2 : DistanceMin, DistanceMax)

Calculate the distance between a line and a region.

The operator distance_lr calculates the orthogonal distance between a line and a region. As input the coordinates of two points on a line (Row1,Column1, Row2,Column2) and a region are expected. The parameters DistanceMin and DistanceMax return the result of the calculation.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of distances is always *N*.

Attention

Due to efficiency of distance_lr holes are ignored.

Parameters

▷ **Region** (input_object) ..... region(-array) ⇝ *object*
Input region.

▷ **Row1** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinate of the first point of the line.

▷ **Column1** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the line.

▷ **Row2** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the line.

▷ **Column2** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the line.

▷ **DistanceMin** (output_control) ..... real(-array) ⇝ *real*
Minimum distance between the line and the region

▷ **DistanceMax** (output_control) ..... real(-array) ⇝ *real*
Maximum distance between the line and the region

Example

```
gen_circle (Circle, 200, 200, 100.5)
Column1 := 300
Column2 := 400
for Row := 50 to 350 by 50
  gen_contour_polygon_xld (Line, [Row,Row], [Column1,Column2])
  distance_lr (Circle, Row, Column1, Row, Column2, DistanceMin, DistanceMax)
endfor
```

Result

`distance_lr` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`distance_lc`, `distance_pr`, `distance_sr`, `diameter_region`

See also

`hamming_distance`, `select_region_point`, `test_region_point`, `smallest_rectangle2`

Module

Foundation

distance_pc (Contour : : Row, Column : DistanceMin, DistanceMax)

Calculate the distance between a point and one contour.

The operator `distance_pc` calculates the distance between one or several points and a single contour. As input the coordinates of the points (`Row`,`Column`) and the contour (`Contour`) are expected. The parameters `DistanceMin` and `DistanceMax` return the result of the calculation. Note that the result corresponds to the distances between the points and the segments of the contour and not the distances between the points and the base points of the contour (see also `distance_contours_xld`).

Parameters

▷ **Contour** (input_object) . . . . . xld_cont ⇝ *object*
Input contour.
▷ **Row** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the point.
▷ **Column** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the point.
▷ **DistanceMin** (output_control) . . . . . real(-array) ⇝ *real*
Minimum distance between the point and the contour.
▷ **DistanceMax** (output_control) . . . . . real(-array) ⇝ *real*
Maximum distance between the point and the contour.

Result

`distance_pc` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

`distance_lc`, `distance_sc`, `distance_cc`, `distance_cc_min`

See also

`distance_pr`, `distance_lr`, `distance_sr`, `hamming_distance`, `select_xld_point`, `test_xld_point`

Module

Foundation

distance_pl (: : Row, Column, Row1, Column1, Row2, Column2 : Distance)

Calculate the distance between one point and one line.

The operator `distance_pl` calculates the orthogonal distance between points (`Row`,`Column`) and lines, given by two arbitrary points on the line. The result is passed in `Distance`.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed distances is always *N*.

Parameters

▷ **Row** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the point.
▷ **Column** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column of the point.
▷ **Row1** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the first point of the line.
▷ **Column1** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the line.
▷ **Row2** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the line.
▷ **Column2** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the line.
▷ **Distance** (output_control) . . . . . real(-array) ⇝ *real*
Distance between the points.

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
draw_point (WindowHandle, Row, Column)
gen_cross_contour_xld (Cross, Row, Column, 15, 0)
draw_line (WindowHandle, Row1, Column1, Row2, Column2)
gen_contour_polygon_xld (Contour, [Row1,Row2], [Column1,Column2])
distance_pl (Row, Column, Row1, Column1, Row2, Column2, Distance)
```

Result

`distance_pl` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

`distance_ps`

See also

`distance_pp`, `distance_pr`

Module

Foundation

distance_point_line (: : PointX, PointY, PointZ, Point1X, Point1Y, Point1Z, Point2X, Point2Y, Point2Z : Distance)

Calculate the distance between a 3D point and a 3D line given by two points on the line.

`distance_point_line` calculates the orthogonal distance between 3D points and 3D lines. The lines are specified by two points on the line ($\mathbf{Q}$ and $\mathbf{R}$). The result is returned in `Distance`. The input tuples can define 1 or n points. The points are specified by (`PointX`, `PointY`, `PointZ`). The line is specified by two points on the line (`Point1X`, `Point1Y`, `Point1Z`) and (`Point2X`, `Point2Y`, `Point2Z`).

Let $\mathbf{P}$ denote the point (`PointX`, `PointY`, `PointZ`), $\mathbf{Q}$ the first point (`Point1X`, `Point1Y`, `Point1Z`) on the line, and $\mathbf{R}$ the second point (`Point2X`, `Point2Y`, `Point2Z`) on the line. First, the line direction $\mathbf{V} = \mathbf{R} - \mathbf{Q}$ and its length $\|\mathbf{V}\|$ are computed. If $\|\mathbf{V}\| = 0$, the points $\mathbf{Q}$ and $\mathbf{R}$ do not define a line and an error is returned. Furthermore, let D denote the distance `Distance`. Then, $D = \|(\mathbf{P} - \mathbf{Q}) \times \mathbf{V}\| / \|\mathbf{V}\|$, where $\times$ denotes the cross product of two vectors.

Parameters

▷ **PointX** (input_control) ..... point3d.x(-array) ⇝ *real*
X coordinate of the original points.

▷ **PointY** (input_control) ..... point3d.y(-array) ⇝ *real*
Y coordinate of the original points.

▷ **PointZ** (input_control) ..... point3d.z(-array) ⇝ *real*
Z coordinate of the original points.

▷ **Point1X** (input_control) ..... point3d.x(-array) ⇝ *real* / integer
X coordinate of the first point on the line.

▷ **Point1Y** (input_control) ..... point3d.y(-array) ⇝ *real* / integer
Y coordinate of the first point on the line.

▷ **Point1Z** (input_control) ..... point3d.z(-array) ⇝ *real* / integer
Z coordinate of the first point on the line.

▷ **Point2X** (input_control) ..... point3d.x(-array) ⇝ *real* / integer
X coordinate of the second point on the line.

▷ **Point2Y** (input_control) . . . . . point3d.y(-array) ⇝ *real* / integer
Y coordinate of the second point on the line.

▷ **Point2Z** (input_control) . . . . . point3d.z(-array) ⇝ *real* / integer
Z coordinate of the second point on the line.

▷ **Distance** (output_control) . . . . . real(-array) ⇝ *real*
Distance between the points and the lines.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_line_of_sight`

Alternatives

`distance_point_pluecker_line`

See also

`point_pluecker_line_to_hom_mat3d`

Module

Foundation

distance_point_pluecker_line (: : PointX, PointY, PointZ, LineDirectionX, LineDirectionY, LineDirectionZ, LineMomentX, LineMomentY, LineMomentZ : Distance)

Calculate the distance between a 3D point and a 3D line given by Plücker coordinates.

`distance_point_pluecker_line` calculates the orthogonal distance between 3D points and 3D lines. The result is returned in `Distance`. The input tuples can define 1 or n points $\mathbf{P}$ and lines, respectively. The points are specified by (`PointX`, `PointY`, `PointZ`). The lines are given in Plücker coordinates $\mathbf{L}$ (`LineDirectionX`, `LineDirectionY`, `LineDirectionZ`) and $\mathbf{M}$ (`LineMomentX`, `LineMomentY`, `LineMomentZ`). For the definition of Plücker coordinates, see "`Solution Guide III-C - 3D Vision`".

Let $\mathbf{P}$ denote the point (`PointX`, `PointY`, `PointZ`), $\mathbf{L}$ the direction (`LineDirectionX`, `LineDirectionY`, `LineDirectionZ`), and $\mathbf{M}$ the moment (`LineMomentX`, `LineMomentY`, `LineMomentZ`) of the Plücker line. Furthermore, let D denote the distance `Distance`. Then, $D = \|\mathbf{P} \times \mathbf{L} - \mathbf{M}\|$, where $\times$ denotes the cross product of two vectors.

Parameters

▷ **PointX** (input_control) . . . . . point3d.x(-array) ⇝ *real*
X coordinates of the original points.

▷ **PointY** (input_control) . . . . . point3d.y(-array) ⇝ *real*
Y coordinates of the original points.

▷ **PointZ** (input_control) . . . . . point3d.z(-array) ⇝ *real*
Z coordinates of the original points.

▷ **LineDirectionX** (input_control) . . . . . point3d.x(-array) ⇝ *real* / integer
X component of the direction vector of the corresponding line.

▷ **LineDirectionY** (input_control) . . . . . point3d.y(-array) ⇝ *real* / integer
Y component of the direction vector of the corresponding line.

▷ **LineDirectionZ** (input_control) . . . . . point3d.z(-array) ⇝ *real* / integer
Z component of the direction vector of the corresponding line.

▷ **LineMomentX** (input_control) . . . . . point3d.x(-array) ⇝ *real* / integer
X component of the moment vector of the corresponding line.

▷ **LineMomentY** (input_control) . . . . . point3d.y(-array) ⇝ *real* / integer
Y component of the moment vector of the corresponding line.

▷ **LineMomentZ** (input_control) . . . . . point3d.z(-array) ⇝ *real* / integer
Z component of the moment vector of the corresponding line.

▷ **Distance** (output_control) . . . . . real(-array) ⇝ *real*
Distance between the points and the lines.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_to_pluecker_line, point_direction_to_pluecker_line

Alternatives

distance_point_line

See also

point_pluecker_line_to_hom_mat3d

Module

Foundation

distance_pp (: : Row1, Column1, Row2, Column2 : Distance)

Calculate the distance between two points.

The operator distance_pp calculates the distance between pairs of points according to the following formula:

$$\texttt{Distance} = \sqrt{((\texttt{Row1} - \texttt{Row2})^2 + (\texttt{Column1} - \texttt{Column2})^2)}$$

The result is returned in Distance.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed distances is always *N*.

Parameters

▷ **Row1** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the first point.

▷ **Column1** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the first point.

▷ **Row2** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the second point.

▷ **Column2** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the second point.

▷ **Distance** (output_control) . . . . . real(-array) ⇝ *real*
Distance between the points.

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
draw_point (WindowHandle, Row1, Column1)
gen_cross_contour_xld (Cross, Row1, Column1, 15, 0)
draw_point (WindowHandle, Row2, Column2)
gen_cross_contour_xld (Cross, Row2, Column2, 15, 0)
distance_pp (Row1, Column1, Row2, Column2, Distance)
```

Result

distance_pp returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

distance_ps

See also

distance_pl, distance_pr

Module

Foundation

distance_pr (Region : : Row, Column : DistanceMin, DistanceMax)

Calculate the distance between a point and a region.

The operator distance_pr calculates the distance between a point and one region. As input the coordinates of the points (Row,Column) and a region are expected. If a point is inside of the region, its minimum distance is zero. The parameters DistanceMin and DistanceMax return the result of the calculation.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of distances is always *N*.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Input region.
- ▷ **Row** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the point.
- ▷ **Column** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the point.
- ▷ **DistanceMin** (output_control) ..... real(-array) ⇝ *real*
  Minimum distance between the point and the region.
- ▷ **DistanceMax** (output_control) ..... real(-array) ⇝ *real*
  Maximum distance between the point and the region.

Example

```
gen_circle (Circle, 200, 200, 100.5)
draw_point (WindowHandle, Row, Column)
gen_cross_contour_xld (Cross, Row, Column, 15, 0)
distance_pr (Circle, Row, Column, DistanceMin, DistanceMax)
```

Result

distance_pr returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

distance_pc, distance_lr, distance_sr, diameter_region

See also

hamming_distance, select_region_point, test_region_point, smallest_rectangle2

Module

Foundation

distance_ps (: : Row, Column, Row1, Column1, Row2, Column2 : DistanceMin, DistanceMax)

Calculate the distances between a point and a line segment.

The operator distance_ps calculates the minimum and maximum distance between a point (Row,Column) and a line segment which is represented by the start point (Row1,Column1) and the end point (Row2,Column2). DistanceMax is the maximum distance between the point and the end points of the line segment. DistanceMin is identical to distance_pl in the case that the point is "between" the two endpoints. Otherwise, the minimum distance to one of the end points is used.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed distances is always *N*.

Parameters

- ▷ **Row** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the first point.
- ▷ **Column** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the first point.
- ▷ **Row1** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the first point of the line segment.
- ▷ **Column1** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the first point of the line segment.
- ▷ **Row2** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the second point of the line segment.
- ▷ **Column2** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the second point of the line segment.
- ▷ **DistanceMin** (output_control) ..... real(-array) ⇝ *real*
  Minimum distance between the point and the line segment.
- ▷ **DistanceMax** (output_control) ..... real(-array) ⇝ *real*
  Maximum distance between the point and the line segment.

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
draw_point (WindowHandle, Row, Column)
gen_cross_contour_xld (Cross, Row, Column, 15, 0)
draw_line (WindowHandle, Row1, Column1, Row2, Column2)
gen_contour_polygon_xld (Contour, [Row1,Row2], [Column1,Column2])
distance_ps (Row, Column, Row1, Column1, Row2, Column2, \
             DistanceMin, DistanceMax)
```

Result

distance_ps returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

distance_pl

See also

distance_pp, distance_pr

Module

Foundation

distance_rr_min (Regions1, Regions2 : : : MinDistance, Row1, Column1, Row2, Column2)

Minimum distance between the contour pixels of two regions each.

The operator distance_rr_min calculates the minimum distance of pairs of regions. If several regions are passed in Regions1 and Regions2 the distance between the contour pixels of each i-th element is calculated and then forms the i-th entry in the output parameter MinDistance. The Euclidean distance is used. The parameters (Row1, Column1) and (Row2, Column2) indicate the position on the contour of Regions1 and Regions2, respectively, that have the minimum distance.

The calculation is carried out by comparing all contour pixels (see get_region_contour). This means in particular that holes in the regions are ignored. Furthermore, it is not checked whether one region lies completely within the other region. In this case, a minimum distance > 0 is returned. It is also not checked whether both regions contain a nonempty intersection. In the latter case, a minimum distance of 0 or > 0 can be returned, depending on whether the contours of the regions contain a common point or not.

Attention

Both input parameters must contain the same number of regions. The regions must not be empty.

Parameters

- ▷ **Regions1** (input_object) . . . region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **Regions2** (input_object) . . . region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **MinDistance** (output_control) . . . real(-array) ⇝ *real*
  Minimum distance between contours of the regions.
  Assertion: 0 <= MinDistance
- ▷ **Row1** (output_control) . . . point.y(-array) ⇝ *integer*
  Line index on contour in Regions1.
- ▷ **Column1** (output_control) . . . point.x(-array) ⇝ *integer*
  Column index on contour in Regions1.
- ▷ **Row2** (output_control) . . . point.y(-array) ⇝ *integer*
  Line index on contour in Regions2.
- ▷ **Column2** (output_control) . . . point.x(-array) ⇝ *integer*
  Column index on contour in Regions2.

Complexity

If $N1$,$N2$ are the lengths of the contours the runtime complexity is $O(N1 * N2)$.

Result

The operator distance_rr_min returns the value 2 (H_MSG_TRUE) if the input is not empty. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

distance_rr_min_dil, dilation1, intersection

Module

Foundation

distance_rr_min_dil (Regions1, Regions2 : : : MinDistance)

Minimum distance between two regions with the help of dilation.

The operator distance_rr_min_dil calculates the minimum distance between pairs of regions, by iteratively applying dilations on both regions until their intersection is non empty. If several regions are passed in Regions1 and Regions2 the distance between the i-th elements in each case is calculated. It then forms the i-th entry in the output parameter MinDistance. The calculation is carried out with the help of dilation with the Golay element *'h'* (see dilation_golay for further information). The result is:

$n_i * 2 - 1$ where n_i is the number of iterations. The mask *'h'* has the effect that precisely the maximum metrics are calculated.

Attention

Both parameters must contain the same number of regions. The regions must not be empty.

Parameters

- ▷ **Regions1** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **Regions2** (input_object) ..... region(-array) ⇝ *object*
  Regions to be examined.
- ▷ **MinDistance** (output_control) ..... integer(-array) ⇝ *integer*
  Minimum distances of the regions.
  Assertion: -1 <= MinDistance

Result

The operator distance_rr_min_dil returns the value 2 (H_MSG_TRUE) if the input is not empty. Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

threshold, regiongrowing, connection

Alternatives

distance_rr_min, dilation1, intersection

Module

Foundation

distance_sc (Contour : : Row1, Column1, Row2,
Column2 : DistanceMin, DistanceMax)

Calculate the distance between a line segment and one contour.

The operator distance_sc calculates the distance between a line segment and the line segments of a contour. Row1, Column1, Row2, Column2 are the start and end coordinates of a line segment, Contour represents the input contour. The parameters DistanceMin and DistanceMax contain the resulting distances.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of distances is always *N*.

Parameters

▷ **Contour** (input_object) . . . . . xld_cont(-array) ⇝ *object*
Input contour.

▷ **Row1** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the first point of the line segment.

▷ **Column1** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the line segment.

▷ **Row2** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the line segment.

▷ **Column2** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the line segment.

▷ **DistanceMin** (output_control) . . . . . real(-array) ⇝ *real*
Minimum distance between the line segment and the contour.

▷ **DistanceMax** (output_control) . . . . . real(-array) ⇝ *real*
Maximum distance between the line segment and the contour.

Result

distance_sr returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

distance_lc, distance_pc, distance_cc, distance_cc_min

See also

distance_sr, distance_lr, distance_pr, select_xld_point, test_xld_point

Module

Foundation

distance_sl (: : RowA1, ColumnA1, RowA2, ColumnA2, RowB1, ColumnB1, RowB2, ColumnB2 : DistanceMin, DistanceMax)

Calculate the distances between a line segment and a line.

The operator distance_sl calculates the minimum and maximum orthogonal distance between a line segment and a line. As input the coordinates of two points on the line segment (RowA1,ColumnA1,RowA2,ColumnA2) and on the line (RowB1,ColumnB1,RowB2,ColumnB2) are expected. The parameters DistanceMin and DistanceMax return the result of the calculation. If the line segments are intersecting, DistanceMin returns zero.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed distances is always *N*.

Parameters

▷ **RowA1** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the first point of the line segment.

▷ **ColumnA1** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the line segment.

▷ **RowA2** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the line segment.
▷ **ColumnA2** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the line segment.
▷ **RowB1** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the first point of the line.
▷ **ColumnB1** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the first point of the line.
▷ **RowB2** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinate of the second point of the line.
▷ **ColumnB2** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinate of the second point of the line.
▷ **DistanceMin** (output_control) . . . . . real(-array) ⇝ *real*
Minimum distance between the line segment and the line.
▷ **DistanceMax** (output_control) . . . . . real(-array) ⇝ *real*
Maximum distance between the line segment and the line.

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
Row1 := 300
Column1 := 200
Row2 := 100
Column2 := 400
gen_contour_polygon_xld (Line1, [Row1,Row2], [Column1,Column2])
dev_display (Line1)
Column := 50
Row := 100
Offset := 0
for RowEnd := 100 to 500 by 100
  gen_contour_polygon_xld (Line2, [Row,RowEnd], \
                           [Column+Offset,Column+Offset])
  dev_display (Line1)
  distance_sl (Row, Column+Offset, RowEnd, Column+Offset, Row1, Column1, \
               Row2, Column2,DistanceMin, DistanceMax)
  Offset := Offset+40
endfor
```

Result

`distance_sl` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

`distance_pl`

See also

`distance_ps`, `distance_pp`

Module

Foundation

distance_sr (Region : : Row1, Column1, Row2, Column2 : DistanceMin, DistanceMax)

Calculate the distance between a line segment and one region.

The operator distance_sr calculates the distance between a line segment and a region. Row1, Column1, Row2, Column2 are the start and end coordinates of a line segment. The parameters DistanceMin and DistanceMax contain the resulting distances.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of distances is always *N*.

Attention

To enhance distance_sr, holes are ignored.

Parameters

- ▷ **Region** (input_object) ..... region(-array) ⇝ *object*
  Input region.
- ▷ **Row1** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the first point of the line segment.
- ▷ **Column1** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the first point of the line segment.
- ▷ **Row2** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the second point of the line segment.
- ▷ **Column2** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the second point of the line segment.
- ▷ **DistanceMin** (output_control) ..... real(-array) ⇝ *real*
  Minimum distance between the line segment and the region.
- ▷ **DistanceMax** (output_control) ..... real(-array) ⇝ *real*
  Maximum distance between the line segment and the region.

Example

```
gen_circle (Circle, 200, 200, 100.5)
Column1 := 300
Column2 := 400
for Row := 50 to 350 by 50
  gen_contour_polygon_xld (Line, [Row,Row], [Column1,Column2])
  distance_sr (Circle, Row, Column1, Row, Column2, DistanceMin, DistanceMax)
endfor
```

Result

distance_sr returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

distance_sc, distance_lr, distance_pr, diameter_region

See also

hamming_distance, select_region_point, test_region_point, smallest_rectangle2

Module

Foundation

distance_ss (: : RowA1, ColumnA1, RowA2, ColumnA2, RowB1, ColumnB1, RowB2, ColumnB2 : DistanceMin, DistanceMax)

Calculate the distances between two line segments.

The operator distance_ss calculates the minimum and maximum distance between two line segments. As input the coordinates of the start and end point of the first line segment (RowA1,ColumnA1, RowA2,ColumnA2) and of the second line segment (RowB1,ColumnB1,RowB2,ColumnB2) are used. The parameters DistanceMin and DistanceMax return the result of the calculation. If the line segments are intersecting, DistanceMin returns zero.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of computed distances is always *N*.

Parameters

- ▷ **RowA1** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the first point of the line segment.
- ▷ **ColumnA1** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the first point of the line segment.
- ▷ **RowA2** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the second point of the line segment.
- ▷ **ColumnA2** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the second point of the line segment.
- ▷ **RowB1** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the first point of the line.
- ▷ **ColumnB1** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column of the first point of the line.
- ▷ **RowB2** (input_control) ..... point.y(-array) ⇝ *real* / integer
  Row coordinate of the second point of the line.
- ▷ **ColumnB2** (input_control) ..... point.x(-array) ⇝ *real* / integer
  Column coordinate of the second point of the line.
- ▷ **DistanceMin** (output_control) ..... real(-array) ⇝ *real*
  Minimum distance between the line segments.
- ▷ **DistanceMax** (output_control) ..... real(-array) ⇝ *real*
  Maximum distance between the line segments.

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
Row1 := 300
Column1 := 200
Row2 := 100
Column2 := 300
gen_contour_polygon_xld (Line1, [Row1,Row2], [Column1,Column2])
Column := 100
Offset := 30
for Row := 40 to 100 by 20
  gen_contour_polygon_xld (Line2, [Row, Row+Offset], \
                           [Column, Column+Offset])
  distance_ss (Row, Column, Row+Offset, Column+Offset, Row1, Column1, \
               Row2, Column2, DistanceMin, DistanceMax)
  Offset := Offset+50
endfor
```

Result

distance_ss returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on internal data level.

Alternatives

distance_pp

See also

distance_pl, distance_ps

Module

Foundation

get_distance_transform_xld_contour (: Contour : DistanceTransformID :)

Get the reference contour used to build the XLD distance transform.

get_distance_transform_xld_contour returns the reference contour Contour that was used to build the XLD distance transform DistanceTransformID. This can be used for visualization purposes while comparing a given contour against the reference contour using apply_distance_transform_xld.

Parameters

▷ **Contour** (output_object) ..... xld_cont(-array) ⇝ *object*
Reference contour.

▷ **DistanceTransformID** (input_control) ..... xld_dist_trans ⇝ *handle*
Handle of the XLD distance transform.

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_distance_transform_xld, deserialize_distance_transform_xld

See also

get_distance_transform_xld_param, set_distance_transform_xld_param, create_distance_transform_xld, apply_distance_transform_xld, clear_distance_transform_xld, write_distance_transform_xld, serialize_distance_transform_xld

Module

Foundation

get_distance_transform_xld_param (: : DistanceTransformID, GenParamName : GenParamValue)

Get the parameters used to build an XLD distance transform.

get_distance_transform_xld_param returns the parameters used to build the XLD distance transform DistanceTransformID. The names of the parameters are passed in GenParamName and their values are returned in GenParamValue. GenParamName can contain the names of parameters *'mode'* and *'max_distance'*.

Parameters

▷ **DistanceTransformID** (input_control) ..... xld_dist_trans ⇝ *handle*
Handle of the XLD distance transform.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters.
Default: 'mode'
List of values: GenParamName ∈ {'mode', 'max_distance'}

▷ **GenParamValue** (output_control) . . . . . attribute.value(-array) ⇝ *string* / real
Values of the generic parameters.
Number of elements: GenParamValue == 1 || GenParamValue == 2
List of values: GenParamValue ∈ {'point_to_point', 'point_to_segment', 5.0}

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_distance_transform_xld, deserialize_distance_transform_xld

Possible Successors

set_distance_transform_xld_param

See also

get_distance_transform_xld_contour, create_distance_transform_xld, apply_distance_transform_xld, clear_distance_transform_xld, write_distance_transform_xld, serialize_distance_transform_xld

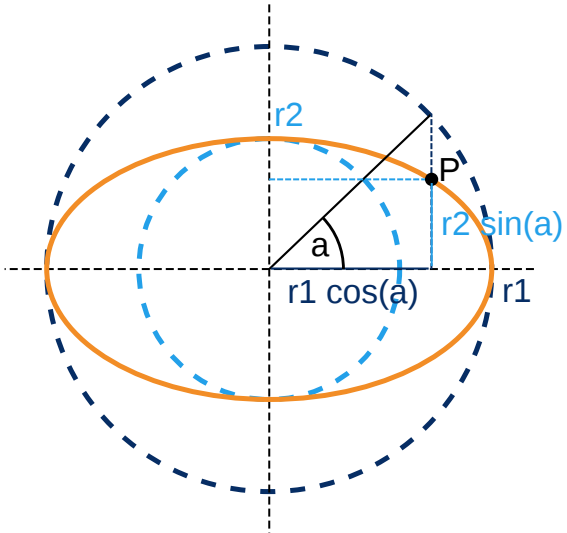
Module

Foundation

get_points_ellipse (: : Angle, Row, Column, Phi, Radius1, Radius2 : RowPoint, ColPoint)

Calculate points on the perimeter of an ellipse.

get_points_ellipse returns the point (RowPoint,ColPoint) on the specified ellipse corresponding to the angle in Angle. With the parameter Angle you are setting the eccentric anomaly, which denotes the angle used for the parametric equation (see the figure below) and refers to the main axis of the ellipse. The ellipse itself is characterized by the center (Row, Column), the orientation of the main axis Phi relative to the horizontal axis, the length of the larger (Radius1) and the smaller half axis (Radius2). The angles are measured counter clockwise in radiant.



The point P with its coordinates RowPoint and ColPoint belongs to the ellipse (orange). It is determined over the angle a and the lengths of the half axis r1 (Radius1) and r2 (Radius2). For simplicity of the visualization and angle Phi=*0* was used.

Parameters

▷ **Angle** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Angle corresponding to the resulting point [rad].
Default: 0
Restriction: Angle >= 0 && Angle <= 6.283185307

▷ **Row** (input_control) . . . . . ellipse.center.y ⇝ *real*
Row coordinate of the center of the ellipse.

▷ **Column** (input_control) . . . . . ellipse.center.x ⇝ *real*
Column coordinate of the center of the ellipse.

▷ **Phi** (input_control) . . . . . ellipse.angle.rad ⇝ *real*
Orientation of the main axis [rad].
Restriction: Phi >= 0 && Phi <= 6.283185307

▷ **Radius1** (input_control) . . . . . ellipse.radius1 ⇝ *real*
Length of the larger half axis.
Restriction: Radius1 > 0

▷ **Radius2** (input_control) . . . . . ellipse.radius2 ⇝ *real*
Length of the smaller half axis.
Restriction: Radius2 >= 0

▷ **RowPoint** (output_control) . . . . . point.y(-array) ⇝ *real*
Row coordinate of the point on the ellipse.

▷ **ColPoint** (output_control) . . . . . point.x(-array) ⇝ *real*
Column coordinates of the point on the ellipse.

Example

```
draw_ellipse(WindowHandle,Row,Column,Phi,Radius1,Radius2)
get_points_ellipse([0,3.14],Row,Column,Phi,Radius1,Radius2,\
                   RowPoint,ColPoint)
```

Result

get_points_ellipse returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fit_ellipse_contour_xld, draw_ellipse, gen_ellipse_contour_xld

See also

gen_ellipse_contour_xld

Module

Foundation

intersection_circle_contour_xld (Contour : : CircleRow, CircleColumn, CircleRadius, CircleStartPhi, CircleEndPhi, CirclePointOrder : Row, Column)

Calculate the intersection points of a circle or circular arc and an XLD contour

intersection_circle_contour_xld calculates the intersection points of a circle or circular arc and the XLD Contour. The circle is defined by its center (CircleRow,CircleColumn) and its radius CircleRadius. In addition to that, a circular arc is characterized by the angle of the start point CircleStartPhi, the angle of the end point CircleEndPhi, and the point order CirclePointOrder along the boundary. If CirclePointOrder is set to *'positive'*, the circular arc is defined counterclockwise. If CirclePointOrder is set to *'negative'*, the circular arc is defined clockwise. The intersection points, if any, are returned in (Row,Column).

Parameters

- ▷ **Contour** (input_object) ..... xld_cont ⇝ *object*
  XLD contour.
- ▷ **CircleRow** (input_control) ..... circle.center.y ⇝ *real* / integer
  Row coordinate of the center of the circle or circular arc.
- ▷ **CircleColumn** (input_control) ..... circle.center.x ⇝ *real* / integer
  Column coordinate of the center of the circle or circular arc.
- ▷ **CircleRadius** (input_control) ..... circle.radius ⇝ *real* / integer
  Radius of the circle or circular arc.
- ▷ **CircleStartPhi** (input_control) ..... angle.rad ⇝ *real*
  Angle of the start point of the circle or circular arc [rad].
  Default: 0.0
- ▷ **CircleEndPhi** (input_control) ..... angle.rad ⇝ *real*
  Angle of the end point of the circle or circular arc [rad].
  Default: 6.28318
- ▷ **CirclePointOrder** (input_control) ..... string ⇝ *string*
  Point order along the circle or circular arc.
  Default: 'positive'
  List of values: CirclePointOrder ∈ {'positive', 'negative'}
- ▷ **Row** (output_control) ..... point.y(-array) ⇝ *real*
  Row coordinates of the intersection points.
- ▷ **Column** (output_control) ..... point.x(-array) ⇝ *real*
  Column coordinates of the intersection points.

Result

If the parameters are valid, the operator intersection_circle_contour_xld returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

intersection_segment_circle

See also

intersection_line_circle

Module

Foundation

intersection_circles (: : Circle1Row, Circle1Column, Circle1Radius, Circle1StartPhi, Circle1EndPhi, Circle1PointOrder, Circle2Row, Circle2Column, Circle2Radius, Circle2StartPhi, Circle2EndPhi, Circle2PointOrder : Row, Column, IsOverlapping)

Calculate the intersection points of two circles or circular arcs

intersection_circles calculates the intersection points of two circles or circular arcs. The points, if any, are returned in (Row,Column). The circles are defined by their center (Circle1Row,Circle1Column), and (Circle2Row,Circle2Column) respectively, and their radii Circle1Radius, and Circle2Radius respectively. In addition to that, a circular arc is characterized by the angle of the start point (Circle1StartPhi,Circle2StartPhi), the angle of the end point (Circle1EndPhi,Circle2EndPhi), and the point order (Circle1PointOrder,Circle2PointOrder) along the boundary. If Circle1PointOrder is set to *'positive'*, the circular arc is defined counterclockwise. If Circle1PointOrder is set to *'negative'*, the circular arc is defined clockwise. The same applies for Circle2PointOrder. If both circles or circular arcs have a part in common IsOverlapping returns the value 1, otherwise 0 is returned. In this case the endpoints of the mutual arc are returned in (Row,Column).

Parameters

- ▷ **Circle1Row** (input_control) ..... circle.center.y ⇝ *real* / integer
  Row coordinate of the center of the first circle or circular arc.
- ▷ **Circle1Column** (input_control) ..... circle.center.x ⇝ *real* / integer
  Column coordinate of the center of the first circle or circular arc.
- ▷ **Circle1Radius** (input_control) ..... circle.radius ⇝ *real* / integer
  Radius of the first circle or circular arc.
- ▷ **Circle1StartPhi** (input_control) ..... angle.rad ⇝ *real*
  Angle of the start point of the first circle or circular arc [rad].
  Default: 0.0
- ▷ **Circle1EndPhi** (input_control) ..... angle.rad ⇝ *real*
  Angle of the end point of the first circle or circular arc [rad].
  Default: 6.28318
- ▷ **Circle1PointOrder** (input_control) ..... string ⇝ *string*
  Point order along the first circle or circular arc.
  Default: 'positive'
  List of values: Circle1PointOrder ∈ {'positive', 'negative'}
- ▷ **Circle2Row** (input_control) ..... circle.center.y ⇝ *real* / integer
  Row coordinate of the center of the second circle or circular arc.
- ▷ **Circle2Column** (input_control) ..... circle.center.x ⇝ *real* / integer
  Column coordinate of the center of the second circle or circular arc.
- ▷ **Circle2Radius** (input_control) ..... circle.radius ⇝ *real* / integer
  Radius of the second circle or circular arc.
- ▷ **Circle2StartPhi** (input_control) ..... angle.rad ⇝ *real*
  Angle of the start point of the second circle or circular arc [rad].
  Default: 0.0
- ▷ **Circle2EndPhi** (input_control) ..... angle.rad ⇝ *real*
  Angle of the end point of the second circle or circular arc [rad].
  Default: 6.28318
- ▷ **Circle2PointOrder** (input_control) ..... string ⇝ *string*
  Point order along the second circle or circular arc.
  Default: 'positive'
  List of values: Circle2PointOrder ∈ {'positive', 'negative'}
- ▷ **Row** (output_control) ..... point.y(-array) ⇝ *real*
  Row coordinates of the intersection points.
- ▷ **Column** (output_control) ..... point.x(-array) ⇝ *real*
  Column coordinates of the intersection points.
- ▷ **IsOverlapping** (output_control) ..... integer ⇝ *integer*
  Do both circles or circular arcs have a part in common?

Result

If the parameters are valid, the operator intersection_circles returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

intersection_contours_xld (Contour1, Contour2 : : IntersectionType : Row, Column, IsOverlapping)

Calculate the intersection points of two XLD contours

intersection_contours_xld calculates the intersection points of the XLD Contour1 and XLD Contour2 which, if any, are returned in (Row,Column). The value in IntersectionType defines the way to calculate the intersections points. By setting IntersectionType = *'self'*, only the self intersections within both contours are returned, i.e., the intersections within Contour1 and XLD Contour2. For IntersectionType = *'mutual'*, only the intersections between both contours are taken into account. The default value is IntersectionType = *'all'*. In this case both, the self and the mutual intersections are returned in Row,Column). If parts of the contours overlap in more than one point IsOverlapping returns the value 1, otherwise 0 is returned. IsOverlapping is set with regard to both the self and mutual overlap, regardless of the setting in IntersectionType. In case of a mutual overlap, the endpoints of the mutual segment are returned in (Row,Column).

Parameters

▷ **Contour1** (input_object) ..... xld_cont ⇝ *object*
First XLD contour.

▷ **Contour2** (input_object) ..... xld_cont ⇝ *object*
Second XLD contour.

▷ **IntersectionType** (input_control) ..... string ⇝ *string*
Intersection points to be returned.
Default: 'all'
List of values: IntersectionType ∈ {'all', 'self', 'mutual'}

▷ **Row** (output_control) ..... point.y(-array) ⇝ *real*
Row coordinates of the intersection points.

▷ **Column** (output_control) ..... point.x(-array) ⇝ *real*
Column coordinates of the intersection points.

▷ **IsOverlapping** (output_control) ..... integer ⇝ *integer*
Does a part of a contour lie above another contour part?

Result

If the parameters are valid, the operator intersection_contours_xld returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

intersection_segments, intersection_segment_contour_xld

See also

intersection_segment_line, intersection_lines, intersection_line_contour_xld

Module

Foundation

intersection_line_circle (: : LineRow1, LineColumn1, LineRow2, LineColumn2, CircleRow, CircleColumn, CircleRadius, CircleStartPhi, CircleEndPhi, CirclePointOrder : Row, Column)

Calculate the intersection points of a line and a circle or circular arc

intersection_line_circle calculates the intersection points of a line and a circle or circular arc. The line is defined by the points (LineRow1,LineColumn1) and (LineRow2,LineColumn2). The circle is defined by its center (CircleRow,CircleColumn) and its radius CircleRadius. In addition to that, a circular arc is characterized by the angle of the start point CircleStartPhi, the angle of the end point CircleEndPhi, and the point order CirclePointOrder along the boundary. If CirclePointOrder is set to *'positive'*, the circular arc is defined counterclockwise. If CirclePointOrder is set to *'negative'*, the circular arc is defined clockwise. The intersection points, if any, are returned in (Row,Column).

Parameters

- ▷ **LineRow1** (input_control) ..... point.y ⇝ *real* / integer
  Row coordinate of the first point of the line.
- ▷ **LineColumn1** (input_control) ..... point.x ⇝ *real* / integer
  Column coordinate of the first point of the line.
- ▷ **LineRow2** (input_control) ..... point.y ⇝ *real* / integer
  Row coordinate of the second point of the line.
- ▷ **LineColumn2** (input_control) ..... point.x ⇝ *real* / integer
  Column coordinate of the second point of the line.
- ▷ **CircleRow** (input_control) ..... circle.center.y ⇝ *real* / integer
  Row coordinate of the center of the circle or circular arc.
- ▷ **CircleColumn** (input_control) ..... circle.center.x ⇝ *real* / integer
  Column coordinate of the center of the circle or circular arc.
- ▷ **CircleRadius** (input_control) ..... circle.radius ⇝ *real* / integer
  Radius of the circle or circular arc.
- ▷ **CircleStartPhi** (input_control) ..... angle.rad ⇝ *real*
  Angle of the start point of the circle or circular arc [rad].
  Default: 0.0
- ▷ **CircleEndPhi** (input_control) ..... angle.rad ⇝ *real*
  Angle of the end point of the circle or circular arc [rad].
  Default: 6.28318
- ▷ **CirclePointOrder** (input_control) ..... string ⇝ *string*
  Point order along the circle or circular arc.
  Default: 'positive'
  List of values: CirclePointOrder ∈ {'positive', 'negative'}
- ▷ **Row** (output_control) ..... point.y(-array) ⇝ *real*
  Row coordinates of the intersection points.
- ▷ **Column** (output_control) ..... point.x(-array) ⇝ *real*
  Column coordinates of the intersection points.

Result

If the parameters are valid, the operator intersection_line_circle returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

intersection_segment_circle, intersection_circle_contour_xld

Module

Foundation

intersection_line_contour_xld (Contour : : LineRow1, LineColumn1, LineRow2, LineColumn2 : Row, Column, IsOverlapping)

Calculate the intersection points of a line and an XLD contour

intersection_line_contour_xld calculates the intersection points of a line and an XLD Contour. The line is defined by the points (LineRow1,LineColumn1) and (LineRow2,LineColumn2). The intersection points, if any, are returned in (Row,Column). If a part of the XLD contour lies on the line IsOverlapping returns the value 1, otherwise 0 is returned. In this case the endpoints of the XLD segment are returned in (Row,Column).

Parameters

- ▷ **Contour** (input_object) ..... xld_cont ⇝ *object*
  XLD contour.
- ▷ **LineRow1** (input_control) ..... point.y ⇝ *real* / integer
  Row coordinate of the first point of the line.
- ▷ **LineColumn1** (input_control) ..... point.x ⇝ *real* / integer
  Column coordinate of the first point of the line.
- ▷ **LineRow2** (input_control) ..... point.y ⇝ *real* / integer
  Row coordinate of the second point of the line.
- ▷ **LineColumn2** (input_control) ..... point.x ⇝ *real* / integer
  Column coordinate of the second point of the line.
- ▷ **Row** (output_control) ..... point.y(-array) ⇝ *real*
  Row coordinates of the intersection points.
- ▷ **Column** (output_control) ..... point.x(-array) ⇝ *real*
  Column coordinates of the intersection points.
- ▷ **IsOverlapping** (output_control) ..... integer ⇝ *integer*
  Does a part of the XLD contour lie on the line?

Result

If the parameters are valid, the operator intersection_line_contour_xld returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

intersection_segment_line

See also

intersection_segments, intersection_lines, intersection_segment_contour_xld, intersection_contours_xld

Module

Foundation

intersection_lines (: : Line1Row1, Line1Column1, Line1Row2, Line1Column2, Line2Row1, Line2Column1, Line2Row2, Line2Column2 : Row, Column, IsOverlapping)

Calculate the intersection point of two lines

intersection_lines calculates the intersection point of two lines, which are defined by two of their points (Line1Row1,Line1Column1), (Line1Row2,Line1Column2), and (Line2Row1,Line2Column1), (Line2Row2,Line2Column2) respectively. The intersection point, if it exists, is returned in (Row,Column). If both lines are identical, IsOverlapping returns the value 1, otherwise 0 is returned. In this case no intersection point is returned in (Row,Column).

Parameters

▷ **Line1Row1** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the first point of the first line.

▷ **Line1Column1** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the first point of the first line.

▷ **Line1Row2** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the second point of the first line.

▷ **Line1Column2** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the second point of the first line.

▷ **Line2Row1** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the first point of the second line.

▷ **Line2Column1** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the first point of the second line.

▷ **Line2Row2** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the second point of the second line.

▷ **Line2Column2** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the second point of the second line.

▷ **Row** (output_control) . . . . . point.y ⇝ *real*
Row coordinate of the intersection point.

▷ **Column** (output_control) . . . . . point.x ⇝ *real*
Column coordinate of the intersection point.

▷ **IsOverlapping** (output_control) . . . . . integer ⇝ *integer*
Are both lines identical?

Result

If the parameters are valid, the operator `intersection_lines` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`intersection_segments`, `intersection_segment_line`, `intersection_segment_contour_xld`, `intersection_line_contour_xld`, `intersection_contours_xld`

Module

Foundation

intersection_segment_circle (: : SegmentRow1, SegmentColumn1, SegmentRow2, SegmentColumn2, CircleRow, CircleColumn, CircleRadius, CircleStartPhi, CircleEndPhi, CirclePointOrder : Row, Column)

Calculate the intersection points of a segment and a circle or circular arc

`intersection_segment_circle` calculates the intersection point of a segment and a circle or circular arc. The segment is defined by its endpoints (`SegmentRow1`,`SegmentColumn1`) and (`SegmentRow2`,`SegmentColumn2`). The circle is defined by its center (`CircleRow`,`CircleColumn`) and its radius `CircleRadius`. In addition to that, a circular arc is characterized by the angle of the start point `CircleStartPhi`, the angle of the end point `CircleEndPhi`, and the point order `CirclePointOrder` along the boundary. If `CirclePointOrder` is set to *'positive'*, the circular arc is defined counterclockwise. If `CirclePointOrder` is set to *'negative'*, the circular arc is defined clockwise. The intersection points, if any, are returned in (`Row`,`Column`).

Parameters

- ▷ **SegmentRow1** (input_control) . . . . . point.y ⇝ *real* / integer
  Row coordinate of the first point of the segment.
- ▷ **SegmentColumn1** (input_control) . . . . . point.x ⇝ *real* / integer
  Column coordinate of the first point of the segment.
- ▷ **SegmentRow2** (input_control) . . . . . point.y ⇝ *real* / integer
  Row coordinate of the second point of the segment.
- ▷ **SegmentColumn2** (input_control) . . . . . point.x ⇝ *real* / integer
  Column coordinate of the second point of the segment.
- ▷ **CircleRow** (input_control) . . . . . circle.center.y ⇝ *real* / integer
  Row coordinate of the center of the circle or circular arc.
- ▷ **CircleColumn** (input_control) . . . . . circle.center.x ⇝ *real* / integer
  Column coordinate of the center of the circle or circular arc.
- ▷ **CircleRadius** (input_control) . . . . . circle.radius ⇝ *real* / integer
  Radius of the circle or circular arc.
- ▷ **CircleStartPhi** (input_control) . . . . . angle.rad ⇝ *real*
  Angle of the start point of the circle or circular arc [rad].
  Default: 0.0
- ▷ **CircleEndPhi** (input_control) . . . . . angle.rad ⇝ *real*
  Angle of the end point of the circle or circular arc [rad].
  Default: 6.28318
- ▷ **CirclePointOrder** (input_control) . . . . . string ⇝ *string*
  Point order along the circle or circular arc.
  Default: 'positive'
  List of values: `CirclePointOrder` ∈ {'positive', 'negative'}
- ▷ **Row** (output_control) . . . . . point.y(-array) ⇝ *real*
  Row coordinates of the intersection points.
- ▷ **Column** (output_control) . . . . . point.x(-array) ⇝ *real*
  Column coordinates of the intersection points.

Result

If the parameters are valid, the operator `intersection_segment_circle` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`intersection_circle_contour_xld`

See also

`intersection_line_circle`

Module

Foundation

intersection_segment_contour_xld (Contour : : SegmentRow1, SegmentColumn1, SegmentRow2, SegmentColumn2 : Row, Column, IsOverlapping)

Calculate the intersection points of a segment and an XLD contour

`intersection_segment_contour_xld` calculates the intersection points of a segment and an XLD `Contour`. The segment is defined by its endpoints (`SegmentRow1`,`SegmentColumn1`) and (`SegmentRow2`,`SegmentColumn2`). The intersection points, if any, are returned in (`Row`,`Column`). If the

segment and the XLD contour have a part in common IsOverlapping returns the value 1, otherwise 0 is returned. In this case the endpoints of the mutual segment are returned in (Row,Column).

Parameters

- ▷ **Contour** (input_object) ..... xld_cont ⇝ *object*
  XLD contour.
- ▷ **SegmentRow1** (input_control) ..... point.y ⇝ *real* / integer
  Row coordinate of the first point of the segment.
- ▷ **SegmentColumn1** (input_control) ..... point.x ⇝ *real* / integer
  Column coordinate of the first point of the segment.
- ▷ **SegmentRow2** (input_control) ..... point.y ⇝ *real* / integer
  Row coordinate of the second point of the segment.
- ▷ **SegmentColumn2** (input_control) ..... point.x ⇝ *real* / integer
  Column coordinate of the second point of the segment.
- ▷ **Row** (output_control) ..... point.y(-array) ⇝ *real*
  Row coordinates of the intersection points.
- ▷ **Column** (output_control) ..... point.x(-array) ⇝ *real*
  Column coordinates of the intersection points.
- ▷ **IsOverlapping** (output_control) ..... integer ⇝ *integer*
  Do the segment and the XLD contour have a part in common?

Result

If the parameters are valid, the operator intersection_segment_contour_xld returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

intersection_segments, intersection_contours_xld

See also

intersection_segment_line, intersection_lines, intersection_line_contour_xld

Module

Foundation

intersection_segment_line (: : SegmentRow1, SegmentColumn1, SegmentRow2, SegmentColumn2, LineRow1, LineColumn1, LineRow2, LineColumn2 : Row, Column, IsOverlapping)

Calculate the intersection point of a segment and a line

intersection_segment_line calculates the intersection point of a segment and a line. The segment is defined by its endpoints (SegmentRow1,SegmentColumn1) and (SegmentRow2,SegmentColumn2). The line is defined by the two points (LineRow1,LineColumn1) and (LineRow2,LineColumn2). The intersection point, if it exists, is returned in (Row,Column). If the segment and the line have a part in common, IsOverlapping returns the value 1, otherwise 0 is returned. In this case the endpoints of the mutual segment are returned in (Row,Column).

Parameters

- ▷ **SegmentRow1** (input_control) ..... point.y ⇝ *real* / integer
  Row coordinate of the first point of the segment.
- ▷ **SegmentColumn1** (input_control) ..... point.x ⇝ *real* / integer
  Column coordinate of the first point of the segment.

▷ **SegmentRow2** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the second point of the segment.

▷ **SegmentColumn2** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the second point of the segment.

▷ **LineRow1** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the first point of the line.

▷ **LineColumn1** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the first point of the line.

▷ **LineRow2** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the second point of the line.

▷ **LineColumn2** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the second point of the line.

▷ **Row** (output_control) . . . . . point.y(-array) ⇝ *real*
Row coordinate of the intersection point.

▷ **Column** (output_control) . . . . . point.x(-array) ⇝ *real*
Column coordinate of the intersection point.

▷ **IsOverlapping** (output_control) . . . . . integer ⇝ *integer*
Do the segment and the line have a part in common?

Result

If the parameters are valid, the operator `intersection_segment_line` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`intersection_line_contour_xld`

See also

`intersection_segments`, `intersection_lines`, `intersection_segment_contour_xld`, `intersection_contours_xld`

Module

Foundation

intersection_segments (: : Segment1Row1, Segment1Column1, Segment1Row2, Segment1Column2, Segment2Row1, Segment2Column1, Segment2Row2, Segment2Column2 : Row, Column, IsOverlapping)

Calculate the intersection point of two line segments

`intersection_segments` calculates the intersection point of two line segments, which are defined by their endpoints (`Segment1Row1`,`Segment1Column1`), (`Segment1Row2`,`Segment1Column2`), and (`Segment2Row1`,`Segment2Column1`), (`Segment2Row2`,`Segment2Column2`) respectively. The intersection point, if it exists, is returned in (`Row`,`Column`). If both segments have a part in common, `IsOverlapping` returns the value 1, otherwise 0 is returned. In this case the endpoints of the mutual segment are returned in (`Row`,`Column`).

Parameters

▷ **Segment1Row1** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the first point of the first segment.

▷ **Segment1Column1** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the first point of the first segment.

▷ **Segment1Row2** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the second point of the first segment.

▷ **Segment1Column2** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the second point of the first segment.

▷ **Segment2Row1** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the first point of the second segment.

▷ **Segment2Column1** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the first point of the second segment.

▷ **Segment2Row2** (input_control) . . . . . point.y ⇝ *real* / integer
Row coordinate of the second point of the second segment.

▷ **Segment2Column2** (input_control) . . . . . point.x ⇝ *real* / integer
Column coordinate of the second point of the second segment.

▷ **Row** (output_control) . . . . . point.y(-array) ⇝ *real*
Row coordinate of the intersection point.

▷ **Column** (output_control) . . . . . point.x(-array) ⇝ *real*
Column coordinate of the intersection point.

▷ **IsOverlapping** (output_control) . . . . . integer ⇝ *integer*
Do both segments have a part in common?

Result

If the parameters are valid, the operator `intersection_segments` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`intersection_segment_contour_xld`, `intersection_contours_xld`

See also

`intersection_segment_line`, `intersection_lines`, `intersection_line_contour_xld`

Module

Foundation

pluecker_line_to_point_direction (: : LineDirectionX, LineDirectionY, LineDirectionZ, LineMomentX, LineMomentY, LineMomentZ : PointX, PointY, PointZ, DirectionX, DirectionY, DirectionZ)

Convert a 3D line given by Plücker coordinates to a 3D line given by a point and a direction.

`pluecker_line_to_point_direction` converts a 3D line given by Plücker coordinates to a line given by a point on the line and the direction of the line. The line is given in Plücker coordinates $\mathbf{L}$ (`LineDirectionX`, `LineDirectionY`, `LineDirectionZ`) and $\mathbf{M}$ (`LineMomentX`, `LineMomentY`, `LineMomentZ`). The direction is given by (`DirectionX`, `DirectionY`, `DirectionZ`) of the line. and the point on the line by (`PointX`, `PointY`, `PointZ`). For the definition of Plücker coordinates, see "`Solution Guide III-C - 3D Vision`". All input tuples must be of same length.

Let $\mathbf{L}$ denote the line direction (`LineDirectionX`, `LineDirectionY`, `LineDirectionZ`), $\mathbf{M}$ the line moment (`LineMomentX`, `LineMomentY`, `LineMomentZ`), $\mathbf{P}$ the point (`PointX`, `PointY`, `PointZ`) on the line, and $\mathbf{D}$ the direction (`DirectionX`, `DirectionY`, `DirectionZ`) of the line. Then, $\mathbf{P} = \mathbf{L} \times \mathbf{M}$ and $\mathbf{D} = \mathbf{L}$. Note that $\mathbf{P}$ is the point on the line closest to the origin.

Parameters

▷ **LineDirectionX** (input_control) . . . . . point3d.x(-array) ⇝ *real*
X component of the direction vector of the line.

▷ **LineDirectionY** (input_control) . . . . . point3d.y(-array) ⇝ *real*
Y component of the direction vector of the line.

▷ **LineDirectionZ** (input_control) . . . . . point3d.z(-array) ⇝ *real*
Z component of the direction vector of the line.

▷ **LineMomentX** (input_control) . . . . . point3d.x(-array) ⇝ *real*
X component of the moment vector of the line.

▷ **LineMomentY** (input_control) . . . . . point3d.y(-array) ⇝ *real*
Y component of the moment vector of the line.

▷ **LineMomentZ** (input_control) . . . . . point3d.z(-array) ⇝ *real*
Z component of the moment vector of the line.

▷ **PointX** (output_control) . . . . . point3d.x(-array) ⇝ *real*
X coordinate of the first point on the line.

▷ **PointY** (output_control) . . . . . point3d.y(-array) ⇝ *real*
Y coordinate of the first point on the line.

▷ **PointZ** (output_control) . . . . . point3d.z(-array) ⇝ *real*
Z coordinate of the first point on the line.

▷ **DirectionX** (output_control) . . . . . point3d.x(-array) ⇝ *real*
X coordinates of the direction of the line.

▷ **DirectionY** (output_control) . . . . . point3d.y(-array) ⇝ *real*
Y coordinates of the direction of the line.

▷ **DirectionZ** (output_control) . . . . . point3d.z(-array) ⇝ *real*
Z coordinates of the direction of the line.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

pluecker_line_to_points

See also

point_direction_to_pluecker_line

Module

Foundation

pluecker_line_to_points (: : LineDirectionX, LineDirectionY, LineDirectionZ, LineMomentX, LineMomentY, LineMomentZ : Point1X, Point1Y, Point1Z, Point2X, Point2Y, Point2Z)

Convert a 3D line given by Plücker coordinates to a 3D line given by two points.

pluecker_line_to_points converts a 3D line given by Plücker coordinates $\mathbf{L}$ (LineDirectionX, LineDirectionY, LineDirectionZ) and $\mathbf{M}$ (LineMomentX, LineMomentY, LineMomentZ). to a line given by two points on the line (Point1X, Point1Y, Point1Z) and (Point2X, Point2Y, Point2Z). For the definition of Plücker coordinates, see "Solution Guide III-C - 3D Vision". All input tuples must be of same length.

Let $\mathbf{L}$ denote the line direction (LineDirectionX, LineDirectionY, LineDirectionZ), $\mathbf{M}$ the line moment (LineMomentX, LineMomentY, LineMomentZ), $\mathbf{P}$ the first point (Point1X, Point1Y, Point1Z) on the line, and $\mathbf{Q}$ the second point (Point2X, Point2Y, Point2Z) on the line. Then, $\mathbf{P} = \mathbf{L} \times \mathbf{M}$ and $\mathbf{Q} = \mathbf{P} + \mathbf{L}$. Note that $\mathbf{P}$ is the point on the line closest to the origin.

Parameters

▷ **LineDirectionX** (input_control) . . . . . point3d.x(-array) ⇝ *real*
X component of the direction vector of the line.

▷ **LineDirectionY** (input_control) . . . . . point3d.y(-array) ⇝ *real*
Y component of the direction vector of the line.

▷ **LineDirectionZ** (input_control) . . . . . point3d.z(-array) ⇝ *real*
Z component of the direction vector of the line.

▷ **LineMomentX** (input_control) . . . . . point3d.x(-array) ⇝ *real*
X component of the moment vector of the line.

▷ **LineMomentY** (input_control) . . . . . point3d.y(-array) ⇝ *real*
Y component of the moment vector of the line.

▷ **LineMomentZ** (input_control) . . . . . point3d.z(-array) ⇝ *real*
Z component of the moment vector of the line.

▷ **Point1X** (output_control) . . . . . point3d.x(-array) ⇝ *real*
X coordinate of the first point on the line.

▷ **Point1Y** (output_control) . . . . . point3d.y(-array) ⇝ *real*
Y coordinate of the first point on the line.

▷ **Point1Z** (output_control) . . . . . point3d.z(-array) ⇝ *real*
Z coordinate of the first point on the line.

▷ **Point2X** (output_control) . . . . . point3d.x(-array) ⇝ *real*
X coordinate of the second point on the line.

▷ **Point2Y** (output_control) . . . . . point3d.y(-array) ⇝ *real*
Y coordinate of the second point on the line.

▷ **Point2Z** (output_control) . . . . . point3d.z(-array) ⇝ *real*
Z coordinate of the second point on the line.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

distance_point_line

Alternatives

pluecker_line_to_point_direction

See also

points_to_pluecker_line

Module

Foundation

point_direction_to_pluecker_line (: : PointX, PointY, PointZ, DirectionX, DirectionY, DirectionZ : LineDirectionX, LineDirectionY, LineDirectionZ, LineMomentX, LineMomentY, LineMomentZ)

Convert a 3D line given by a point and a direction to Plücker coordinates.

point_direction_to_pluecker_line converts the 3D line given by the point (PointX, PointY, PointZ) and the direction (DirectionX, DirectionY, DirectionZ) to Plücker coordinates $\mathbf{L}$ (LineDirectionX, LineDirectionY, LineDirectionZ) and $\mathbf{M}$ (LineMomentX, LineMomentY, LineMomentZ). For the definition of Plücker coordinates, see "Solution Guide III-C - 3D Vision". All input tuples must be of same length.

Let $\mathbf{P}$ denote the point (PointX, PointY, PointZ) and $\mathbf{D}$ the direction (DirectionX, DirectionY, DirectionZ) of the line. To compute the line direction $\mathbf{L}$, the length $\|\mathbf{D}\|$ of the vector $\mathbf{D}$ is computed. If $\|\mathbf{D}\| = 0$, the direction $\mathbf{D}$ does not define a line and an error is returned. The further calculations to Plücker coordinates are given in points_to_pluecker_line.

Parameters

▷ **PointX** (input_control) . . . . . point3d.x(-array) ⇝ *real*
X coordinates of the point on the line.

▷ **PointY** (input_control) . . . . . point3d.y(-array) ⇝ *real*
Y coordinates of the point on the line.

▷ **PointZ** (input_control) . . . . . point3d.z(-array) ⇝ *real*
Z coordinates of the point on the line.

▷ **DirectionX** (input_control) . . . . . point3d.x(-array) ⇝ *real*
X coordinates of the direction of the line.

▷ **DirectionY** (input_control) . . . . . point3d.y(-array) ⇝ *real*
Y coordinates of the direction of the line.

▷ **DirectionZ** (input_control) . . . . . point3d.z(-array) ⇝ *real*
Z coordinates of the direction of the line.

▷ **LineDirectionX** (output_control) . . . . . point3d.x(-array) ⇝ *real*
X component of the direction vector of the line.

▷ **LineDirectionY** (output_control) . . . . . point3d.y(-array) ⇝ *real*
Y component of the direction vector of the line.

▷ **LineDirectionZ** (output_control) . . . . . point3d.z(-array) ⇝ *real*
Z component of the direction vector of the line.

▷ **LineMomentX** (output_control) . . . . . point3d.x(-array) ⇝ *real*
X component of the moment vector of the line.

▷ **LineMomentY** (output_control) . . . . . point3d.y(-array) ⇝ *real*
Y component of the moment vector of the line.

▷ **LineMomentZ** (output_control) . . . . . point3d.z(-array) ⇝ *real*
Z component of the moment vector of the line.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

distance_point_pluecker_line, point_pluecker_line_to_hom_mat3d

Alternatives

points_to_pluecker_line

See also

pluecker_line_to_point_direction

Module

Foundation

points_to_pluecker_line (: : Point1X, Point1Y, Point1Z, Point2X, Point2Y, Point2Z : LineDirectionX, LineDirectionY, LineDirectionZ, LineMomentX, LineMomentY, LineMomentZ)

Convert a 3D line given by two points to Plücker coordinates.

points_to_pluecker_line converts the 3D line given by the two points (Point1X, Point1Y, Point1Z) and (Point2X, Point2Y, Point2Z) to Plücker coordinates $\mathbf{L}$ = (LineDirectionX, LineDirectionY, LineDirectionZ) and $\mathbf{M}$ = (LineMomentX, LineMomentY, LineMomentZ). For the definition of Plücker coordinates, see "Solution Guide III-C - 3D Vision". All input tuples must be of same length.

Let $\mathbf{P}$ denote the first point (Point1X, Point1Y, Point1Z) and $\mathbf{Q}$ the second point (Point2X, Point2Y, Point2Z) on the line. To compute the line direction $\mathbf{L}$, the vector $\mathbf{D} = \mathbf{Q} - \mathbf{P}$ and its length $\|\mathbf{D}\|$ are computed.

If $\|\mathbf{D}\| = 0$, the points do not define a line and an error is returned. The line direction is given by $\mathbf{L} = \mathbf{D}/\|\mathbf{D}\|$. The line moment is given by the cross product $\mathbf{M} = \mathbf{P} \times \mathbf{L}$.

Parameters

- ▷ **Point1X** (input_control) ..... point3d.x(-array) ⇝ *real*
  X coordinates of the first point on the line.
- ▷ **Point1Y** (input_control) ..... point3d.y(-array) ⇝ *real*
  Y coordinates of the first point on the line.
- ▷ **Point1Z** (input_control) ..... point3d.z(-array) ⇝ *real*
  Z coordinates of the first point on the line.
- ▷ **Point2X** (input_control) ..... point3d.x(-array) ⇝ *real*
  X coordinates of the second point on the line.
- ▷ **Point2Y** (input_control) ..... point3d.y(-array) ⇝ *real*
  Y coordinates of the second point on the line.
- ▷ **Point2Z** (input_control) ..... point3d.z(-array) ⇝ *real*
  Z coordinates of the second point on the line.
- ▷ **LineDirectionX** (output_control) ..... point3d.x(-array) ⇝ *real*
  X component of the direction vector of the line.
- ▷ **LineDirectionY** (output_control) ..... point3d.y(-array) ⇝ *real*
  Y component of the direction vector of the line.
- ▷ **LineDirectionZ** (output_control) ..... point3d.z(-array) ⇝ *real*
  Z component of the direction vector of the line.
- ▷ **LineMomentX** (output_control) ..... point3d.x(-array) ⇝ *real*
  X component of the moment vector of the line.
- ▷ **LineMomentY** (output_control) ..... point3d.y(-array) ⇝ *real*
  Y component of the moment vector of the line.
- ▷ **LineMomentZ** (output_control) ..... point3d.z(-array) ⇝ *real*
  Z component of the moment vector of the line.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_line_of_sight`

Possible Successors

`distance_point_pluecker_line`, `point_pluecker_line_to_hom_mat3d`

Alternatives

`point_direction_to_pluecker_line`

See also

`pluecker_line_to_points`

Module

Foundation

projection_pl (: : Row, Column, Row1, Column1, Row2, Column2 : RowProj, ColProj)

Calculate the projection of a point onto a line.

The operator `projection_pl` calculates the projection of a point (`Row`,`Column`) onto a line which is represented by the two points (`Row1`,`Column1`) and (`Row2`,`Column2`). The coordinates of the projected point are returned in `RowProj` and `ColProj`.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of projections is always *N*.

Parameters

▷ **Row** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinate of the point.

▷ **Column** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinate of the point.

▷ **Row1** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinate of the first point on the line.

▷ **Column1** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinate of the first point on the line.

▷ **Row2** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinate of the second point on the line.

▷ **Column2** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinate of the second point on the line.

▷ **RowProj** (output_control) ..... real(-array) ⇝ *real*
Row coordinate of the projected point.

▷ **ColProj** (output_control) ..... real(-array) ⇝ *real*
Column coordinate of the projected point

Example

```
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
Row1 := 300
Column1 := 200
Row2 := 140
Column2 := 400
Rows := 300
Columns := 170
dev_set_color ('cadet blue')
gen_contour_polygon_xld (Contour, [Row1,Row2], [Column1,Column2])
Offset := 0
for Rows := 40 to 280 by 40
  dev_set_color ('red')
  gen_cross_contour_xld (Point,Rows+Offset, Columns, 6, 0)
  projection_pl (Rows+Offset, Columns, Row1, Column1, Row2, Column2, \
                 RowProj, ColProj)
  dev_set_color ('blue')
  gen_cross_contour_xld (RowP, RowProj, ColProj, 6, 0)
  Offset := Offset+30
endfor
```

Result

`projection_pl` returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Module

Foundation

read_distance_transform_xld (: : FileName : DistanceTransformID)

Read an XLD distance transform from a file.

read_distance_transform_xld reads an XLD distance transform which previously has been stored with write_distance_transform_xld from the file given by FileName and returns the handle DistanceTransformID. The default HALCON extension for the XLD distance transform is 'hdtc'.

get_distance_transform_xld_contour can be used to get the reference contour that was used to build the XLD distance transform DistanceTransformID.

Parameters

▷ **FileName** (input_control) ..... filename.write ⇝ *string*
Name of the file.
File extension: .hdtc

▷ **DistanceTransformID** (output_control) ..... xld_dist_trans ⇝ *handle*
Handle of the XLD distance transform.

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

apply_distance_transform_xld, get_distance_transform_xld_contour, get_distance_transform_xld_param

See also

write_distance_transform_xld, serialize_distance_transform_xld, deserialize_distance_transform_xld, create_distance_transform_xld, set_distance_transform_xld_param, clear_distance_transform_xld

Module

Foundation

serialize_distance_transform_xld (
: : DistanceTransformID : SerializedItemHandle)

Serialize an XLD distance transform.

serialize_distance_transform_xld serializes the XLD distance transform DistanceTransformID. The serialized XLD distance transform is returned in the handle SerializedItemHandle and can be deserialized with deserialize_distance_transform_xld.

Parameters

▷ **DistanceTransformID** (input_control) ..... xld_dist_trans ⇝ *handle*
Handle of the XLD distance transform.

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized XLD distance transform.

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_distance_transform_xld

Possible Successors

clear_distance_transform_xld

Alternatives

write_distance_transform_xld

See also

deserialize_distance_transform_xld, read_distance_transform_xld, apply_distance_transform_xld, get_distance_transform_xld_contour, get_distance_transform_xld_param, set_distance_transform_xld_param

Module

Foundation

set_distance_transform_xld_param (: : DistanceTransformID, GenParamName, GenParamValue :)

Set new parameters for an XLD distance transform.

set_distance_transform_xld_param sets new parameters for the XLD distance transform DistanceTransformID. The names and values of the parameters are passed in GenParamName and GenParamValue, respectively. GenParamName can contain the names of parameters *'mode'* and *'max_distance'*. The XLD distance transform of the original reference contour is then rebuilt with updated values of parameters.

Parameters

▷ **DistanceTransformID** (input_control) . . . . . xld_dist_trans ⇝ *handle*
Handle of the XLD distance transform.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Names of the generic parameters.
Default: 'mode'
List of values: GenParamName ∈ {'mode', 'max_distance'}

▷ **GenParamValue** (input_control) . . . . . attribute.value(-array) ⇝ *string* / real
Values of the generic parameters.
Default: 'point_to_point'
Suggested values: GenParamValue ∈ {'point_to_point', 'point_to_segment', 5.0}

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DistanceTransformID

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

get_distance_transform_xld_param

Possible Successors

apply_distance_transform_xld

See also

write_distance_transform_xld, serialize_distance_transform_xld,

get_distance_transform_xld_contour, read_distance_transform_xld, deserialize_distance_transform_xld, clear_distance_transform_xld

Module

Foundation

write_distance_transform_xld (: : DistanceTransformID, FileName :)

Write an XLD distance transform into a file.

write_distance_transform_xld writes the XLD distance transform DistanceTransformID into the file given by FileName. The default HALCON extension for the XLD distance transform is 'hdtc'.

The stored XLD distance transform can be read in with read_distance_transform_xld.

Parameters

- ▷ **DistanceTransformID** (input_control) ..... xld_dist_trans ⇝ *handle*
  Handle of the XLD distance transform.
- ▷ **FileName** (input_control) ..... filename.write ⇝ *string*
  Name of the file.
  File extension: .hdtc

Result

If all parameters are correct, the operator returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_distance_transform_xld

Possible Successors

clear_distance_transform_xld

Alternatives

serialize_distance_transform_xld

See also

read_distance_transform_xld, deserialize_distance_transform_xld, get_distance_transform_xld_contour, apply_distance_transform_xld, get_distance_transform_xld_param, set_distance_transform_xld_param

Module

Foundation

26.4 Grid Rectification

connect_grid_points (Image : ConnectingLines : Row, Column, Sigma, MaxDist :)

Establish connections between the grid points of the rectification grid.

connect_grid_points searches for connecting lines between the grid points (Row,Column) of the rectification grid. The connecting lines are extracted from the input image Image by a combination of an edge detector, a smoothing filter, and a line detector, each of size σ. The σ to be used is determined as follows: When a single value is passed in Sigma, this value is used. When a tuple of three values (*'sigma_min'*, *'sigma_max'*, *'sigma_step'*) is passed, connect_grid_points tests every σ within a range from *'sigma_min'* to *'sigma_max'* with a step width of *'sigma_step'* and chooses the σ that causes the greatest number of connecting lines. The same happens

when a tuple of only two values *'sigma_min'* and *'sigma_max'* is passed. However, in this case a fixed step width of *0.05* is used.

Then, the extracted connecting lines are split at the grid points and those line segments are selected that start as well as end at a grid point. Note that edge detectors typically don't work very accurately in the proximity of edge junctions, and thus in general the connecting lines will not hit the grid points. Therefore, actually those connecting lines are split and selected that start at, end at, or pass a grid point at a maximum distance of `MaxDist`. The connecting lines are modified in order to start and end exactly in the corresponding grid points, and are returned in `ConnectingLines` as XLD contours.

Additionally, `connect_grid_points` calculates for each output XLD contour its type of transition and stores it in its global attribute *'bright_dark'*. The attribute is set to *1.0*, if the connecting line forms a bright-dark transition (left to right, viewed from start point to end point), otherwise it is set to *0.0*. For further information about global contour attributes see `get_contour_global_attrib_xld`.

Attention

For a reliable determination of the type of bright-dark transition as well as for the following rectification, it is necessary that each connecting line has at least three contour points. Therefore, connecting lines with only two contour points are not returned. Note, that the parameter `MaxDist` has a substantial influence on the length of the returned connecting lines, because all contour points of a possible connecting line that are closer than `MaxDist` to a grid point are replaced by one single contour point. If `MaxDist` is too big, some of the connecting lines might get lost.

Parameters

- ▷ **`Image`** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
  Input image.
- ▷ **`ConnectingLines`** (output_object) ..... xld-array ⇝ *object*
  Output contours.
- ▷ **`Row`** (input_control) ..... point.y-array ⇝ *real*
  Row coordinates of the grid points.
- ▷ **`Column`** (input_control) ..... point.x-array ⇝ *real*
  Column coordinates of the grid points.
  Restriction: number(Column) == number(Row)
- ▷ **`Sigma`** (input_control) ..... number(-array) ⇝ *integer* / real
  Size of the applied Gaussians.
  Number of elements: 1 <= Sigma && Sigma <= 3
  Default: 0.9
  Suggested values: `Sigma` ∈ {0.7, 0.9, 1.1, 1.3, 1.5}
  Restriction: 0.7 <= Sigma
- ▷ **`MaxDist`** (input_control) ..... number ⇝ *real* / integer
  Maximum distance of the connecting lines from the grid points.
  Default: 5.5
  Suggested values: `MaxDist` ∈ {1.5, 3.5, 5.5, 7.5, 9.5}
  Restriction: 0.0 <= MaxDist

Result

`connect_grid_points` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`saddle_points_sub_pix`

Possible Successors

`gen_grid_rectification_map`

Module

Calibration

create_rectification_grid (: : Width, NumSquares, GridFile :)

Generate a PostScript file, which describes the rectification grid.

create_rectification_grid generates a checkered pattern with NumSquares × NumSquares alternating black and white squares. This pattern is Width meters wide (and high). Around the pattern there is an inner frame of *0.3* times the width of one square, which continues the checkered pattern. The pattern is completed by a solid white outer frame of *0.7* times the width of one square. In the center of the pattern there are two circular marks, one black on a white square and one white on a black square. The radius of both marks correspond to one third of the side length of the respective enclosing square. These marks are used by gen_grid_rectification_map to rotate the detected layout of the grid points into the correct orientation. It is assumed that the black mark is positioned to the left of the white mark, when oriented correctly. The file GridFile contains the PostScript description of the rectification grid.

Parameters

▷ **Width** (input_control) . . . . . real ⇝ *real*
Width of the checkered pattern in meters (without the two frames).
Default: 0.17
Suggested values: Width ∈ {1.2, 0.8, 0.6, 0.4, 0.2, 0.1}
Recommended increment: 0.1
Restriction: 0.0 < Width

▷ **NumSquares** (input_control) . . . . . integer ⇝ *integer*
Number of squares per row and column.
Default: 17
Suggested values: NumSquares ∈ {11, 13, 15, 17, 19, 21, 23, 25, 27}
Recommended increment: 2
Restriction: 2 <= NumSquares

▷ **GridFile** (input_control) . . . . . filename.write ⇝ *string*
File name of the PostScript file.
Default: 'rectification_grid.ps'
File extension: .ps

Result

find_rectification_grid returns 2 (H_MSG_TRUE) if all parameter values are correct and the file has been written successfully. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

find_rectification_grid, saddle_points_sub_pix, connect_grid_points, gen_grid_rectification_map

Module

Foundation

find_rectification_grid (Image : GridRegion : MinContrast, Radius :)

Segment the rectification grid region in the image.

find_rectification_grid searches in the image Image for image parts that contain the rectification grid and returns them in the region GridRegion. To do so, essentially image areas with a contrast of at least MinContrast are extracted and the holes in these areas are filled up. Then, an opening with the radius Radius is applied to these areas to eliminate smaller areas of high contrast.

During grid-rectification, a careful reduction of the input region to those image parts that actually contain the rectification grid is useful for two purposes: First, the computing time can be reduced and secondly, saddle_points_sub_pix and connect_grid_points can be prevented from detecting false grid points and connecting lines.

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
Input image.

▷ **GridRegion** (output_object) ..... region ⇝ *object*
Output region containing the rectification grid.

▷ **MinContrast** (input_control) ..... number ⇝ *real* / integer
Minimum contrast.
Default: 8.0
Suggested values: MinContrast ∈ {2.0, 4.0, 8.0, 16.0, 32.0}
Restriction: MinContrast >= 0

▷ **Radius** (input_control) ..... real ⇝ *real* / integer
Radius of the circular structuring element.
Default: 7.5
Suggested values: Radius ∈ {1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 9.5, 12.5, 15.5, 19.5, 25.5, 33.5, 45.5, 60.5, 110.5}
Restriction: Radius >= 0.5

Example

```
find_rectification_grid (Image, GridRegion, 8, 10)
dilation_circle (GridRegion, GridRegionDilated, 5.5)
reduce_domain (Image, GridRegionDilated, ImageReduced)
saddle_points_sub_pix (ImageReduced, 'facet', 1.5, 5, Row, Column)
connect_grid_points (ImageReduced, ConnectingLines, Row, Column, 1.1, 5.5)
gen_grid_rectification_map (ImageReduced, ConnectingLines, Map, Meshes, 20, \
                            'auto', Row, Column, 'bilinear')
map_image (Image, Map, ImageMapped)
```

Result

find_rectification_grid returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

dilation_circle, reduce_domain

Module

Calibration

gen_arbitrary_distortion_map (: Map : GridSpacing, Row, Column, GridWidth, ImageWidth, ImageHeight, MapType :)

Generate a projection map that describes the mapping between an arbitrarily distorted image and the rectified image.

gen_arbitrary_distortion_map computes the mapping Map between an arbitrarily distorted image and the rectified image. The coordinates Row and Column describe a distorted grid, which will be mapped onto a regular grid in the rectified image. The coordinates of the (distorted) grid points must be passed line by line in Row and Column. GridWidth is the width of the point grid in grid points. GridSpacing is the distance between

two adjacent grid points in the rectified image. Because the grid points are always mapped to the centers of their corresponding pixels a grid cell requires (GridSpacing + 1) × (GridSpacing + 1) pixels in the rectified image. Note however, that, of course, adjacent grid cells overlap by one pixel.

To compute the mapping Map, additionally the width ImageWidth and height ImageHeight of the images to be rectified must be passed.

MapType is used to specify the type of the output Map. If *'bilinear'* interpolation is chosen, Map consists of one image containing five channels. In the first channel for each pixel in the resulting image the linearized coordinates of the pixel in the input image is stored that is in the upper left position relative to the transformed coordinates. The four other channels contain the weights of the four neighboring pixels of the transformed coordinates which are used for the bilinear interpolation, in the following order:

2	3
4	5

The second channel, for example, contains the weights of the pixels that lie to the upper left relative to the transformed coordinates. If *'coord_map_sub_pix'* is chosen, Map consists of one vector field image, in which for each pixel of the resulting image the subpixel precise coordinates in the input image are stored.

As mentioned above, four adjacent pixels in the distorted image are required to interpolate the gray value of one pixel in the mapped image. If at least one of these pixels is located outside of the distorted image the gray value can not be calculated. The domain of the resulting Map domain is reduced accordingly.

In contrary to gen_grid_rectification_map, gen_arbitrary_distortion_map is used when the coordinates (Row,Column) of the grid points in the distorted image are already known or the relevant part of the image consist of regular grid structures, which the coordinates can be derived from.

If you want to re-use the created map in another program, you can save it as a multi-channel image with the operator write_image, using the format *'tiff'*.

Parameters

- ▷ **Map** (output_object) ..... multichannel-image ⇝ *object :* int4 / uint2 / vector_field
  Image containing the mapping data.
- ▷ **GridSpacing** (input_control) ..... integer ⇝ *integer*
  Distance of the grid points in the rectified image.
  Restriction: GridSpacing > 0
- ▷ **Row** (input_control) ..... point.y-array ⇝ *real*
  Row coordinates of the grid points in the distorted image.
- ▷ **Column** (input_control) ..... point.x-array ⇝ *real*
  Column coordinates of the grid points in the distorted image.
  Restriction: number(Row) == number(Column)
- ▷ **GridWidth** (input_control) ..... integer ⇝ *integer*
  Width of the point grid (number of grid points).
- ▷ **ImageWidth** (input_control) ..... extent.x ⇝ *integer*
  Width of the images to be rectified.
  Restriction: ImageWidth > 0
- ▷ **ImageHeight** (input_control) ..... extent.y ⇝ *integer*
  Height of the images to be rectified.
  Restriction: ImageHeight > 0
- ▷ **MapType** (input_control) ..... string ⇝ *string*
  Type of mapping.
  Default: 'bilinear'
  List of values: MapType ∈ {'bilinear', 'coord_map_sub_pix'}

Result

gen_arbitrary_distortion_map returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

map_image

See also

create_rectification_grid, find_rectification_grid, connect_grid_points, gen_grid_rectification_map

Module

Calibration

gen_grid_rectification_map (Image, ConnectingLines : Map, Meshes : GridSpacing, Rotation, Row, Column, MapType :)

Compute the mapping between the distorted image and the rectified image based upon the points of a regular grid.

gen_grid_rectification_map calculates the mapping between the grid points (Row,Column), which have been actually detected in the distorted image Image (typically using saddle_points_sub_pix), and the corresponding grid points of the ideal regular point grid. First, all paths that lead from their initial point via exactly four different connecting lines back to the initial point are assembled from the grid points (Row,Column) and the connecting lines ConnectingLines (detected by connect_grid_points). In case that the input of grid points (Row,Column) and of connecting lines ConnectingLines was meaningful, one such 'mesh' corresponds to exactly one grid cell in the rectification grid. Afterwards, the meshes are combined to the point grid. According to the value of Rotation, the point grid is rotated by *0*, *90*, *180* or *270* degrees. Note that the point grid does not necessarily have the correct orientation. When passing *'auto'* in Rotation, the point grid is rotated such that the black circular mark in the rectification grid is positioned to the left of the white one (see also create_rectification_grid). Finally, the mapping Map between the distorted image and the rectified image is calculated by interpolation between the grid points. Each grid cell, for which the coordinates (Row,Column) of all four corner points are known, is projected onto a square of GridSpacing × GridSpacing pixels.

MapType is used to specify the type of the output Map. If *'bilinear'* interpolation is chosen, Map consists of one image containing five channels. In the first channel for each pixel in the resulting image the linearized coordinates of the pixel in the input image is stored that is in the upper left position relative to the transformed coordinates. The four other channels contain the weights of the four neighboring pixels of the transformed coordinates which are used for the bilinear interpolation, in the following order:

2	3
4	5

The second channel, for example, contains the weights of the pixels that lie to the upper left relative to the transformed coordinates. If *'coord_map_sub_pix'* is chosen, Map consists of one vector field image, in which for each pixel of the resulting image the subpixel precise coordinates in the input image are stored.

gen_grid_rectification_map additionally returns the calculated meshes as XLD contours in Meshes.

In contrary to gen_arbitrary_distortion_map, gen_grid_rectification_map and its predecessors are used when the coordinates (Row,Column) of the grid points in the distorted image are neither known nor can be derived from the image contents.

If you want to re-use the created map in another program, you can save it as a multi-channel image with the operator write_image, using the format *'tiff'*.

Attention

Each input XLD contour ConnectingLines must own the global attribute *'bright_dark'*, as it is described with connect_grid_points!

Parameters

▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2
Input image.

▷ **ConnectingLines** (input_object) ..... xld-array ⇝ *object*
Input contours.

▷ **Map** (output_object) ..... multichannel-image ⇝ *object :* int4 / uint2 / vector_field
Image containing the mapping data.

▷ **Meshes** (output_object) ..... xld-array ⇝ *object*
Output contours.

▷ **GridSpacing** (input_control) ..... integer ⇝ *integer*
Distance of the grid points in the rectified image.
Restriction: GridSpacing > 0

▷ **Rotation** (input_control) ..... string ⇝ *string* / integer
Rotation to be applied to the point grid.
Default: 'auto'
List of values: `Rotation` ∈ {'auto', 0, 90, 180, 270}

▷ **Row** (input_control) ..... point.y-array ⇝ *real*
Row coordinates of the grid points.

▷ **Column** (input_control) ..... point.x-array ⇝ *real*
Column coordinates of the grid points.
Restriction: number(Column) == number(Row)

▷ **MapType** (input_control) ..... string ⇝ *string*
Type of mapping.
Default: 'bilinear'
List of values: `MapType` ∈ {'bilinear', 'coord_map_sub_pix'}

Result

`gen_grid_rectification_map` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`connect_grid_points`

Possible Successors

`map_image`

See also

`gen_arbitrary_distortion_map`

Module

Calibration

26.5 Hough

hough_circle_trans (Region : HoughImage : Radius :)

Return the Hough-Transform for circles with a given radius.

The operator `hough_circle_trans` calculates the Hough transform for circles with a certain `Radius` in the regions passed by `Region`. Hereby the centers of all possible circles in the parameter space (the Hough or accumulator space respectively) will be accumulated for each point in the image space. Circle hypotheses supported by many points in the input region thereby generate a maximum in the area showing the circle's center in the output image (`HoughImage`). The circles' centers in the image space can be deduced from the coordinates of these maximums by subtracting the `Radius`. If more than one radius is transmitted, all Hough images will be shifted according to the maximal radius.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
Binary edge image in which the circles are to be detected.

▷ **HoughImage** (output_object) . . . . . image(-array) ⇝ *object :* int2
Hough transform for circles with a given radius.
Number of elements: HoughImage == Radius

▷ **Radius** (input_control) . . . . . integer(-array) ⇝ *integer*
Radius of the circle to be searched in the image.
Number of elements: 1 <= Radius <= 500
Default: 12
Value range: 3 ≤ Radius (lin)
Minimum increment: 1
Recommended increment: 1

Result

The operator `hough_circle_trans` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system ('no_object_result',<Result>)`, the behavior in case of empty region is set via `set_system ('empty_region_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

hough_circles (RegionIn : RegionOut : Radius, Percent, Mode :)

Detect centers of circles for a specific radius using the Hough transform.

`hough_circles` detects the centers of circles in regions with the help of the Hough transform for circles with a specific radius.

Parameters

▷ **RegionIn** (input_object) . . . . . region ⇝ *object*
Binary edge image in which the circles are to be detected.

▷ **RegionOut** (output_object) . . . . . region(-array) ⇝ *object*
Centers of those circles which are included in the edge image by `Percent` percent.
Number of elements: RegionOut == Radius * Percent * Mode

▷ **Radius** (input_control) . . . . . integer(-array) ⇝ *integer*
Radius of the circle to be searched in the image.
Number of elements: 1 <= Radius <= 500
Default: 12
Value range: 2 ≤ `Radius` ≤ 500 (lin)
Minimum increment: 1
Recommended increment: 1

▷ **Percent** (input_control) . . . . . integer(-array) ⇝ *integer*
Indicates the percentage (approximately) of the (ideal) circle which must be present in the edge image `RegionIn`.
Number of elements: 1 <= Percent <= 100
Default: 60
Value range: 10 ≤ `Percent` ≤ 100 (lin)
Minimum increment: 1
Recommended increment: 5

▷ **Mode** (input_control) . . . . . integer(-array) ⇝ *integer*
The mode defines the position of the circle in question:
0 - the radius is equivalent to the outer border of the set pixels.
1 - the radius is equivalent to the centers of the circle lines' pixels.
2 - both 0 and 1 (a little more fuzzy, but more reliable in contrast to circles set slightly differently, necessitates 50 % more processing capacity compared to 0 or 1 alone).
Number of elements: 1 <= Mode <= 3
List of values: Mode ∈ {0, 1, 2}

Result

The operator hough_circles returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>), the behavior in case of empty region is set via set_system ('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

hough_line_trans (Region : HoughImage : AngleResolution :)

Produce the Hough transform for lines within regions.

The operator hough_line_trans calculates the Hough transform for lines in those regions transmitted by Region. Thereby the angles and the lengths of the lines' normal vectors are registered in the parameter space (the Hough- or accumulator space respectively). This means that the parametrization is executed according to the HNF.

The result is registered in a newly generated Int2-Image (HoughImage), whereby the x-axis is equivalent to the angle between the normal vector and the x-axis (in the original image), and the y-axis is equivalent to the distance of the line from the origin.

The angle ranges from -90 to 180 degrees and will be registered with a resolution of $1/$AngleResolution, which means that one pixel in x-direction is equivalent to $1/$AngleResolution and that the HoughImage has a width of $270 *$ AngleResolution $+ 1$ pixel. The height of the HoughImage corresponds to the distance between the lower right corner of the surrounding rectangle of the input region and the origin.

The maxima in the result image are equivalent to the parameter values of the lines in the original image.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
Binary edge image in which lines are to be detected.

▷ **HoughImage** (output_object) . . . . . image ⇝ *object :* int2
Hough transform for lines.

▷ **AngleResolution** (input_control) . . . . . integer ⇝ *integer*
Adjusting the resolution in the angle area.
Default: 4
List of values: AngleResolution ∈ {1, 2, 4, 8}

Result

The operator hough_line_trans returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>), the behavior in case of empty region is set via set_system ('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, skeleton

Possible Successors

threshold, local_max

See also

hough_circle_trans, gen_region_hline

Module

Foundation

hough_line_trans_dir (ImageDir : HoughImage : DirectionUncertainty, AngleResolution :)

Compute the Hough transform for lines using local gradient direction.

The operator hough_line_trans_dir calculates the Hough transform for lines in those regions passed in the domain of ImageDir. To do so, the angles and the lengths of the lines' normal vectors are registered in the parameter space (the so-called Hough or accumulator space).

In contrast to hough_line_trans, additionally the edge direction in ImageDir (e.g., returned by sobel_dir or edges_image) is taken into account. This results in a more efficient computation and in a reduction of the noise in the Hough space.

The parameter DirectionUncertainty describes how much the edge direction of the individual points within a line is allowed to vary. For example, with DirectionUncertainty = *10* a horizontal line (i.e., edge direction = 0 degrees) may contain points with an edge direction between -10 and +10 degrees. The higher DirectionUncertainty is chosen, the higher the computation time will be. For DirectionUncertainty = *180* hough_line_trans_dir shows the same behavior as hough_line_trans, i.e., the edge direction is ignored. DirectionUncertainty should be chosen at least as high as the step width of the edge direction stored in ImageDir. The minimum step width is 2 degrees (defined by the image type 'direction').

The result is stored in a newly generated UINT2-Image (HoughImage), where the x-axis (i.e., columns) represents the angle between the normal vector and the x-axis of the original image, and the y-axis (i.e., rows) represents the distance of the line from the origin.

The angle ranges from -90 to 180 degrees and will be stored with a resolution of $1/$AngleResolution, which means that one pixel in x-direction is equivalent to $1/$AngleResolution degrees and that the HoughImage has a width of $270 * $AngleResolution$ + 1$ pixels. The height of the HoughImage corresponds to the distance between the lower right corner of the surrounding rectangle of the input region and the origin.

The local maxima in the result image are equivalent to the parameter values of the lines in the original image.

Parameters

▷ **ImageDir** (input_object) ..... singlechannelimage ⇝ *object :* direction
Image containing the edge direction. The edges must be described by the image domain.

▷ **HoughImage** (output_object) ..... image ⇝ *object :* uint2
Hough transform.

▷ **DirectionUncertainty** (input_control) ..... angle.deg ⇝ *integer*
Uncertainty of the edge direction (in degrees).
Default: 2
Value range: $2 \leq$ DirectionUncertainty ≤ 180
Minimum increment: 2

▷ **AngleResolution** (input_control) ..... integer ⇝ *integer*
Resolution in the angle area (in 1/degrees).
Default: 4
List of values: AngleResolution $\in \{1, 2, 4, 8\}$

Result

The operator hough_line_trans_dir returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input is set via the operator set_system('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

edges_image, sobel_dir, threshold, hysteresis_threshold, nonmax_suppression_dir, reduce_domain

Possible Successors

binomial_filter, gauss_filter, threshold, local_max, plateaus_center

See also

hough_line_trans, hough_lines, hough_lines_dir

Module

Foundation

hough_lines (RegionIn : : AngleResolution, Threshold, AngleGap, DistGap : Angle, Dist)

Detect lines in edge images with the help of the Hough transform and returns it in HNF.

The operator hough_lines allows the selection of linelike structures in a region, whereby it is not necessary that the individual points of a line are connected. This process is based on the Hough transform.

The parameter AngleResolution defines the degree of exactness concerning the determination of the angles. It amounts to 1/AngleResolution degree. The parameter Threshold determines by how many points of the original region a line's hypothesis must at least be supported in order to be selected into the output. The parameters AngleGap and DistGap define a neighborhood of the points in the Hough image in order to determine the local maxima: AngleGap describes the minimum distance of two maxima in the Hough image in angle direction and DistGap in distance direction, respectively. Thus, maxima exceeding Threshold but lying close to an even higher maximum are eliminated. If multiple maxima in this neighborhood are equally high, all of them are returned. This elimination can particularly be helpful when searching for short and long lines simultaneously. The lines are returned in Hessian Normal Form (HNF), that is by the direction Angle and length Dist of their normal vectors.

Parameters

▷ **RegionIn** (input_object) ..... region ⇝ *object*
Binary edge image in which the lines are to be detected.

▷ **AngleResolution** (input_control) ..... integer ⇝ *integer*
Adjusting the resolution in the angle area.
Default: 4
List of values: AngleResolution ∈ {1, 2, 4, 8}

▷ **Threshold** (input_control) ..... integer ⇝ *integer*
Threshold value in the Hough image.
Default: 100
Value range: 2 ≤ Threshold

▷ **AngleGap** (input_control) ..... integer ⇝ *integer*
Minimal distance of two maxima in the Hough image (direction: angle).
Default: 5
Value range: 0 ≤ AngleGap

▷ **DistGap** (input_control) . . . . . integer ⇝ *integer*
Minimal distance of two maxima in the Hough image (direction: distance).
Default: 5
Value range: 0 ≤ DistGap

▷ **Angle** (output_control) . . . . . hesseline.angle.rad-array ⇝ *real*
Angles (in radians) of the detected lines' normal vectors.
Value range: -1.5707963 ≤ Angle ≤ 3.1415927

▷ **Dist** (output_control) . . . . . hesseline.distance-array ⇝ *real*
Distance of the detected lines from the origin.
Number of elements: Dist == Angle
Value range: 0 ≤ Dist

Result

The operator hough_lines returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>), the behavior in case of empty region is set via set_system ('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, skeleton

Possible Successors

select_matching_lines

See also

hough_line_trans, gen_region_hline, hough_circles

Module

Foundation

```
hough_lines_dir ( ImageDir : HoughImage,
    Lines : DirectionUncertainty, AngleResolution, Smoothing,
    FilterSize, Threshold, AngleGap, DistGap, GenLines : Angle,
    Dist )
```

Detect lines in edge images with the help of the Hough transform using local gradient direction and return them in normal form.

The operator hough_lines_dir selects line-like structures in a region based on the Hough transform. The individual points of a line can be unconnected. The region is given by the domain of ImageDir.

In contrast to hough_lines, additionally the edge direction in ImageDir (e.g., returned by sobel_dir or edges_image) is taken into account. This results in a more efficient computation and in a reduction of the noise in the Hough space.

The parameter DirectionUncertainty describes how much the edge direction of the individual points within a line is allowed to vary. For example, with DirectionUncertainty = *10* a horizontal line (i.e., edge direction = 0 degrees) may contain points with an edge direction between -10 and +10 degrees. The higher DirectionUncertainty is chosen, the higher the computation time will be. For DirectionUncertainty = *180* hough_lines_dir shows the same behavior as hough_lines, i.e., the edge direction is ignored. DirectionUncertainty should be chosen at least as high as the step width of the edge direction stored in ImageDir. The minimum step width is 2 degrees (defined by the image type 'direction').

The parameter AngleResolution defines how accurately the angles are determined. The accuracy amounts to 1/AngleResolution degrees. A subsequent smoothing of the Hough space results in an increased stability. The smoothing filter can be selected by Smoothing, the degree of smoothing by the parameter FilterSize (see mean_image or gauss_image for details). The parameter Threshold determines by how many points

of the original region a line's hypothesis must at least be supported in order to be selected into the output. The parameters AngleGap and DistGap define a neighborhood of the points in the Hough image in order to determine the local maxima: AngleGap describes the minimum distance of two maxima in the Hough image in angle direction and DistGap in distance direction, respectively. Thus, maxima exceeding Threshold but lying close to an even higher maximum are eliminated. If multiple maxima in this neighborhood are equally high, all of them are returned. This elimination can particularly be helpful when searching for short and long lines simultaneously. Besides the not smoothed Hough image HoughImage, the lines are returned in Hessian Normal Form (HNF), that is by the direction Angle and length Dist of their normal vectors. If the parameter GenLines is set to *'true'*, additionally those regions in ImageDir are returned that contributed to the local maxima in Hough space. They are stored in the parameter Lines.

Parameters

- ▷ **ImageDir** (input_object) . . . . . singlechannelimage ⇝ *object :* direction
  Image containing the edge direction. The edges are described by the image domain.
- ▷ **HoughImage** (output_object) . . . . . image ⇝ *object :* uint2
  Hough transform.
- ▷ **Lines** (output_object) . . . . . region-array ⇝ *object*
  Regions of the input image that contributed to the local maxima.
- ▷ **DirectionUncertainty** (input_control) . . . . . angle.deg ⇝ *integer*
  Uncertainty of edge direction (in degrees).
  Default: 2
  Value range: 2 ≤ DirectionUncertainty ≤ 180
  Minimum increment: 2
- ▷ **AngleResolution** (input_control) . . . . . integer ⇝ *integer*
  Resolution in the angle area (in 1/degrees).
  Default: 4
  List of values: AngleResolution ∈ {1, 2, 4, 8}
- ▷ **Smoothing** (input_control) . . . . . string ⇝ *string*
  Smoothing filter for hough image.
  Default: 'mean'
  List of values: Smoothing ∈ {'none', 'mean', 'gauss'}
- ▷ **FilterSize** (input_control) . . . . . integer ⇝ *integer*
  Required smoothing filter size.
  Default: 5
  List of values: FilterSize ∈ {3, 5, 7, 9, 11}
- ▷ **Threshold** (input_control) . . . . . integer ⇝ *integer*
  Threshold value in the Hough image.
  Default: 100
  Value range: 1 ≤ Threshold
- ▷ **AngleGap** (input_control) . . . . . integer ⇝ *integer*
  Minimum distance of two maxima in the Hough image (direction: angle).
  Default: 5
  Value range: 0 ≤ AngleGap
- ▷ **DistGap** (input_control) . . . . . integer ⇝ *integer*
  Minimum distance of two maxima in the Hough image (direction: distance).
  Default: 5
  Value range: 0 ≤ DistGap
- ▷ **GenLines** (input_control) . . . . . string ⇝ *string*
  Create line regions if *'true'*.
  Default: 'true'
  List of values: GenLines ∈ {'true', 'false'}
- ▷ **Angle** (output_control) . . . . . hesseline.angle.rad-array ⇝ *real*
  Angles (in radians) of the detected lines' normal vectors.
  Value range: -1.5707963 ≤ Angle ≤ 3.1415927
- ▷ **Dist** (output_control) . . . . . hesseline.distance-array ⇝ *real*
  Distance of the detected lines from the origin.
  Number of elements: Dist == Angle
  Value range: 0 ≤ Dist

Result

The operator hough_lines returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

edges_image, sobel_dir, threshold, nonmax_suppression_dir, reduce_domain, skeleton

Possible Successors

gen_region_hline, select_matching_lines

See also

hough_line_trans_dir, hough_line_trans, gen_region_hline, hough_circles

Module

Foundation

select_matching_lines (RegionIn : RegionLines : AngleIn, DistIn, LineWidth, Thresh : AngleOut, DistOut)

Select those lines from a set of lines (in HNF) which fit best into a region.

Lines which fit best into a region can be selected from a set of lines which are available in HNF with the help of the operator select_matching_lines; the region itself is also transmitted as a parameter (RegionIn). The width of the lines can be indicated by the parameter LineWidth. The selected lines will be returned in HNF and as regions (RegionLines).

The lines are selected iteratively in a loop: At first, the line showing the greatest overlap with the input region is selected from the set of input lines. This line will then be taken over into the output set whereby all points belonging to that line will not be considered in the further steps determining overlaps. The loop will be left when the maximum overlap value of the region and the lines falls below a certain threshold value (Thresh). The selected lines will be returned as regions as well as in HNF.

Parameters

▷ **RegionIn** (input_object) . . . . . region ⇝ *object*
Region in which the lines are to be matched.

▷ **RegionLines** (output_object) . . . . . region(-array) ⇝ *object*
Region array containing the matched lines.

▷ **AngleIn** (input_control) . . . . . hesseline.angle.rad(-array) ⇝ *real*
Angles (in radians) of the normal vectors of the input lines.
Value range: -1.5707963 ≤ AngleIn ≤ 3.1415927

▷ **DistIn** (input_control) . . . . . hesseline.distance(-array) ⇝ *real*
Distances of the input lines form the origin.
Number of elements: DistIn == AngleIn

▷ **LineWidth** (input_control) . . . . . integer ⇝ *integer*
Widths of the lines.
Default: 7
Value range: 1 ≤ LineWidth

▷ **Thresh** (input_control) . . . . . integer ⇝ *integer*
Threshold value for the number of line points in the region.
Default: 100
Value range: 1 ≤ Thresh

▷ **AngleOut** (output_control) ..... hesseline.angle.rad(-array) ⇝ *real*
Angles (in radians) of the normal vectors of the selected lines.
Number of elements: AngleOut <= AngleIn
Value range: -1.5707963 ≤ AngleOut ≤ 3.1415927

▷ **DistOut** (output_control) ..... hesseline.distance(-array) ⇝ *real*
Distances of the selected lines from the origin.
Number of elements: DistOut == AngleOut
Value range: 0 ≤ DistOut

Result

The operator select_matching_lines returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result',<Result>), the behavior in case of empty region is set via set_system ('empty_region_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hough_lines

Module

Foundation

26.6 Interpolation

clear_scattered_data_interpolator (
: : ScatteredDataInterpolatorHandle :)

Clear a scattered data interpolator.

clear_scattered_data_interpolator clears the scattered data interpolator given by ScatteredDataInterpolatorHandle and frees all memory required for the interpolator. After calling clear_scattered_data_interpolator, the scattered data interpolator can no longer be used. The handle ScatteredDataInterpolatorHandle becomes invalid.

Parameters

▷ **ScatteredDataInterpolatorHandle** (input_control) ..... scattered_data_interpolator(-array) ⇝ *handle*
Handle of the scattered data interpolator

Result

If the parameters are valid, the operator clear_scattered_data_interpolator returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- ScatteredDataInterpolatorHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_scattered_data_interpolator, interpolate_scattered_data

Module

Foundation

create_scattered_data_interpolator (: : Method, Rows, Columns, Values, GenParamName, GenParamValue : ScatteredDataInterpolatorHandle)

Creates an interpolator for the interpolation of scattered data.

create_scattered_data_interpolator creates the interpolator ScatteredDataInterpolatorHandle for the interpolation of scattered data, given as data points in Rows and Columns with the corresponding measurement Values. With the parameter Method the algorithm is specified, which should be used for the interpolation with interpolate_scattered_data. So far, only the *'thin_plate_splines'* are supported. This method interpolates on a global scale, which means that all points are regarded for the interpolation, no matter how far away they are. The influence of far points is correlated to $r^2 \cdot \log(r)$ where r defines the distance of two points. In comparison to interpolate_scattered_data_image, create_scattered_data_interpolator also supports subpixel coordinates for Rows and Columns. After the creation, interpolate_scattered_data is called for the interpolation of the unknown values. By splitting up the creation (create_scattered_data_interpolator) and the evaluation (interpolate_scattered_data) of the interpolator, interpolating at different data points in subsequent steps becomes more efficient since the interpolator has to be created only once.

The following parameters can be adjusted with GenParamName and GenParamValue:

'alpha': The parameter *'alpha'* is a smoothing factor. For *'alpha'* $= 0$, all points passed in Rows, Columns, and Values are interpolated exactly. With *'alpha'* getting larger, the interpolation smoothes the points in way that all interpolated points lie on a common plane.
Default: 0
Restriction: 'alpha' ≥ 0

Parameters

▷ **Method** (input_control) . . . . . string ⇝ *string*
Method for the interpolation
Default: 'thin_plate_splines'
Suggested values: Method ∈ {'thin_plate_splines'}

▷ **Rows** (input_control) . . . . . point.y-array ⇝ *real* / integer
Row coordinates of the points used for the interpolation

▷ **Columns** (input_control) . . . . . point.x-array ⇝ *real* / integer
Column coordinates of the points used for the interpolation

▷ **Values** (input_control) . . . . . number-array ⇝ *real* / integer
Values of the points used for the interpolation

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters that can be adjusted
Default: []
Suggested values: GenParamName ∈ {'alpha'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / string / integer
Values of the generic parameters that can be adjusted
Default: []
Suggested values: GenParamValue ∈ {0, 1.0, 10.0, 100.0}

▷ **ScatteredDataInterpolatorHandle** (output_control) . . . . . scattered_data_interpolator ⇝ *handle*
Handle of the scattered data interpolator

Result

If the parameters are valid, the operator create_scattered_data_interpolator returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

interpolate_scattered_data

Module

Foundation

interpolate_scattered_data (: : ScatteredDataInterpolatorHandle, Row, Column : ValueInterpolated)

Interpolation of scattered data using a scattered data interpolator.

interpolate_scattered_data interpolates the values of the scattered data points (Row,Column) by using the scattered data interpolator ScatteredDataInterpolatorHandle and returns the result in ValueInterpolated. In comparison to interpolate_scattered_data_image, interpolate_scattered_data also supports subpixel coordinates for Row and Column. Before calling interpolate_scattered_data, the scattered data interpolator must be created with create_scattered_data_interpolator.

Parameters

▷ **ScatteredDataInterpolatorHandle** (input_control) . . . . . scattered_data_interpolator ⇝ *handle*
Handle of the scattered data interpolator

▷ **Row** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
Row coordinates of points to be interpolated

▷ **Column** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
Column coordinates of points to be interpolated

▷ **ValueInterpolated** (output_control) . . . . . number(-array) ⇝ *real* / integer
Values of interpolated points

Result

If the parameters are valid, the operator interpolate_scattered_data returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_scattered_data_interpolator

Possible Successors

clear_scattered_data_interpolator, gen_object_model_3d_from_points

Module

Foundation

interpolate_scattered_data_image (Image, RegionInterpolate : ImageInterpolated : Method, GenParamName, GenParamValue :)

Interpolation of an image.

interpolate_scattered_data_image interpolates the Image at the region RegionInterpolate and returns the result in ImageInterpolated. The difference of the domain of the Image and the region RegionInterpolate specifies the valid data points that can be used for the interpolation whereas RegionInterpolate specifies the points, where the gray values of the Image should be determined. With the parameter Method the interpolation algorithm is specified. So far, only the *'thin_plate_splines'* are supported. This method interpolates on a global scale, which means that all points are regarded for the interpolation, no matter how far away they are. The influence of far points is correlated to $r^2 \cdot \log(r)$ where r defines the distance of two points. If the same Image is interpolated at different points in subsequent steps, the operator create_scattered_data_interpolator may be more efficient.

The following parameters can be adjusted with GenParamName and GenParamValue:

'alpha': The parameter *'alpha'* is a smoothing factor. For *'alpha'* = 0, all points in the image Image are interpolated exactly. With *'alpha'* getting larger, the interpolation smoothes the image points in way that all interpolated points of the result image ImageInterpolated lie on a common plane.
Default: is *0*
Restriction: 'alpha' $\geq$ *0*

interpolate_scattered_data_image is best used with very few data points, e.g, less than 1000. In order to reconstruct destroyed image data in the region RegionInterpolate, the data points should be reduced to an appropriate number, e.g., by only using the border pixels of the hole regions. An example program is shown below.

Parameters

- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte / uint2 / real
  Image to interpolate
- ▷ **RegionInterpolate** (input_object) ..... region ⇝ *object*
  Region to interpolate
- ▷ **ImageInterpolated** (output_object) ..... singlechannelimage ⇝ *object :* real
  Interpolated image
- ▷ **Method** (input_control) ..... string ⇝ *string*
  Method for the interpolation
  Default: 'thin_plate_splines'
  Suggested values: Method ∈ {'thin_plate_splines'}
- ▷ **GenParamName** (input_control) ..... attribute.name-array ⇝ *string*
  Names of the generic parameters that can be adjusted
  Default: []
  Suggested values: GenParamName ∈ {'alpha'}
- ▷ **GenParamValue** (input_control) ..... attribute.value-array ⇝ *string* / integer / real
  Values of the generic parameters that can be adjusted
  Default: []
  Suggested values: GenParamValue ∈ {0, 1.0, 10.0, 100.0}

Example

```
* This example program shows how to use the scattered data interpolator
* to fill holes in an image
gen_image_surface_second_order (ImageData, 'real', 1, 1, 0, 0, 0, 1, \
                                24, 24, 48, 48)
gen_circle (Circle, 12, 12, 6)
difference (ImageData, Circle, Region)
reduce_domain (ImageData, Region, ImageReduced)
dev_clear_window ()
dev_display (ImageReduced)
stop()
*
* Select only border pixels for the interpolation
dilation_circle (Circle, CircleDilation, 1.5)
intersection (CircleDilation, Region, RegionBorderData)
dev_clear_window ()
```

```
dev_display (ImageReduced)
dev_display (RegionBorderData)
stop()
*
* Interpolate pixels
reduce_domain (ImageData, RegionBorderData, ImageReducedBorder)
interpolate_scattered_data_image (ImageReducedBorder, Circle, \
                                  ImageInterpolated, \
                                  'thin_plate_splines', [], [])
paint_gray (ImageInterpolated, ImageData, ImageFilled)
dev_clear_window ()
dev_display (ImageFilled)
```

Result

If the parameters are valid, the operator `interpolate_scattered_data_image` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`interpolate_scattered_data_points_to_image`

Module

Foundation

interpolate_scattered_data_points_to_image (
: ImageInterpolated : Method, Rows, Columns, Values, Width, Height, GenParamName, GenParamValue :)

Creating an image from the interpolation of scattered data.

`interpolate_scattered_data_points_to_image` interpolates the data points, given in `Rows` and `Columns` with the corresponding measurement `Values`, and returns the result in `ImageInterpolated`. The size of the output image is defined by its `Width` and its `Height` with the upper left corner at (0,0). In order to interpolate at negative coordinates of (`Rows`,`Columns`), simply translate all input points accordingly. With the parameter `Method` the interpolation algorithm is specified. So far, only the *'thin_plate_splines'* are supported. This method interpolates on a global scale, which means that all points are regarded for the interpolation, no matter how far away they are. The influence of far points is correlated to $r^2 \cdot \log(r)$ where r defines the distance of two points. In comparison to `interpolate_scattered_data_image`, `interpolate_scattered_data_points_to_image` also supports subpixel coordinates for `Rows` and `Columns`. If the same data points (`Rows`,`Columns`,`Values`) are used for the interpolation of different output images in subsequent steps, the operator `create_scattered_data_interpolator` may be more efficient.

The following parameters can be adjusted with `GenParamName` and `GenParamValue`:

'alpha': The parameter *'alpha'* is a smoothing factor. For *'alpha'* = 0, all points passed in (`Rows`,`Columns`,`Values`) are interpolated exactly. With *'alpha'* getting larger, the interpolation smoothes the points in way that all interpolated points of the result image `ImageInterpolated` lie on a common plane.

Default: 0.

Restriction: 'alpha' ≥ 0

Parameters

▷ **ImageInterpolated** (output_object) . . . . . singlechannelimage ⇝ *object :* real
Interpolated image

▷ **Method** (input_control) . . . . . string ⇝ *string*
Method for the interpolation
Default: 'thin_plate_splines'
Suggested values: Method ∈ {'thin_plate_splines'}

▷ **Rows** (input_control) . . . . . point.y-array ⇝ *real* / integer
Row coordinates of the points used for the interpolation

▷ **Columns** (input_control) . . . . . point.x-array ⇝ *real* / integer
Column coordinates of the points used for the interpolation

▷ **Values** (input_control) . . . . . number-array ⇝ *real* / integer
Values of the points used for the interpolation

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the interpolated image
Default: 640

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the interpolated image
Default: 480

▷ **GenParamName** (input_control) . . . . . attribute.name-array ⇝ *string*
Names of the generic parameters that can be adjusted
Default: []
Suggested values: GenParamName ∈ {'alpha'}

▷ **GenParamValue** (input_control) . . . . . attribute.value-array ⇝ *real* / string / integer
Values of the generic parameters that can be adjusted
Default: []
Suggested values: GenParamValue ∈ {0, 1.0, 10.0, 100.0}

Result

If the parameters are valid, the operator interpolate_scattered_data_points_to_image returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

interpolate_scattered_data_image

Module

Foundation

26.7 Lines

line_orientation (: : RowBegin, ColBegin, RowEnd, ColEnd : Phi)

Calculate the orientation of lines.

The operator line_orientation returns the orientation ($-pi/2 < \texttt{Phi} \leq pi/2$) of the given lines. If more than one line is to be treated the line and column indices can be passed as tuples. In this case Phi is, of course, also a tuple and contains the corresponding orientations.

Parameters

▷ **RowBegin** (input_control) . . . . . line.begin.y(-array) ⇝ *real* / integer
Row coordinates of the starting points of the input lines.

▷ **ColBegin** (input_control) . . . . . line.begin.x(-array) ⇝ *real* / integer
Column coordinates of the starting points of the input lines.

▷ **RowEnd** (input_control) . . . . . line.end.y(-array) ⇝ *real* / integer
Row coordinates of the ending points of the input lines.

▷ **ColEnd** (input_control) . . . . . line.end.x(-array) ⇝ *real* / integer
Column coordinates of the ending points of the input lines.

▷ **Phi** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Orientation of the input lines.

Result

`line_orientation` always returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`sobel_amp`, `edges_image`, `threshold`, `hysteresis_threshold`, `split_skeleton_region`, `split_skeleton_lines`

Possible Successors

`set_line_width`, `disp_line`

Alternatives

`line_position`

See also

`line_position`, `detect_edge_segments`

Module

Foundation

line_position (: : RowBegin, ColBegin, RowEnd, ColEnd : RowCenter, ColCenter, Length, Phi)

Calculate the center of gravity, length, and orientation of a line.

The operator `line_position` returns the center (`RowCenter`, `ColCenter`), the (Euclidean) length (`Length`) and the orientation ($-pi/2 <$ `Phi` $\leq pi/2$) of the given lines. If more than one line is to be treated the line and column indices can be passed as tuples. In this case the output parameters, of course, are also tuples.

The routine is applied, for example, to model lines in order to determine search regions for the edge detection (`detect_edge_segments`).

Parameters

▷ **RowBegin** (input_control) . . . . . line.begin.y(-array) ⇝ *real* / integer
Row coordinates of the starting points of the input lines.

▷ **ColBegin** (input_control) . . . . . line.begin.x(-array) ⇝ *real* / integer
Column coordinates of the starting points of the input lines.

▷ **RowEnd** (input_control) . . . . . line.end.y(-array) ⇝ *real* / integer
Row coordinates of the ending points of the input lines.

▷ **ColEnd** (input_control) . . . . . line.end.x(-array) ⇝ *real* / integer
Column coordinates of the ending points of the input lines.

▷ **RowCenter** (output_control) . . . . . point.y(-array) ⇝ *real*
Row coordinates of the centers of gravity of the input lines.

▷ **ColCenter** (output_control) . . . . . point.x(-array) ⇝ *real*
Column coordinates of the centers of gravity of the input lines.

▷ **Length** (output_control) . . . . . real(-array) ⇝ *real*
Euclidean length of the input lines.

▷ **Phi** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Orientation of the input lines.

Result

`line_position` always returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`sobel_amp`, `edges_image`, `threshold`, `hysteresis_threshold`, `split_skeleton_region`, `split_skeleton_lines`

Possible Successors

`set_line_width`, `disp_line`

Alternatives

`line_orientation`

See also

`line_orientation`, `detect_edge_segments`

Module

Foundation

26.8 Mosaicking

```
adjust_mosaic_images ( Images : CorrectedImages : From, To,
    ReferenceImage, HomMatrices2D, EstimationMethod,
    EstimateParameters, OECFModel : )
```

Apply an automatic color correction to panorama images.

`adjust_mosaic_images` performs the radiometric adjustment of images in panoramas. The images to be corrected have to be passed in `Images`, the corrected images will be returned in `CorrectedImages`.

The parameters `From` and `To` must contain the source and destination indices of all image pairs in the panorama. The projective 3x3-matrix of each image pair has to be passed in `HomMatrices2D`. The image, which will be used as the reference for brightness and white balance, is selected with the parameter `ReferenceImage`.

This means, that one image specifies the "ideal" brightness and white balance settings. All other images will be corrected such that their brightness and white balance match that of the reference image. In other words, the reference image will not be changed, but all other images will be.

`EstimationMethod` is used for choosing whether a fast but less accurate, or a slower but more accurate determination method should be used. This is done by setting `EstimationMethod` either to *'standard'* or *'gold_standard'*.

The methods based on *'standard'* use only the average gray value difference of all images in the overlap area between each image pair. With *'gold_standard'* the gray value difference for each pixel in the overlap area is taken into account explicitly.

The error function that is minimized in all cases is computed as the summed square of the difference between the corresponding gray values.

The availability of the individual method is depending on the selected `EstimateParameters`, which determines the model to be used for estimating the radiometric adjustment terms. It is always possible to determine the

amount of vignetting in the images by selecting *'vignetting'*. However, if selected, EstimationMethod must be set to *'gold_standard'*. For the remainder of the radiometric adjustment three different options are available:

Estimation of vignetting in images is based on the commonly known

$$\cos^4 Alpha$$

approach. This approach assumes that vignetting does not exist in the center of the image and increases with the opening angle by the above equation.

1. Image adjustment with the additive model. This should only be used to adjust images with very small differences in exposure or white balance. To choose this method, EstimateParameters must be set to *'add_gray'*. This model can be selected either exclusively and only with EstimationMethod = *'standard'* or in combination with EstimateParameters = *'vignetting'* and only with EstimationMethod = *'gold_standard'*.

This model is based on the assumption, that the gray value differences between the images can be corrected by adding an individual value to each image accept the reference image. Basically, the modification to every image can be expressed as a call of scale_image:

```
scale_image(Image_n,CorrectedImage_n,1.0,Add_n)
```

where Add_n is the correction term for this image.

2. Image adjustment with the linear model. In this model, images are expected to be taken with a camera using a linear transfer function. The adjustment terms are consequently represented as multiplication factors. To select this model, EstimateParameters must be set to *'mult_gray'*. It can be called with EstimationMethod = *'standard'* or EstimationMethod = *'gold_standard'*. A combined call with EstimateParameters = *'vignetting'* is also possible, EstimationMethod must be set to *'gold_standard'* in that case.

This model is based on the assumption that the gray value differences between the images can be corrected by multiplying the gray values in each image by a factor. Basically, the modification to every image can again be expressed as a call of scale_image:

```
scale_image(Image_n,CorrectedImage_n,Mult_n,0)
```

where Mult_n is the correction term for this image.

3. Image adjustment with the calibrated model. In this model, images are assumed to be taken with a camera using a nonlinear transfer function. A function of the OECF class selected with OECFModel is used to approximate the actually used OECF in the process of image acquisition. As with the linear model, the correction terms are represented as multiplication factors. This model can be selected by choosing EstimateParameters = *['mult_gray','response']* and must be called with EstimationMethod = *'gold_standard'*. It is possible to determine the amount of vignetting as well in this case by choosing EstimateParameters = *'vignetting'*.

This model is similar to the linear model. However, in this case the camera may have a nonlinear response. This means that before the gray values of the images can be multiplied by their respective correction factor, the gray values must be backprojected to a linear response. To do so, the camera's response must be determined. Since the response usually does not change over an image sequence, this parameter is assumed to be constant throughout the whole image sequence.

Any kind of function could be considered to be used as an OECF. As in the operator radiometric_self_calibration, a polynomial fitting might be used, but for typical images in a mosaicking application this would not work very well. The reason for this is that polynomial fitting has too many parameters that need to be determined. Instead, only simpler types of response functions can be estimated. Currently, only so-called Laguerre-functions are available.

The response of a Laguerre-type OECF is determined by only one parameter called Phi. In a first step, the whole gray value spectrum (in case of 8bit images the values 0 to 255) is converted to floating point numbers in the interval [0:1]. Then, the OECF backprojection is calculated based on this and the resulting gray values are once again converted to the original interval.

The inverse transform of the gray values back to linear values based on a Laguerre-type OECF is described by the following equation:

$$I\_l = I\_nl + \frac{2}{\pi} \cdot \arctan\left(\frac{Phi \cdot \sin(\pi \cdot I\_nl)}{1 - Phi \cdot \cos(\pi \cdot I\_nl)}\right)$$

with I_l the linear gray value and I_nl the (nonlinear) gray value.

The parameter `OECFModel` is only used if the calibrated model has been chosen. Otherwise, any input for `OECFModel` will be ignored.

The parameter `EstimateParameters` can also be used to influence the performance and memory consumption of the operator. With *'no_cache'* the internal caching mechanism can be disabled. This switch only has and influence if `EstimationMethod` is set to *'gold_standard'*. Otherwise this switch will be ignored. When disabling the internal caching, the operator uses far less memory, but therefore calculates the corresponding gray value pairs in each iteration of the minimization algorithm again. Therefore, disabling caching is only advisable if all physical memory is used up at some point of the calculation and the operating system starts using swap space.

A second option to influence the performance is using subsampling. When setting `EstimateParameters` to *'subsampling_2'*, images are internally zoomed down by a factor of 2. Despite the suggested value list, not only factors of 2 and 4 are available, but any integer number might be specified by appending it to subsampling_ in `EstimateParameters`. With this, the amount of image data is tremendously reduced, which leads to a much faster computation of the internal minimization. In fact, using moderate subsampling might even lead to better results since it also decreases the influence of slightly misaligned pixels. Although subsampling also influences the minimization if `EstimationMethod` is set to *'standard'*, it is mostly useful for *'gold_standard'*.

Some more general remarks on using `adjust_mosaic_images` in applications:

- Estimation of vignetting will only work well if significant vignetting is visible in the images. Otherwise, the operator may lead to erratic results.
- Estimation of the response is rather slow because the problem is quite complex. Therefore, it is advisable not to determine the response in time critical applications. Apart from this, the response can only be determined correctly if there are relatively large brightness differences between the images.
- It is not possible to correct saturation. If there are saturated areas in an image, they will remain saturated.
- `adjust_mosaic_images` can only be used to correct different brightness in images, which is caused by different exposure (shutter time, aperture) or different light intensity. It cannot be used to correct brightness differences based on inhomogeneous illumination within each image.

Parameters

▷ **`Images`** (input_object) ..... (multichannel-)image-array ⇝ *object :* byte
Input images.

▷ **`CorrectedImages`** (output_object) ..... (multichannel-)image-array ⇝ *object :* byte
Output images.

▷ **`From`** (input_control) ..... integer-array ⇝ *integer*
List of source images.

▷ **`To`** (input_control) ..... integer-array ⇝ *integer*
List of destination images.

▷ **`ReferenceImage`** (input_control) ..... integer ⇝ *integer*
Reference image.

▷ **`HomMatrices2D`** (input_control) ..... real-array ⇝ *real*
Projective matrices.

▷ **`EstimationMethod`** (input_control) ..... string ⇝ *string*
Estimation algorithm for the correction.
Default: 'standard'
List of values: `EstimationMethod` ∈ {'standard', 'gold_standard'}

▷ **`EstimateParameters`** (input_control) ..... string(-array) ⇝ *string*
Parameters to be estimated.
Default: ['mult_gray']
Suggested values: `EstimateParameters` ∈ {'add_gray', 'mult_gray', 'response', 'vignetting', 'subsampling_2', 'subsampling_4', 'no_cache'}

▷ **`OECFModel`** (input_control) ..... string ⇝ *string*
Model of OECF to be used.
Default: ['laguerre']
List of values: `OECFModel` ∈ {'laguerre'}

Example

```
* For the input data to stationary_camera_self_calibration, please
* refer to the example for stationary_camera_self_calibration.
stationary_camera_self_calibration (4, 640, 480, 1, From, To, \
                                    HomMatrices2D, Rows1, Cols1, \
                                    Rows2, Cols2, NumMatches, \
                                    'gold_standard', \
                                    ['focus','principal_point'], \
                                    'true', CameraMatrix, Kappa, \
                                    RotationMatrices, X, Y, Z, Error)
adjust_mosaic_images(Images,CorrectedImages,From,To,1,HomMatrices2D, \
                     'gold_standard',['mult_gray','response'],'laguerre')
```

Result

If the parameters are valid, the operator `adjust_mosaic_images` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`stationary_camera_self_calibration`

Possible Successors

`gen_spherical_mosaic`

References

David Hasler, Sabine Süsstrunk: Mapping colour in image stitching applications. Journal of Visual Communication and Image Representation, 15(1):65-90, 2004.

Module

Foundation

bundle_adjust_mosaic (: : NumImages, ReferenceImage, MappingSource, MappingDest, HomMatrices2D, Rows1, Cols1, Rows2, Cols2, NumCorrespondences, Transformation : MosaicMatrices2D, Rows, Cols, Error)

Perform a bundle adjustment of an image mosaic.

`bundle_adjust_mosaic` performs a bundle adjustment of an image mosaic. This can be used to determine the geometry of a mosaic as robustly as possible, and hence to determine the transformations of the images in the mosaic more accurately than with single image pairs.

To achieve this, the projective transformation for each overlapping image pair in the mosaic should be determined with `proj_match_points_ransac`. For example, for a 2×2 block of images in the following layout

1	2
3	4

the following projective transformations should be determined, assuming that all images overlap each other: 1↦2, 1↦3, 1↦4, 2↦3, 2↦4 and 3↦4. The indices of the images that determine the respective transformation are given by `MappingSource` and `MappingDest`. The indices are start at 1. Consequently, in the above example `MappingSource` = *[1,1,1,2,2,3]* and `MappingDest` = *[2,3,4,3,4,4]* must be used. The number of images in the mosaic is given by `NumImages`. It is used to check whether each image can be reached by a chain of

transformations. The index of the reference image is given by ReferenceImage. On output, this image has the identity matrix as its transformation matrix.

The 3×3 projective transformation matrices that correspond to the image pairs are passed in HomMatrices2D. Additionally, the coordinates of the matched point pairs in the image pairs must be passed in Rows1, Cols1, Rows2, and Cols2. They can be determined from the output of proj_match_points_ransac with tuple_select or with the HDevelop function subset. To enable bundle_adjust_mosaic to determine which point pair belongs to which image pair, NumCorrespondences must contain the number of found point matches for each image pair.

The parameter Transformation determines the class of transformations that is used in the bundle adjustment to transform the image points. This can be used to restrict the allowable transformations. For Transformation = *'projective'*, projective transformations are used (see vector_to_proj_hom_mat2d). For Transformation = *'affine'*, affine transformations are used (see vector_to_hom_mat2d), for Transformation = *'similarity'*, similarity transformations (see vector_to_similarity), and for Transformation = *'rigid'* rigid transformations (see vector_to_rigid).

The resulting bundle-adjusted transformations are returned as an array of 3×3 projective transformation matrices in MosaicMatrices2D. In addition, the points reconstructed by the bundle adjustment are returned in (Rows, Cols). The average projection error of the reconstructed points is returned in Error. This can be used to check whether the optimization has converged to useful values.

Parameters

- ▷ **NumImages** (input_control) ..... integer ⇝ *integer*
  Number of different images that are used for the calibration.
  Restriction: NumImages >= 2
- ▷ **ReferenceImage** (input_control) ..... integer ⇝ *integer*
  Index of the reference image.
- ▷ **MappingSource** (input_control) ..... integer-array ⇝ *integer*
  Indices of the source images of the transformations.
- ▷ **MappingDest** (input_control) ..... integer-array ⇝ *integer*
  Indices of the target images of the transformations.
- ▷ **HomMatrices2D** (input_control) ..... hom_mat2d-array ⇝ *real*
  Array of 3×3 projective transformation matrices.
- ▷ **Rows1** (input_control) ..... point.x-array ⇝ *real* / integer
  Row coordinates of corresponding points in the respective source images.
- ▷ **Cols1** (input_control) ..... point.y-array ⇝ *real* / integer
  Column coordinates of corresponding points in the respective source images.
- ▷ **Rows2** (input_control) ..... point.x-array ⇝ *real* / integer
  Row coordinates of corresponding points in the respective destination images.
- ▷ **Cols2** (input_control) ..... point.y-array ⇝ *real* / integer
  Column coordinates of corresponding points in the respective destination images.
- ▷ **NumCorrespondences** (input_control) ..... integer-array ⇝ *integer*
  Number of point correspondences in the respective image pair.
- ▷ **Transformation** (input_control) ..... string ⇝ *string*
  Transformation class to be used.
  Default: 'projective'
  List of values: Transformation ∈ {'projective', 'affine', 'similarity', 'rigid'}
- ▷ **MosaicMatrices2D** (output_control) ..... hom_mat2d-array ⇝ *real*
  Array of 3×3 projective transformation matrices that determine the position of the images in the mosaic.
- ▷ **Rows** (output_control) ..... point.x-array ⇝ *real*
  Row coordinates of the points reconstructed by the bundle adjustment.
- ▷ **Cols** (output_control) ..... point.y-array ⇝ *real*
  Column coordinates of the points reconstructed by the bundle adjustment.
- ▷ **Error** (output_control) ..... real(-array) ⇝ *real*
  Average error per reconstructed point.

Example

```
* Assume that Images contains the four images of the mosaic in the
* layout given in the above description.  Then the following example
* computes the bundle-adjusted transformation matrices.
From := [1,1,1,2,2,3]
To := [2,3,4,3,4,4]
HomMatrices2D := []
Rows1 := []
Cols1 := []
Rows2 := []
Cols2 := []
NumMatches := []
for J := 0 to |From|-1 by 1
    select_obj (Images, ImageF, From[J])
    select_obj (Images, ImageT, To[J])
    points_foerstner (ImageF, 1, 2, 3, 100, 0.1, 'gauss', 'true', \
                      RowsF, ColsF, _, _, _, _, _, _, _, _)
    points_foerstner (ImageT, 1, 2, 3, 100, 0.1, 'gauss', 'true', \
                      RowsT, ColsT, _, _, _, _, _, _, _, _)
    proj_match_points_ransac (ImageF, ImageT, RowsF, ColsF, RowsT, ColsT, \
                              'ncc', 10, 0, 0, 480, 640, 0, 0.5, \
                              'gold_standard', 2, 42, HomMat2D, \
                              Points1, Points2)
    HomMatrices2D := [HomMatrices2D,HomMat2D]
    Rows1 := [Rows1,subset(RowsF,Points1)]
    Cols1 := [Cols1,subset(ColsF,Points1)]
    Rows2 := [Rows2,subset(RowsT,Points2)]
    Cols2 := [Cols2,subset(ColsT,Points2)]
    NumMatches := [NumMatches,|Points1|]
endfor
bundle_adjust_mosaic (4, 1, From, To, HomMatrices2D, Rows1, Cols1, \
                      Rows2, Cols2, NumMatches, 'rigid', MosaicMatrices, \
                      Rows, Columns, Error)
gen_bundle_adjusted_mosaic (Images, MosaicImage, HomMatrices2D, \
                            'default', 'false', TransMat2D)
```

Result

If the parameters are valid, the operator `bundle_adjust_mosaic` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`proj_match_points_ransac`, `proj_match_points_ransac_guided`

Possible Successors

`gen_bundle_adjusted_mosaic`

See also

`gen_projective_mosaic`

Module

Matching

gen_bundle_adjusted_mosaic (Images : MosaicImage : HomMatrices2D, StackingOrder, TransformDomain : TransMat2D)

Combine multiple images into a mosaic image.

gen_bundle_adjusted_mosaic combines the input images contained in the object Images into a mosaic image MosaicImage. The relative positions of the images are defined by 3×3 projective transformation matrices. The array HomMatrices2D contains a sequence of these linearized matrices. The transformation matrices can be computed with bundle_adjust_mosaic.

The origin of MosaicImage and its size are automatically chosen so that all of the input images are completely visible.

The order in which the images are added to the mosaic is given by the array StackingOrder. The first index in this array will end up at the bottom of the image stack while the last one will be on top. If *'default'* is given instead of an array of integers, the canonical order (images in the order used in Images) will be used.

The parameter TransformDomain can be used to determine whether the domains of Images are also transformed. Since the transformation of the domains costs runtime, this parameter should be used to specify whether this is desired or not. If TransformDomain is set to *'false'* the domain of the input images is ignored and the complete images are transformed.

On output, the parameter TransMat2D contains a 3×3 projective transformation matrix that describes the translation that was necessary to transform all images completely into the output image.

Parameters

- ▷ **Images** (input_object) ..... (multichannel-)image-array ⇝ *object :* byte / uint2 / real
  Input images.
- ▷ **MosaicImage** (output_object) ..... (multichannel-)image ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **HomMatrices2D** (input_control) ..... hom_mat2d-array ⇝ *real*
  Array of 3×3 projective transformation matrices.
- ▷ **StackingOrder** (input_control) ..... string(-array) ⇝ *string* / integer
  Stacking order of the images in the mosaic.
  Default: 'default'
  Suggested values: StackingOrder ∈ {'default'}
- ▷ **TransformDomain** (input_control) ..... string ⇝ *string*
  Should the domains of the input images also be transformed?
  Default: 'false'
  List of values: TransformDomain ∈ {'true', 'false'}
- ▷ **TransMat2D** (output_control) ..... hom_mat2d ⇝ *real*
  3×3 projective transformation matrix that describes the translation that was necessary to transform all images completely into the output image.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

bundle_adjust_mosaic

Alternatives

gen_projective_mosaic

See also

projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_point_2d, projective_trans_pixel

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2000.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

Matching

gen_cube_map_mosaic (Images : Front, Rear, Left, Right, Top, Bottom : CameraMatrices, RotationMatrices, CubeMapDimension, StackingOrder, Interpolation :)

Create 6 cube map images of a spherical mosaic.

gen_cube_map_mosaic creates 6 cube map images of a spherical mosaic Front, Left, Rear, Right, Top and Bottom from the input images passed in Images. The pose of the images in space, which is used to compute the position of the images with respect to the surface of the sphere, can be determined with stationary_camera_self_calibration. The camera and rotation matrices computed with stationary_camera_self_calibration can be used in CameraMatrices and RotationMatrices. A spherical mosaic can only be created from images that were taken with a stationary camera (see stationary_camera_self_calibration).

The width and height of the output cube map images can be selected by setting the parameter CubeMapDimension. The value represents the width and height in pixels.

The mode in which the images are added to the mosaic is given by StackingOrder. For StackingOrder $=$ *'voronoi'*, the points in the mosaic image are determined from the Voronoi cell of the respective input image. This means that the gray values are taken from the points of the input image to whose center the pixel in the mosaic image has the smallest distance on the sphere. This mode has the advantage that vignetting and uncorrected radial distortions are less noticeable in the mosaic image because they typically are symmetric with respect to the image center. Alternatively, with the choice of parameters described in the following, a mode can be selected that has the same effect as if the images were painted successively into the mosaic image. Here, the order in which the images are added to the mosaic image is important. Therefore, an array of integer values can be passed in StackingOrder. The first index in this array will end up at the bottom of the image stack while the last one will be on top. If *'default'* is given instead of an array of integers, the canonical order (images in the order used in Images) will be used. Hence, if neither *'voronoi'* nor *'default'* are used, StackingOrder must contain a permutation of the numbers 1,...,n, where n is the number of images passed in Images. It should be noted that the mode *'voronoi'* cannot always be used. For example, at least two images must be passed to use this mode. Furthermore, for very special configurations of the positions of the image centers on the sphere, the Voronoi cells cannot be determined uniquely. With StackingOrder $=$ *'blend'*, an additional mode is available, which blends the images of the mosaic smoothly. This way seams between the images become less apparent. The seam lines between the images are the same as in *'voronoi'*. This mode leads to visually more appealing images, but requires significantly more resources. If the mode *'voronoi'* or *'blend'* cannot be used for whatever reason the mode is switched internally to *'default'* automatically.

The parameter Interpolation selects the desired interpolation mode for creating the cube maps. *'bilinear'* and *'bicubic'* interpolation is available for all modes of StackingOrder. *'nearest_neighbor'* is only available if StackingOrder is set to *'default'* or *'voronoi'*.

Parameters

- ▷ **Images** (input_object) ..... (multichannel-)image-array ⇝ *object :* byte / uint2 / real
  Input images.
- ▷ **Front** (output_object) ..... (multichannel-)image ⇝ *object :* byte / uint2 / real
  Front cube map.
- ▷ **Rear** (output_object) ..... (multichannel-)image ⇝ *object :* byte / uint2 / real
  Rear cube map.
- ▷ **Left** (output_object) ..... (multichannel-)image ⇝ *object :* byte / uint2 / real
  Left cube map.
- ▷ **Right** (output_object) ..... (multichannel-)image ⇝ *object :* byte / uint2 / real
  Right cube map.
- ▷ **Top** (output_object) ..... (multichannel-)image ⇝ *object :* byte / uint2 / real
  Top cube map.

▷ **Bottom** (output_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2 / real
Bottom cube map.

▷ **CameraMatrices** (input_control) . . . . . hom_mat2d-array ⇝ *real*
(Array of) 3×3 projective camera matrices that determine the internal camera parameters.

▷ **RotationMatrices** (input_control) . . . . . hom_mat2d-array ⇝ *real*
Array of 3×3 transformation matrices that determine rotation of the camera in the respective image.

▷ **CubeMapDimension** (input_control) . . . . . number ⇝ *integer*
Width and height of the resulting cube maps.
Default: 1000
Restriction: CubeMapDimension >= 0

▷ **StackingOrder** (input_control) . . . . . string(-array) ⇝ *string* / integer
Mode of adding the images to the mosaic image.
Default: 'voronoi'
Suggested values: `StackingOrder` ∈ {'blend', 'voronoi', 'default'}

▷ **Interpolation** (input_control) . . . . . string ⇝ *string*
Mode of image interpolation.
Default: 'bilinear'
Suggested values: `Interpolation` ∈ {'nearest_neighbor', 'bilinear', 'bicubic'}

Example

```
* For the input data to stationary_camera_self_calibration, please
* refer to the example for stationary_camera_self_calibration.
stationary_camera_self_calibration (4, 640, 480, 1, From, To, \
                                    HomMatrices2D, Rows1, Cols1, \
                                    Rows2, Cols2, NumMatches, \
                                    'gold_standard', \
                                    ['focus','principal_point'], \
                                    'true', CameraMatrix, Kappa, \
                                    RotationMatrices, X, Y, Z, Error)
gen_cube_map_mosaic (Images, Front, Left, Rear, Right, Top, Bottom, \
                     CameraMatrix, RotationMatrices, 1000, 'default', \
                     'bicubic')

* Alternatively, if kappa should be determined, the following calls
* can be made:
stationary_camera_self_calibration (4, 640, 480, 1, From, To, \
                                    HomMatrices2D, Rows1, Cols1, \
                                    Rows2, Cols2, NumMatches, \
                                    'gold_standard', \
                                    ['focus','principal_point','kappa'], \
                                    'true', CameraMatrix, Kappa, \
                                    RotationMatrices, X, Y, Z, Error)
cam_mat_to_cam_par (CameraMatrix, Kappa, 640, 480, CamParam)
change_radial_distortion_cam_par ('fixed', CamParam, 0, CamParOut)
gen_radial_distortion_map (Map, CamParam, CamParOut, 'bilinear')
map_image (Images, Map, ImagesRect)
gen_cube_map_mosaic (Images, Front, Left, Rear, Right, Top, Bottom, \
                     CameraMatrix, RotationMatrices, 1000, 'default', \
                     'bicubic')
```

Result

If the parameters are valid, the operator `gen_cube_map_mosaic` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

stationary_camera_self_calibration

Alternatives

gen_spherical_mosaic, gen_projective_mosaic

References

Lourdes Agapito, E. Hayman, I. Reid: "Self-Calibration of Rotating and Zooming Cameras"; International Journal of Computer Vision; vol. 45, no. 2; pp. 107–127; 2001.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

Matching

gen_projective_mosaic (Images : MosaicImage : StartImage, MappingSource, MappingDest, HomMatrices2D, StackingOrder, TransformDomain : MosaicMatrices2D)

Combine multiple images into a mosaic image.

gen_projective_mosaic combines the input images contained in the object Images into a mosaic image MosaicImage. The relative positions of the images are defined by 3×3 projective transformation matrices. The array HomMatrices2D contains a sequence of these linearized matrices. The values in MappingSource and MappingDest are the indices of the images that the corresponding matrix applies to. MappingSource=*4* and MappingDest=*7* means that the matrix describes the transformation of the image number *4* into the projective plane of image *7*. The transformation matrices between the respective image pairs given by MappingSource and MappingDest are typically determined with proj_match_points_ransac.

As usual for operators that access image objects (e.g., select_obj), the images are numbered starting from 1, i.e., MappingSource, MappingDest, StartImage, and StackingOrder) must contain values between 1 and the number of images passed in Images.

The parameter StartImage states which image defines the image plane of the final image, that is, which input image remains unchanged in the output image. This is usually an image that is located near the center of the image mosaic.

The origin of MosaicImage and its size are automatically chosen so that all of the input images are completely visible.

The order in which the images are added to the mosaic is given by the array StackingOrder. The first index in this array will end up at the bottom of the image stack while the last one will be on top. If *'default'* is given instead of an array of integers, the canonical order (images in the order used in Images) will be used.

The parameter TransformDomain can be used to determine whether the domains of Images are also transformed. Since the transformation of the domains costs runtime, this parameter should be used to specify whether this is desired or not. If TransformDomain is set to *'false'* the domain of the input images is ignored and the complete images are transformed.

On output, the parameter MosaicMatrices2D contains a set of 3×3 projective transformation matrices that describe for each image in Images the mapping of the image to its position in the mosaic.

Parameters

▷ **Images** (input_object) . . . . . (multichannel-)image-array ⇝ *object :* byte / uint2 / real
Input images.

▷ **MosaicImage** (output_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2 / real
Output image.

▷ **StartImage** (input_control) . . . . . integer ⇝ *integer*
Index of the central input image.

▷ **MappingSource** (input_control) . . . . . integer-array ⇝ *integer*
Indices of the source images of the transformations.

▷ **MappingDest** (input_control) . . . . . integer-array ⇝ *integer*
Indices of the target images of the transformations.

▷ **HomMatrices2D** (input_control) . . . . . hom_mat2d-array ⇝ *real*
Array of 3×3 projective transformation matrices.

▷ **StackingOrder** (input_control) . . . . . string(-array) ⇝ *string* / integer
Stacking order of the images in the mosaic.
Default: 'default'
Suggested values: StackingOrder ∈ {'default'}

▷ **TransformDomain** (input_control) . . . . . string ⇝ *string*
Should the domains of the input images also be transformed?
Default: 'false'
List of values: TransformDomain ∈ {'true', 'false'}

▷ **MosaicMatrices2D** (output_control) . . . . . hom_mat2d-array ⇝ *real*
Array of 3×3 projective transformation matrices that determine the position of the images in the mosaic.

Example

```
gen_empty_obj (Images)
for J := 1 to 6 by 1
    read_image (Image, 'mosaic/pcb_'+J$'02')
    concat_obj (Images, Image, Images)
endfor
From := [1,2,3,4,5]
To := [2,3,4,5,6]
Num := |From|
ProjMatrices := []
for J := 0 to Num-1 by 1
    F := From[J]
    T := To[J]
    select_obj (Images, ImageF, F)
    select_obj (Images, ImageT, T)
    points_foerstner (ImageF, 1, 2, 3, 200, 0.3, 'gauss', 'false', \
                      RowJunctionsF, ColJunctionsF, CoRRJunctionsF, \
                      CoRCJunctionsF, CoCCJunctionsF, RowAreaF, \
                      ColAreaF, CoRRAreaF, CoRCAreaF, CoCCAreaF)
    points_foerstner (ImageT, 1, 2, 3, 200, 0.3, 'gauss', 'false', \
                      RowJunctionsT, ColJunctionsT, CoRRJunctionsT, \
                      CoRCJunctionsT, CoCCJunctionsT, RowAreaT, \
                      ColAreaT, CoRRAreaT, CoRCAreaT, CoCCAreaT)
    proj_match_points_ransac (ImageF, ImageT, RowJunctionsF, \
                               ColJunctionsF, RowJunctionsT, \
                               ColJunctionsT, 'ncc', 21, 0, 0, 480, 640, \
                               0, 0.5, 'gold_standard', 1, 4364537, \
                               ProjMatrix, Points1, Points2)
    ProjMatrices := [ProjMatrices,ProjMatrix]
endfor
gen_projective_mosaic (Images, MosaicImage, 2, From, To, ProjMatrices, \
                       'default', 'false', MosaicMatrices2D)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

proj_match_points_ransac, proj_match_points_ransac_guided, vector_to_proj_hom_mat2d, hom_vector_to_proj_hom_mat2d

See also

projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_point_2d, projective_trans_pixel

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2000.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

Matching

gen_spherical_mosaic (Images : MosaicImage : CameraMatrices, RotationMatrices, LatMin, LatMax, LongMin, LongMax, LatLongStep, StackingOrder, Interpolation :)

Create a spherical mosaic image.

gen_spherical_mosaic creates a spherical mosaic image MosaicImage from the input images passed in Images. The pose of the images in space, which is used to compute the position of the images with respect to the surface of the sphere, can be determined with stationary_camera_self_calibration. The camera and rotation matrices computed with stationary_camera_self_calibration can be used in CameraMatrices and RotationMatrices. A spherical mosaic can only be created from images that were taken with a stationary camera (see stationary_camera_self_calibration).

The mosaic is computed in spherical coordinates (longitude and latitude). The row axis of MosaicImage corresponds to the latitude, while the column axis corresponds to the longitude. The part of the sphere that is computed by gen_spherical_mosaic is determined by LatMin, LatMax, LongMin, and LongMax. These parameters are specified in degrees and determine a rectangular part of the latitude and longitude coordinates. The latitude *-90* corresponds to the north pole (i.e., the straight up viewing direction), while *90* corresponds to the south pole (i.e., the straight down viewing direction). The longitude *0* corresponds to the straight ahead viewing direction. Negative longitudes correspond to viewing directions to the left, while positive longitudes correspond to viewing directions to the right. Hence, to obtain a complete image of the sphere, LatMin = *-90*, LatMax = *90*, LongMin = *-180*, and LongMax = *180* must be used. In many cases, the mosaic will not cover the entire sphere. In these cases, it is useful to select the desired part of the sphere with the above parameters. This can be done by explicitly specifying the desired rectangle. However, often it is desirable to determine the smallest rectangle that encloses all images automatically. This can be done by using LatMin < *-90*, LatMax > *90*, LongMin < *-180*, and LongMax > *180*. Only the parameters that lie outside the normal range of values are determined automatically.

The angle step per pixel in MosaicImage can be selected with LatLongStep, which also is an angle specified in degrees. With this, the resolution of the mosaic image can be controlled. If LatLongStep is set to *0* the angle step is calculated automatically by trying to preserve the pixel size of the original images as well as possible.

The mode in which the images are added to the mosaic is given by StackingOrder. For StackingOrder = *'voronoi'*, the points in the mosaic image are determined from the Voronoi cell of the respective input image. This means that the gray values are taken from the points of the input image to whose center the pixel in the mosaic image has the smallest distance on the sphere. This mode has the advantage that vignetting and uncorrected radial distortions are less noticeable in the mosaic image because they typically are symmetric with respect to the image center. Alternatively, with the choice of parameters described described in the following, a mode can be selected that has the same effect as if the images were painted successively into the mosaic image. Here, the order in which the images are added to the mosaic image is important. Therefore, an array of integer values can be passed in StackingOrder. The first index in this array will end up at the bottom of the image stack while the last one will be on top. If *'default'* is given instead of an array of integers, the canonical order (images in the order used in Images) will be used. Hence, if neither *'voronoi'* nor *'default'* are used, StackingOrder must contain a permutation of the numbers 1,...,n, where n is the number of images passed in Images. It should be noted that the mode *'voronoi'* cannot always be used. For example, at least two images must be passed to use this mode. Furthermore, for very special configurations of the positions of the image centers on the sphere, the Voronoi cells cannot be determined uniquely. With StackingOrder = *'blend'*, an additional mode is available, which blends the images of the mosaic smoothly. This way seams between the images become less apparent. The seam lines

between the images are the same as in *'voronoi'*. This mode leads to visually more appealing images, but requires significantly more resources. If the mode *'voronoi'* or *'blend'* cannot be used for whatever reason the mode is switched internally to *'default'* automatically.

The parameter `Interpolation` selects the desired interpolation mode for creating the mosaic. *'bilinear'* and *'bicubic'* interpolation is available for all modes of `StackingOrder`. *'nearest_neighbor'* is only available if `StackingOrder` is set to *'default'* or *'voronoi'*.

Parameters

- ▷ **`Images`** (input_object) . . . . . (multichannel-)image-array ⇝ *object :* byte / uint2 / real
  Input images.
- ▷ **`MosaicImage`** (output_object) . . . . . (multichannel-)image ⇝ *object :* byte / uint2 / real
  Output image.
- ▷ **`CameraMatrices`** (input_control) . . . . . hom_mat2d-array ⇝ *real*
  (Array of) 3×3 projective camera matrices that determine the internal camera parameters.
- ▷ **`RotationMatrices`** (input_control) . . . . . hom_mat2d-array ⇝ *real*
  Array of 3×3 transformation matrices that determine rotation of the camera in the respective image.
- ▷ **`LatMin`** (input_control) . . . . . angle.deg ⇝ *real* / integer
  Minimum latitude of points in the spherical mosaic image.
  Default: -90
  Suggested values: `LatMin` ∈ {-100, -90, -80, -70, -60, -50, -40, -30, -20, -10}
  Restriction: LatMin <= 90
- ▷ **`LatMax`** (input_control) . . . . . angle.deg ⇝ *real* / integer
  Maximum latitude of points in the spherical mosaic image.
  Default: 90
  Suggested values: `LatMax` ∈ {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
  Restriction: LatMax >= -90 && LatMax > LatMin
- ▷ **`LongMin`** (input_control) . . . . . angle.deg ⇝ *real* / integer
  Minimum longitude of points in the spherical mosaic image.
  Default: -180
  Suggested values: `LongMin` ∈ {-200, -180, -160, -140, -120, -100, -90, -80, -70, -60, -50, -40, -30, -20, -10}
  Restriction: LongMin <= 180
- ▷ **`LongMax`** (input_control) . . . . . angle.deg ⇝ *real* / integer
  Maximum longitude of points in the spherical mosaic image.
  Default: 180
  Suggested values: `LongMax` ∈ {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 120, 140, 160, 180, 200}
  Restriction: LongMax >= -90 && LongMax > LongMin
- ▷ **`LatLongStep`** (input_control) . . . . . angle.deg ⇝ *real* / integer
  Latitude and longitude angle step width.
  Default: 0.1
  Suggested values: `LatLongStep` ∈ {0, 0.02, 0.05, 0.1, 0.2, 0.5, 1}
  Restriction: LatLongStep >= 0
- ▷ **`StackingOrder`** (input_control) . . . . . string(-array) ⇝ *string* / integer
  Mode of adding the images to the mosaic image.
  Default: 'voronoi'
  Suggested values: `StackingOrder` ∈ {'blend', 'voronoi', 'default'}
- ▷ **`Interpolation`** (input_control) . . . . . string ⇝ *string* / integer
  Mode of interpolation when creating the mosaic image.
  Default: 'bilinear'
  Suggested values: `Interpolation` ∈ {'nearest_neighbor', 'bilinear', 'bicubic'}

Example

```
* For the input data to stationary_camera_self_calibration, please
* refer to the example for stationary_camera_self_calibration.
stationary_camera_self_calibration (4, 640, 480, 1, From, To, \
                                    HomMatrices2D, Rows1, Cols1, \
                                    Rows2, Cols2, NumMatches, \
                                    'gold_standard', \
```

```
                                    ['focus','principal_point'], \
                                    'true', CameraMatrix, Kappa, \
                                    RotationMatrices, X, Y, Z, Error)
gen_spherical_mosaic (Images, MosaicImage, CameraMatrix, \
                      RotationMatrices, -100, 100, -200, 200, 0, \
                      'default','bilinear')

* Alternatively, if kappa should be determined, the following calls
* can be made:
stationary_camera_self_calibration (4, 640, 480, 1, From, To, \
                                    HomMatrices2D, Rows1, Cols1, \
                                    Rows2, Cols2, NumMatches, \
                                    'gold_standard', \
                                    ['focus','principal_point','kappa'], \
                                    'true', CameraMatrix, Kappa, \
                                    RotationMatrices, X, Y, Z, Error)
cam_mat_to_cam_par (CameraMatrix, Kappa, 640, 480, CamParam)
change_radial_distortion_cam_par ('fixed', CamParam, 0, CamParOut)
gen_radial_distortion_map (Map, CamParam, CamParOut, 'bilinear')
map_image (Images, Map, ImagesRect)
gen_spherical_mosaic (ImagesRect, MosaicImage, CameraMatrix, \
                      RotationMatrices, -100, 100, -200, 200, 0, \
                      'default','bilinear')
```

Result

If the parameters are valid, the operator `gen_spherical_mosaic` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`stationary_camera_self_calibration`

Alternatives

`gen_cube_map_mosaic`, `gen_projective_mosaic`

References

Lourdes Agapito, E. Hayman, I. Reid: “Self-Calibration of Rotating and Zooming Cameras”; International Journal of Computer Vision; vol. 45, no. 2; pp. 107–127; 2001.

Olivier Faugeras, Quang-Tuan Luong: “The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications”; MIT Press, Cambridge, MA; 2001.

Module

Matching

proj_match_points_distortion_ransac (Image1, Image2 : : Rows1, Cols1, Rows2, Cols2, GrayMatchMethod, MaskSize, RowMove, ColMove, RowTolerance, ColTolerance, Rotation, MatchThreshold, EstimationMethod, DistanceThreshold, RandSeed : HomMat2D, Kappa, Error, Points1, Points2)

Compute a projective transformation matrix between two images and the radial distortion coefficient by automatically finding correspondences between points.

Given a set of coordinates of characteristic points (Rows1, Cols1) and (Rows2, Cols2) in both input images Image1 and Image2, which must be of identical size, proj_match_points_distortion_ransac automatically determines corresponding points, the homogeneous projective transformation matrix HomMat2D, and the radial distortion coefficient Kappa (κ) that optimally fulfill the following equation:

$$\begin{pmatrix} r_2 \\ c_2 \\ 1 \end{pmatrix} = \texttt{HomMat2D} \cdot \begin{pmatrix} r_1 \\ c_1 \\ 1 \end{pmatrix}.$$

Here, (r_1, c_1) and (r_2, c_2) denote image points that are obtained by undistorting the input image points with the division model (see Calibration):

$$r = \frac{\tilde{r}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)} \qquad c = \frac{\tilde{c}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)}$$

Here, $(\tilde{r}_1, \tilde{c}_1) = (\texttt{Rows1} - 0.5(h-1), \texttt{Cols1} - 0.5(w-1))$
and $(\tilde{r}_2, \tilde{c}_2) = (\texttt{Rows2} - 0.5(h-1), \texttt{Cols2} - 0.5(w-1))$
denote the distorted image points, specified relative to the image center, and w and h denote the width and height of the input images. Thus, proj_match_points_distortion_ransac assumes that the principal point of the camera, i.e., the center of the radial distortions, lies at the center of the image.

The returned Kappa can be used to construct camera parameters that can be used to rectify images or points (see change_radial_distortion_cam_par, change_radial_distortion_image, and change_radial_distortion_points):

$$\begin{aligned} \texttt{CamPar} \quad = \quad & ['area\_scan\_telecentric\_division', 1.0, \texttt{Kappa}, 1.0, 1.0, \\ & 0.5(w-1), 0.5(h-1), w, h] \end{aligned}$$

The matching process is based on characteristic points, which can be extracted with point operators like points_foerstner or points_harris. The matching itself is carried out in two steps: first, gray value correlations of mask windows around the input points in the first and the second image are determined and an initial matching between them is generated using the similarity of the windows in both images. Then, the RANSAC algorithm is applied to find the projective transformation matrix and radial distortion coefficient that maximizes the number of correspondences under the above constraint.

The size of the mask windows used for the matching is MaskSize×MaskSize. Three metrics for the correlation can be selected. If GrayMatchMethod has the value *'ssd'*, the sum of the squared gray value differences is used, *'sad'* means the sum of absolute differences, and *'ncc'* is the normalized cross correlation. For details please refer to binocular_disparity. The metric is minimized (*'ssd'*, *'sad'*) or maximized (*'ncc'*) over all possible point pairs. A thus found matching is only accepted if the value of the metric is below the value of MatchThreshold (*'ssd'*, *'sad'*) or above that value (*'ncc'*).

To increase the algorithm's performance, the search area for the match candidates can be limited to a rectangle by specifying its size and offset. Only points within a window of $2 \cdot$ RowTolerance $\times 2 \cdot$ ColTolerance points are considered. The offset of the center of the search window in the second image with respect to the position of the current point in the first image is given by RowMove and ColMove.

If the transformation contains a rotation, i.e., if the first image is rotated with respect to the second image, the parameter Rotation may contain an estimate for the rotation angle or an angle interval in radians. A good guess will increase the quality of the gray value matching. If the actual rotation differs too much from the specified estimate, the matching will typically fail. In this case, an angle interval should be specified and Rotation is a tuple with two elements. The larger the given interval is the slower the operator is since the RANSAC algorithm is run over all (automatically determined) angle increments within the interval.

After the initial matching has been completed, a randomized search algorithm (RANSAC) is used to determine the projective transformation matrix HomMat2D and the radial distortion coefficient Kappa. It tries to find the parameters that are consistent with a maximum number of correspondences. For a point to be accepted, the distance in pixels to its corresponding transformed point must not exceed the threshold DistanceThreshold.

The parameter EstimationMethod determines which algorithm is used to compute the projective transformation matrix. A linear algorithm is used if EstimationMethod is set to *'linear'*. This algorithm is very

fast and returns accurate results for small to moderate noise of the point coordinates and for most distortions (except for small distortions). For `EstimationMethod` = *'gold_standard'*, a mathematically optimal but slower optimization is used, which minimizes the geometric reprojection error. In general, it is preferable to use `EstimationMethod` = *'gold_standard'*.

The value `Error` indicates the overall quality of the estimation procedure and is the mean symmetric euclidean distance in pixels between the points and their corresponding transformed points.

Point pairs consistent with the above constraints are considered to be corresponding points. `Points1` contains the indices of the matched input points from the first image and `Points2` contains the indices of the corresponding points in the second image.

The parameter `RandSeed` can be used to control the randomized nature of the RANSAC algorithm, and hence to obtain reproducible results. If `RandSeed` is set to a positive number, the operator returns the same result on every call with the same parameters because the internally used random number generator is initialized with `RandSeed`. If `RandSeed` = *0*, the random number generator is initialized with the current time. In this case the results may not be reproducible. The value set for the HALCON system variable *'seed_rand'* (see `set_system`) does not affect the results of `proj_match_points_distortion_ransac`.

Parameters

▷ **`Image1`** (input_object) . . . singlechannelimage ⇝ *object :* byte / uint2
Input image 1.

▷ **`Image2`** (input_object) . . . singlechannelimage ⇝ *object :* byte / uint2
Input image 2.

▷ **`Rows1`** (input_control) . . . point.y-array ⇝ *real* / integer
Input points in image 1 (row coordinate).
Restriction: length(Rows1) >= 5

▷ **`Cols1`** (input_control) . . . point.x-array ⇝ *real* / integer
Input points in image 1 (column coordinate).
Restriction: length(Cols1) == length(Rows1)

▷ **`Rows2`** (input_control) . . . point.y-array ⇝ *real* / integer
Input points in image 2 (row coordinate).
Restriction: length(Rows2) >= 5

▷ **`Cols2`** (input_control) . . . point.x-array ⇝ *real* / integer
Input points in image 2 (column coordinate).
Restriction: length(Cols2) == length(Rows2)

▷ **`GrayMatchMethod`** (input_control) . . . string ⇝ *string*
Gray value match metric.
Default: 'ncc'
List of values: `GrayMatchMethod` ∈ {'ncc', 'ssd', 'sad'}

▷ **`MaskSize`** (input_control) . . . integer ⇝ *integer*
Size of gray value masks.
Default: 10
Suggested values: `MaskSize` ∈ {3, 7, 15}
Value range: 1 ≤ MaskSize

▷ **`RowMove`** (input_control) . . . integer ⇝ *integer*
Average row coordinate offset of corresponding points.
Default: 0

▷ **`ColMove`** (input_control) . . . integer ⇝ *integer*
Average column coordinate offset of corresponding points.
Default: 0

▷ **`RowTolerance`** (input_control) . . . integer ⇝ *integer*
Half height of matching search window.
Default: 200
Restriction: RowTolerance >= 1

▷ **`ColTolerance`** (input_control) . . . integer ⇝ *integer*
Half width of matching search window.
Default: 200
Restriction: ColTolerance >= 1

▷ **Rotation** (input_control) ..... angle.rad(-array) ⇝ *real* / integer
Estimate of the relative rotation of the second image with respect to the first image.
Default: 0.0
Suggested values: Rotation ∈ {0.0, 0.1, -0.1, 0.7854, 1.571, 3.142}

▷ **MatchThreshold** (input_control) ..... number ⇝ *integer* / real
Threshold for gray value matching.
Default: 0.7
Suggested values: MatchThreshold ∈ {0.9, 0.7, 0.5, 10, 20, 50, 100}

▷ **EstimationMethod** (input_control) ..... string ⇝ *string*
Algorithm for the computation of the projective transformation matrix.
Default: 'gold_standard'
List of values: EstimationMethod ∈ {'linear', 'gold_standard'}

▷ **DistanceThreshold** (input_control) ..... number ⇝ *real* / integer
Threshold for the transformation consistency check.
Default: 1
Restriction: DistanceThreshold > 0

▷ **RandSeed** (input_control) ..... integer ⇝ *integer*
Seed for the random number generator.
Default: 0

▷ **HomMat2D** (output_control) ..... hom_mat2d ⇝ *real*
Computed homogeneous projective transformation matrix.

▷ **Kappa** (output_control) ..... real ⇝ *real*
Computed radial distortion coefficient.

▷ **Error** (output_control) ..... real ⇝ *real*
Root-Mean-Square transformation error.

▷ **Points1** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 1.

▷ **Points2** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 2.

Example

```
points_foerstner (Image1, 1, 2, 3, 50, 0.1, 'gauss', 'true', \
                  Rows1, Cols1, _, _, _, _, _, _, _, _)
points_foerstner (Image2, 1, 2, 3, 50, 0.1, 'gauss', 'true', \
                  Rows2, Cols2, _, _, _, _, _, _, _, _)
get_image_size (Image1, Width, Height)
proj_match_points_distortion_ransac (Image1, Image2, Rows1, Cols1, \
                                     Rows2, Cols2, 'ncc', 10, 0, 0, \
                                     Height, Width, 0, 0.5, \
                                     'gold_standard', 1, 42, \
                                     HomMat2D, Kappa, Error, \
                                     Points1, Points2)
CamParDist := ['area_scan_division',0.0,Kappa,1.0,1.0, \
               0.5*(Width-1),0.5*(Height-1),Width,Height]
change_radial_distortion_cam_par ('fixed', CamParDist, 0, CamPar)
change_radial_distortion_image (Image1, Image1, Image1Rect, \
                                CamParDist, CamPar)
change_radial_distortion_image (Image2, Image2, Image2Rect, \
                                CamParDist, CamPar)
concat_obj (Image1Rect, Image2Rect, ImagesRect)
gen_projective_mosaic (ImagesRect, MosaicImage, 1, 1, 2, HomMat2D, \
                       'default', 'false', MosaicMatrices2D)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_foerstner, points_harris

Possible Successors

vector_to_proj_hom_mat2d_distortion, change_radial_distortion_cam_par, change_radial_distortion_image, change_radial_distortion_points, gen_binocular_proj_rectification, projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_point_2d, projective_trans_pixel

Alternatives

proj_match_points_distortion_ransac_guided

See also

proj_match_points_ransac, proj_match_points_ransac_guided, hom_vector_to_proj_hom_mat2d, vector_to_proj_hom_mat2d

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

Matching

proj_match_points_distortion_ransac_guided (Image1, Image2 : : Rows1, Cols1, Rows2, Cols2, GrayMatchMethod, MaskSize, HomMat2DGuide, KappaGuide, DistanceTolerance, MatchThreshold, EstimationMethod, DistanceThreshold, RandSeed : HomMat2D, Kappa, Error, Points1, Points2)

Compute a projective transformation matrix and the radial distortion coefficient between two images by finding correspondences between points based on known approximations of the projective transformation matrix and the radial distortion coefficient.

Given a set of coordinates of characteristic points (Rows1, Cols1) and (Rows2Cols2) in both input images Image1 and Image2, which must have identical size, and given known approximations HomMat2DGuide and KappaGuide for the transformation matrix and the radial distortion coefficient between Image1 and Image2, proj_match_points_distortion_ransac_guided automatically determines corresponding points, the homogeneous projective transformation matrix HomMat2D, and the radial distortion coefficient Kappa (κ) that optimally fulfill the following equation:

$$\begin{pmatrix} r_2 \\ c_2 \\ 1 \end{pmatrix} = \texttt{HomMat2D} \cdot \begin{pmatrix} r_1 \\ c_1 \\ 1 \end{pmatrix}.$$

Here, (r_1, c_1) and (r_2, c_2) denote image points that are obtained by undistorting the input image points with the division model (see Calibration):

$$r = \frac{\tilde{r}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)} \qquad c = \frac{\tilde{c}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)}$$

Here, $(\tilde{r}_1, \tilde{c}_1) = (\texttt{Rows1} - 0.5(h-1), \texttt{Cols1} - 0.5(w-1))$
and $(\tilde{r}_2, \tilde{c}_2) = (\texttt{Rows2} - 0.5(h-1), \texttt{Cols2} - 0.5(w-1))$

denote the distorted image points, specified relative to the image center, and w and h denote the width and height of the input images. Thus, proj_match_points_distortion_ransac_guided assumes that the principal point of the camera, i.e., the center of the radial distortions, lies at the center of the image.

The returned Kappa can be used to construct camera parameters that can be used to rectify images or points (see change_radial_distortion_cam_par, change_radial_distortion_image, and change_radial_distortion_points):

$$\begin{aligned} \texttt{CamPar} \quad = \quad & [\textit{'area\_scan\_telecentric\_division'}, 1.0, \texttt{Kappa}, 1.0, 1.0, \\ & 0.5(w-1), 0.5(h-1), w, h] \end{aligned}$$

The approximations HomMat2DGuide and KappaGuide can, for example, be calculated with proj_match_points_distortion_ransac on lower resolution versions of Image1 and Image2. See the example below.

The matching process is based on characteristic points, which can be extracted with point operators like points_foerstner or points_harris. The matching itself is carried out in two steps: first, gray value correlations of mask windows around the input points in the first and the second image are determined and an initial matching between them is generated using the similarity of the windows in both images. Then, the RANSAC algorithm is applied to find the projective transformation matrix and radial distortion coefficient that maximizes the number of correspondences under the above constraint.

The size of the mask windows used for the matching is MaskSize×MaskSize. Three metrics for the correlation can be selected. If GrayMatchMethod has the value *'ssd'*, the sum of the squared gray value differences is used, *'sad'* means the sum of absolute differences, and *'ncc'* is the normalized cross correlation. For details please refer to binocular_disparity. The metric is minimized (*'ssd'*, *'sad'*) or maximized (*'ncc'*) over all possible point pairs. A thus found matching is only accepted if the value of the metric is below the value of MatchThreshold (*'ssd'*, *'sad'*) or above that value (*'ncc'*).

To increase the algorithm's performance, the search area for the match candidates is limited based on the approximate transformation specified by HomMat2DGuide and KappaGuide. Only points within a distance of DistanceTolerance around the point in Image2 that is obtained when transforming a point in Image1 via HomMat2DGuide and KappaGuide are considered for the matching.

After the initial matching has been completed, a randomized search algorithm (RANSAC) is used to determine the projective transformation matrix HomMat2D and the radial distortion coefficient Kappa. It tries to find the parameters that are consistent with a maximum number of correspondences. For a point to be accepted, the distance to its corresponding transformed point must not exceed the threshold DistanceThreshold. Consequently, DistanceThreshold should be smaller than DistanceTolerance.

The parameter EstimationMethod determines which algorithm is used to compute the projective transformation matrix. A linear algorithm is used if EstimationMethod is set to *'linear'*. This algorithm is very fast and returns accurate results for small to moderate noise of the point coordinates and for most distortions (except for small distortions). For EstimationMethod = *'gold_standard'*, a mathematically optimal but slower optimization is used, which minimizes the geometric reprojection error. In general, it is preferable to use EstimationMethod = *'gold_standard'*.

The value Error indicates the overall quality of the estimation procedure and is the mean symmetric euclidean distance in pixels between the points and their corresponding transformed points.

Point pairs consistent with the above constraints are considered to be corresponding points. Points1 contains the indices of the matched input points from the first image and Points2 contains the indices of the corresponding points in the second image.

The parameter RandSeed can be used to control the randomized nature of the RANSAC algorithm, and hence to obtain reproducible results. If RandSeed is set to a positive number, the operator returns the same result on every call with the same parameters because the internally used random number generator is initialized with RandSeed. If RandSeed = *0*, the random number generator is initialized with the current time. In this case the results may not be reproducible. The value set for the HALCON system variable *'seed_rand'* (see set_system) does not affect the results of proj_match_points_distortion_ransac_guided.

Parameters

▷ **Image1** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image 1.

▷ **Image2** (input_object) . . . . . singlechannelimage ⇝ *object :* byte / uint2
Input image 2.

▷ **Rows1** (input_control) . . . . . point.y-array ⇝ *real* / integer
Input points in image 1 (row coordinate).
Restriction: length(Rows1) >= 5

▷ **Cols1** (input_control) . . . . . point.x-array ⇝ *real* / integer
Input points in image 1 (column coordinate).
Restriction: length(Cols1) == length(Rows1)

▷ **Rows2** (input_control) . . . . . point.y-array ⇝ *real* / integer
Input points in image 2 (row coordinate).
Restriction: length(Rows2) >= 5

▷ **Cols2** (input_control) . . . . . point.x-array ⇝ *real* / integer
Input points in image 2 (column coordinate).
Restriction: length(Cols2) == length(Rows2)

▷ **GrayMatchMethod** (input_control) . . . . . string ⇝ *string*
Gray value match metric.
Default: 'ncc'
List of values: `GrayMatchMethod` ∈ {'ncc', 'ssd', 'sad'}

▷ **MaskSize** (input_control) . . . . . integer ⇝ *integer*
Size of gray value masks.
Default: 10
Suggested values: `MaskSize` ∈ {3, 7, 15}
Value range: 1 ≤ MaskSize

▷ **HomMat2DGuide** (input_control) . . . . . hom_mat2d ⇝ *real*
Approximation of the homogeneous projective transformation matrix between the two images.

▷ **KappaGuide** (input_control) . . . . . real ⇝ *real*
Approximation of the radial distortion coefficient in the two images.

▷ **DistanceTolerance** (input_control) . . . . . real ⇝ *real*
Tolerance for the matching search window.
Default: 20.0
Suggested values: `DistanceTolerance` ∈ {0.2, 0.5, 1.0, 2.0, 3.0, 5.0, 10.0, 20.0, 50.0}
Restriction: DistanceTolerance > 0

▷ **MatchThreshold** (input_control) . . . . . number ⇝ *integer* / real
Threshold for gray value matching.
Default: 0.7
Suggested values: `MatchThreshold` ∈ {0.9, 0.7, 0.5, 10, 20, 50, 100}

▷ **EstimationMethod** (input_control) . . . . . string ⇝ *string*
Algorithm for the computation of the projective transformation matrix.
Default: 'gold_standard'
List of values: `EstimationMethod` ∈ {'linear', 'gold_standard'}

▷ **DistanceThreshold** (input_control) . . . . . number ⇝ *real* / integer
Threshold for transformation consistency check.
Default: 1
Restriction: DistanceThreshold > 0

▷ **RandSeed** (input_control) . . . . . integer ⇝ *integer*
Seed for the random number generator.
Default: 0

▷ **HomMat2D** (output_control) . . . . . hom_mat2d ⇝ *real*
Computed homogeneous projective transformation matrix.

▷ **Kappa** (output_control) . . . . . real ⇝ *real*
Computed radial distortion coefficient.

▷ **Error** (output_control) . . . . . real ⇝ *real*
Root-Mean-Square transformation error.

▷ **Points1** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 1.

▷ **Points2** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 2.

Example

```
Factor := 0.5
zoom_image_factor (Image1, Image1Zoomed, Factor, Factor, 'constant')
zoom_image_factor (Image2, Image2Zoomed, Factor, Factor, 'constant')
points_foerstner (Image1Zoomed, 1, 2, 3, 200, 0.3, 'gauss', 'true', \
                  Rows1, Cols1, _, _, _, _, _, _, _, _)
points_foerstner (Image2Zoomed, 1, 2, 3, 200, 0.3, 'gauss', 'true', \
                  Rows2, Cols2, _, _, _, _, _, _, _, _)
get_image_size (Image1Zoomed, Width, Height)
proj_match_points_distortion_ransac (Image1Zoomed, Image2Zoomed, \
                                     Rows1, Cols1, Rows2, Cols2, \
                                     'ncc', 10, 0, 0, Height, Width, \
                                     0, 0.5, 'gold_standard', 2, 0, \
                                     HomMat2D, Kappa, Error, \
                                     Points1, Points2)
hom_mat2d_scale_local (HomMat2D, Factor, Factor, HomMat2DGuide)
hom_mat2d_scale (HomMat2DGuide, 1.0/Factor, 1.0/Factor, 0, 0, \
                 HomMat2DGuide)
KappaGuide := Kappa*Factor*Factor
points_foerstner (Image1, 1, 2, 3, 200, 0.3, 'gauss', 'true', \
                  Rows1, Cols1, _, _, _, _, _, _, _, _)
points_foerstner (Image2, 1, 2, 3, 200, 0.3, 'gauss', 'true', \
                  Rows2, Cols2, _, _, _, _, _, _, _, _)
proj_match_points_distortion_ransac_guided (Image1, Image2, \
                                            Rows1, Cols1, \
                                            Rows2, Cols2, \
                                            'ncc', 10, \
                                            HomMat2DGuide, \
                                            KappaGuide, 5, 0.5, \
                                            'gold_standard', 2, 0, \
                                            HomMat2D, Kappa, \
                                            Error, Points1, Points2)
get_image_size (Image1, Width, Height)
CamParDist := ['area_scan_division',0.0,Kappa,1.0,1.0, \
               0.5*(Width-1),0.5*(Height-1),Width,Height]
change_radial_distortion_cam_par ('fixed', CamParDist, 0, CamPar)
change_radial_distortion_image (Image1, Image1, Image1Rect, \
                                CamParDist, CamPar)
change_radial_distortion_image (Image2, Image2, Image2Rect, \
                                CamParDist, CamPar)
concat_obj (Image1Rect, Image2Rect, ImagesRect)
gen_projective_mosaic (ImagesRect, MosaicImage, 1, 1, 2, HomMat2D, \
                       'default', 'false', MosaicMatrices2D)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_foerstner, points_harris

Possible Successors

vector_to_proj_hom_mat2d_distortion, change_radial_distortion_cam_par, change_radial_distortion_image, change_radial_distortion_points, gen_binocular_proj_rectification, projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_point_2d, projective_trans_pixel

Alternatives

proj_match_points_distortion_ransac

See also

proj_match_points_ransac, proj_match_points_ransac_guided, hom_vector_to_proj_hom_mat2d, vector_to_proj_hom_mat2d

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

Matching

proj_match_points_ransac (Image1, Image2 : : Rows1, Cols1, Rows2, Cols2, GrayMatchMethod, MaskSize, RowMove, ColMove, RowTolerance, ColTolerance, Rotation, MatchThreshold, EstimationMethod, DistanceThreshold, RandSeed : HomMat2D, Points1, Points2)

Compute a projective transformation matrix between two images by finding correspondences between points.

Given a set of coordinates of characteristic points (Cols1, Rows1) and (Cols2, Rows2) in both input images Image1 and Image2, proj_match_points_ransac automatically determines corresponding points and the homogeneous projective transformation matrix HomMat2D that best transforms the corresponding points from the different images into each other. The characteristic points can, for example, be extracted with points_foerstner or points_harris.

The transformation is determined in two steps: First, gray value correlations of mask windows around the input points in the first and the second image are determined and an initial matching between them is generated using the similarity of the windows in both images.

The size of the mask windows is MaskSize $\times$ MaskSize. Three metrics for the correlation can be selected. If GrayMatchMethod has the value *'ssd'*, the sum of the squared gray value differences is used, *'sad'* means the sum of absolute differences, and *'ncc'* is the normalized cross correlation. For details please refer to binocular_disparity. The metric is minimized (*'ssd'*, *'sad'*) or maximized (*'ncc'*) over all possible point pairs. A thus found matching is only accepted if the value of the metric is below the value of MatchThreshold (*'ssd'*, *'sad'*) or above that value (*'ncc'*).

To increase the algorithm's performance, the search area for the matching operations can be limited. Only points within a window of $2 \cdot$ RowTolerance $\times 2 \cdot$ ColTolerance points are considered. The offset of the center of the search window in the second image with respect to the position of the current point in the first image is given by RowMove and ColMove.

If the transformation contains a rotation, i.e., if the first image is rotated with respect to the second image, the parameter Rotation may contain an estimate for the rotation angle or an angle interval in radians. A good guess will increase the quality of the gray value matching. If the actual rotation differs too much from the specified estimate the matching will typically fail. The larger the given interval, the slower the operator is since the entire algorithm is run for all relevant angles within the interval.

Once the initial matching is complete, a randomized search algorithm (RANSAC) is used to determine the transformation matrix HomMat2D. It tries to find the matrix that is consistent with a maximum number of correspondences. For a point to be accepted, its distance from the coordinates predicted by the transformation must not exceed the threshold DistanceThreshold.

Once a choice has been made, the matrix is further optimized using all consistent points. For this optimization, the `EstimationMethod` can be chosen to either be the slow but mathematically optimal *'gold_standard'* method or the faster *'normalized_dlt'*. Here, the algorithms of `vector_to_proj_hom_mat2d` are used.

Point pairs that still violate the consistency condition for the final transformation are dropped, the matched points are returned as control values. `Points1` contains the indices of the matched input points from the first image, `Points2` contains the indices of the corresponding points in the second image.

The parameter `RandSeed` can be used to control the randomized nature of the RANSAC algorithm, and hence to obtain reproducible results. If `RandSeed` is set to a positive number, the operator yields the same result on every call with the same parameters because the internally used random number generator is initialized with the seed value. If `RandSeed` = *0*, the random number generator is initialized with the current time. Hence, the results may not be reproducible in this case. The value set for the HALCON system variable *'seed_rand'* (see `set_system`) does not affect the results of `proj_match_points_ransac`.

Parameters

- ▷ **`Image1`** (input_object) . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image 1.
- ▷ **`Image2`** (input_object) . . . singlechannelimage ⇝ *object :* byte / uint2
  Input image 2.
- ▷ **`Rows1`** (input_control) . . . point.x-array ⇝ *real* / integer
  Row coordinates of characteristic points in image 1.
- ▷ **`Cols1`** (input_control) . . . point.y-array ⇝ *real* / integer
  Column coordinates of characteristic points in image 1.
- ▷ **`Rows2`** (input_control) . . . point.x-array ⇝ *real* / integer
  Row coordinates of characteristic points in image 2.
- ▷ **`Cols2`** (input_control) . . . point.y-array ⇝ *real* / integer
  Column coordinates of characteristic points in image 2.
- ▷ **`GrayMatchMethod`** (input_control) . . . string ⇝ *string*
  Gray value comparison metric.
  Default: 'ssd'
  List of values: `GrayMatchMethod` ∈ {'ssd', 'sad', 'ncc'}
- ▷ **`MaskSize`** (input_control) . . . integer ⇝ *integer*
  Size of gray value masks.
  Default: 10
  Value range: MaskSize ≤ 90
- ▷ **`RowMove`** (input_control) . . . integer ⇝ *integer*
  Average row coordinate shift.
  Default: 0
- ▷ **`ColMove`** (input_control) . . . integer ⇝ *integer*
  Average column coordinate shift.
  Default: 0
- ▷ **`RowTolerance`** (input_control) . . . integer ⇝ *integer*
  Half height of matching search window.
  Default: 256
- ▷ **`ColTolerance`** (input_control) . . . integer ⇝ *integer*
  Half width of matching search window.
  Default: 256
- ▷ **`Rotation`** (input_control) . . . real(-array) ⇝ *real*
  Range of rotation angles.
  Default: 0.0
  Suggested values: `Rotation` ∈ {0.0, 0.7854, 1.571, 3.142}
- ▷ **`MatchThreshold`** (input_control) . . . number ⇝ *integer* / real
  Threshold for gray value matching.
  Default: 10
  Suggested values: `MatchThreshold` ∈ {10, 20, 50, 100, 0.9, 0.7}
- ▷ **`EstimationMethod`** (input_control) . . . string ⇝ *string*
  Transformation matrix estimation algorithm.
  Default: 'normalized_dlt'
  List of values: `EstimationMethod` ∈ {'normalized_dlt', 'gold_standard'}

▷ **DistanceThreshold** (input_control) . . . . . real ⇝ *real*
Threshold for transformation consistency check.
Default: 0.2

▷ **RandSeed** (input_control) . . . . . integer ⇝ *integer*
Seed for the random number generator.
Default: 0

▷ **HomMat2D** (output_control) . . . . . hom_mat2d ⇝ *real*
Homogeneous projective transformation matrix.

▷ **Points1** (output_control) . . . . . integer-array ⇝ *integer*
Indices of matched input points in image 1.

▷ **Points2** (output_control) . . . . . integer-array ⇝ *integer*
Indices of matched input points in image 2.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_foerstner, points_harris

Possible Successors

projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_point_2d, projective_trans_pixel

Alternatives

hom_vector_to_proj_hom_mat2d, vector_to_proj_hom_mat2d

See also

proj_match_points_ransac_guided

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2000.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

Matching

proj_match_points_ransac_guided (Image1, Image2 : : Rows1, Cols1, Rows2, Cols2, GrayMatchMethod, MaskSize, HomMat2DGuide, DistanceTolerance, MatchThreshold, EstimationMethod, DistanceThreshold, RandSeed : HomMat2D, Points1, Points2)

Compute a projective transformation matrix between two images by finding correspondences between points based on a known approximation of the projective transformation matrix.

Given a set of coordinates of characteristic points (Cols1,Rows1) and (Cols2,Rows2) in both input images Image1 and Image2, and given a known approximation HomMat2DGuide for the transformation matrix between Image1 and Image2, proj_match_points_ransac_guided automatically determines corresponding points and the homogeneous projective transformation matrix HomMat2D that best transforms the corresponding points from the different images into each other. The characteristic points can, for example, be extracted with points_foerstner or points_harris. The approximation HomMat2DGuide can, for example, be calculated with proj_match_points_ransac on lower resolution versions of Image1 and Image2.

The transformation is determined in two steps: First, gray value correlations of mask windows around the input points in the first and the second image are determined and an initial matching between them is generated using the similarity of the windows in both images.

The size of the mask windows is MaskSize × MaskSize. Three metrics for the correlation can be selected. If GrayMatchMethod has the value *'ssd'*, the sum of the squared gray value differences is used, *'sad'* means the sum of absolute differences, and *'ncc'* is the normalized cross correlation. For details please refer to binocular_disparity. The metric is minimized (*'ssd'*, *'sad'*) or maximized (*'ncc'*) over all possible point pairs. A thus found matching is only accepted if the value of the metric is below the value of MatchThreshold (*'ssd'*, *'sad'*) or above that value (*'ncc'*).

To increase the algorithm's performance, the search area for the matching operations is limited based on the approximate transformation HomMat2DGuide. Only points within a distance of DistanceTolerance around the transformed a point in Image2 of a point in Image1 via HomMat2DGuide are considered for the matching.

Once the initial matching is complete, a randomized search algorithm (RANSAC) is used to determine the transformation matrix HomMat2D. It tries to find the matrix that is consistent with a maximum number of correspondences. For a point to be accepted, its distance from the coordinates predicted by the transformation must not exceed the threshold DistanceThreshold.

Once a choice has been made, the matrix is further optimized using all consistent points. For this optimization, the EstimationMethod can be chosen to either be the slow but mathematically optimal *'gold_standard'* method or the faster *'normalized_dlt'*. Here, the algorithms of vector_to_proj_hom_mat2d are used.

Point pairs that still violate the consistency condition for the final transformation are dropped, the matched points are returned as control values. Points1 contains the indices of the matched input points from the first image, Points2 contains the indices of the corresponding points in the second image.

The parameter RandSeed can be used to control the randomized nature of the RANSAC algorithm, and hence to obtain reproducible results. If RandSeed is set to a positive number, the operator yields the same result on every call with the same parameters because the internally used random number generator is initialized with the seed value. If RandSeed = *0*, the random number generator is initialized with the current time. Hence, the results may not be reproducible in this case. The value set for the HALCON system variable *'seed_rand'* (see set_system) does not affect the results of proj_match_points_ransac_guided.

Parameters

- ▷ **Image1** (input_object) . . . . . singlechannelimage ⇝ *object* : byte / uint2
  Input image 1.
- ▷ **Image2** (input_object) . . . . . singlechannelimage ⇝ *object* : byte / uint2
  Input image 2.
- ▷ **Rows1** (input_control) . . . . . point.x-array ⇝ *real* / integer
  Row coordinates of characteristic points in image 1.
- ▷ **Cols1** (input_control) . . . . . point.y-array ⇝ *real* / integer
  Column coordinates of characteristic points in image 1.
- ▷ **Rows2** (input_control) . . . . . point.x-array ⇝ *real* / integer
  Row coordinates of characteristic points in image 2.
- ▷ **Cols2** (input_control) . . . . . point.y-array ⇝ *real* / integer
  Column coordinates of characteristic points in image 2.
- ▷ **GrayMatchMethod** (input_control) . . . . . string ⇝ *string*
  Gray value comparison metric.
  Default: 'ssd'
  List of values: GrayMatchMethod ∈ {'ssd', 'sad', 'ncc'}
- ▷ **MaskSize** (input_control) . . . . . integer ⇝ *integer*
  Size of gray value masks.
  Default: 10
  Value range: MaskSize ≤ 90
- ▷ **HomMat2DGuide** (input_control) . . . . . hom_mat2d ⇝ *real*
  Approximation of the Homogeneous projective transformation matrix between the two images.
- ▷ **DistanceTolerance** (input_control) . . . . . real ⇝ *real*
  Tolerance for the matching search window.
  Default: 20.0
  Suggested values: DistanceTolerance ∈ {0.2, 0.5, 1.0, 2.0, 3.0, 5.0, 10.0, 20.0, 50.0}
- ▷ **MatchThreshold** (input_control) . . . . . number ⇝ *integer* / real
  Threshold for gray value matching.
  Default: 10
  Suggested values: MatchThreshold ∈ {10, 20, 50, 100, 0.9, 0.7}

▷ **EstimationMethod** (input_control) ..... string ⇝ *string*
Transformation matrix estimation algorithm.
Default: 'normalized_dlt'
List of values: EstimationMethod ∈ {'normalized_dlt', 'gold_standard'}

▷ **DistanceThreshold** (input_control) ..... real ⇝ *real*
Threshold for transformation consistency check.
Default: 0.2

▷ **RandSeed** (input_control) ..... integer ⇝ *integer*
Seed for the random number generator.
Default: 0

▷ **HomMat2D** (output_control) ..... hom_mat2d ⇝ *real*
Homogeneous projective transformation matrix.

▷ **Points1** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 1.

▷ **Points2** (output_control) ..... integer-array ⇝ *integer*
Indices of matched input points in image 2.

Example

```
zoom_image_factor (Image1, Image1Zoomed, 0.5, 0.5, 'constant')
zoom_image_factor (Image2, Image2Zoomed, 0.5, 0.5, 'constant')
points_foerstner (Image1Zoomed, 1, 2, 3, 200, 0.3, 'gauss', 'false', \
                  Rows1, Cols1, _, _, _, _, _, _, _, _)
points_foerstner (Image2Zoomed, 1, 2, 3, 200, 0.3, 'gauss', 'false', \
                  Rows2, Cols2, _, _, _, _, _, _, _, _)
get_image_pointer1 (Image1Zoomed, Pointer, Type, Width, Height)
proj_match_points_ransac (Image1Zoomed, Image2Zoomed, Rows1, Cols1, \
                           Rows2, Cols2, 'ncc', 10, 0, 0, \
                           Height, Width, 0, 0.5, 'gold_standard', \
                           5, 0, HomMat2D, Points1, Points2)
hom_mat2d_scale_local (HomMat2D, 0.5, 0.5, HomMat2DGuide)
hom_mat2d_scale (HomMat2DGuide, 2, 2, 0, 0, HomMat2DGuide)
points_foerstner (Image1, 1, 2, 3, 200, 0.3, 'gauss', 'false', \
                  Rows1, Cols1, _, _, _, _, _, _, _, _)
points_foerstner (Image2, 1, 2, 3, 200, 0.3, 'gauss', 'false', \
                  Rows2, Cols2, _, _, _, _, _, _, _, _)
proj_match_points_ransac_guided (Image1, Image2, Rows1, Cols1, \
                                  Rows2, Cols2, 'ncc', 10, \
                                  HomMat2DGuide, 40, 0.5, \
                                  'gold_standard', 10, 0, HomMat2D, \
                                  Points1, Points2)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_foerstner, points_harris

Possible Successors

projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_point_2d, projective_trans_pixel

Alternatives

hom_vector_to_proj_hom_mat2d, vector_to_proj_hom_mat2d

See also

proj_match_points_ransac

References

Richard Hartley, Andrew Zisserman: “Multiple View Geometry in Computer Vision”; Cambridge University Press, Cambridge; 2000.

Olivier Faugeras, Quang-Tuan Luong: “The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications”; MIT Press, Cambridge, MA; 2001.

Module

Matching

Chapter 27

Transformations

27.1 2D Transformations

To specify a location in an image, we need a convention how to do so. Such a convention is set via a coordinate system. There are different coordinate systems used in HALCON. Here, we explain the ones used in 2D.

Pixels are discrete and to address them, we have a coordinate system using only integer values, the pixel coordinate system. For a higher accuracy that goes beyond the pixel grid, we need floating point coordinates, like e.g., $(3.6, 4.1)$. This leads to subpixel accurate coordinate systems. In HALCON, we have three different implementations of subpixel coordinate systems:

- Pixel Centered Coordinates, the HALCON Standard Subpixel Coordinate System
- Edge Centered Coordinates
- Polar Coordinates

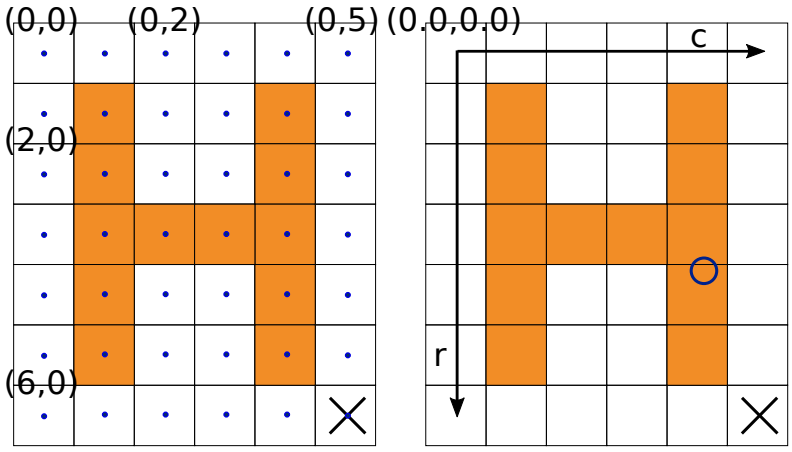
Thereof the first two vary only in the coordinate origin, as visible in the figures below. Calibration makes it possible to map the image coordinates distances to real-world distances. For more information about these Calibrated Coordinates we refer to the "`Solution Guide III-C - 3D Vision`".

HALCON Standard Coordinate System

Pixel Accurate Coordinate System The pixel coordinate system treats the image as a grid of discrete elements, the pixels. In HALCON, we put the origin $(0, 0)$ in the middle of the upper left pixel. Now, we assign the pixel coordinates specifying its row and column like in a matrix.

Note that this implies for an image of size height $\times$ width = $m \times n$ pixels that the row coordinate values run from 0 to $m - 1$ and the column coordinate values from 0 to $n - 1$, as visualized in the figure below.

Subpixel Accurate Coordinate System: Pixel Centered The origin of this coordinate system is in the center of the upper left image pixel, the axes are in row (r) and column (c) direction, respectively. Therewith this convention embeds the pixel coordinate system. The upper left image corner has the coordinates $(-0.5, -0.5)$ and for an image of size height $\times$ width = $m \times n$ pixels the bottom right corner has the coordinates $(m - 0.5, n - 0.5)$ (=$(m - 1 + 0.5, n - 1 + 0.5)$, remember the coordinate values start at 0). It also implies that a pixel (k,l) covers the area of the rectangle $(k - 0.5, l - 0.5)$, $(k + 0.5, l - 0.5)$, $(k - 0.5, l + 0.5)$, $(k+0.5, l+0.5)$. This convention is called the standard coordinate system, or also Image Coordinate System.



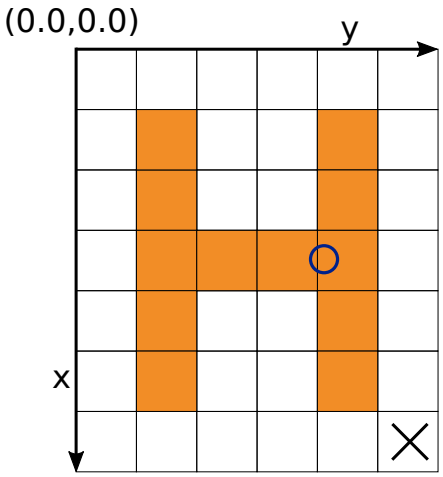
(1) (2)

Visualization of the HALCON standard pixel and subpixel Cartesian coordinate systems. The cross indicates the pixel in the bottom right image corner. Its center has the coordinates $(6, 5)$ (in pixel coordinates (1)), $(6.0, 5.0)$ (in standard subpixel coordinates (2)). The circle center has the coordinates $(3.6, 4.1)$.

HALCON Non-Standard Cartesian Coordinate System

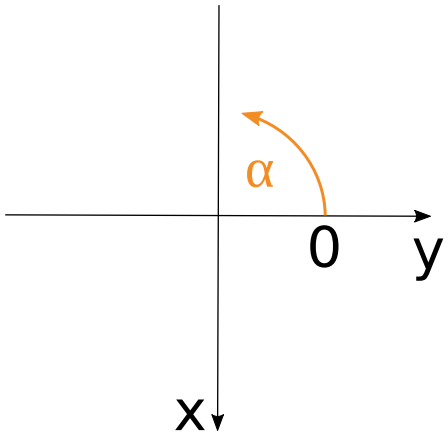
If we rotate an image around its origin by $\pi/2$ (=90 degrees), we want the two images with touching edges but not overlapping with each other. Also, scaling the image is not expected to result in negative image coordinates. For this, the origin $(0.0, 0.0)$ has to be set in an image corner. This motivates the following coordinate system.

Subpixel Accurate Coordinate System: Edge Centered For this coordinate system we set the origin $(0.0, 0.0)$ in the upper left image corner. Thus the center of the upper left pixel has the coordinates $(0.5, 0.5)$ and for an image of size height $\times$ width = $m \times n$ pixels the bottom right corner has the coordinates (m, n). A pixel (k, l) covers the area of the rectangle (k, l), $(k, l + 1)$, $(k + 1, l)$, $(k + 1, l + 1)$.



Visualization of the HALCON non-standard subpixel Cartesian coordinate system. The cross indicates the pixel in the bottom right image corner. Its center has the coordinates $(6.5, 5.5)$. The circle center has the coordinates $(3.6, 4.1)$.

For this coordinate system rotations are defined in the mathematically positive direction and thus counter-clockwise. A rotation of $\pi/2$ (=90 degrees) maps the first axis (= x-axis) onto the second axis (= y-axis). Accordingly, the axes have the assignment row: x coordinate, column: y coordinate.



Visualization of a rotation (α) using the edge centered coordinate system.

Operators Expecting Parameters in any Cartesian Coordinate System

The operator affine_trans_point_2d applies the transformation given by HomMat2D to the point coordinates. This means, affine_trans_point_2d works in both Cartesian Coordinate systems, as long you make sure that the point and the transformation are given in the same coordinate system.

The operators angle_ll and angle_lx may take the input points in pixel centered coordinates, but the returned angle is in the convention of rotations in a mathematically positive direction, thus counterclockwise, and with the horizontal axis as *0*, like in the edge centered coordinate system.

Operators Expecting Parameters in Different Coordinate Systems

In HALCON there is also the case that an operator expects its input in different coordinate systems. On the one hand, the object is expected in its usual coordinates, the standard coordinates. On the other hand, for the transformation matrix HomMat2D, the operator expects edge centered coordinates with their advantages regarding transformations described above. The operator converts the coordinates of the object from HALCON's standard coordinate system (with the origin in the center of the upper left pixel) to the edge centered coordinate system (with the origin in the upper left corner of the upper left pixel). After the transformation with HomMat2D, the result is converted back to the standard coordinate system.

These operators are

- affine_trans_contour_xld
- affine_trans_image
- affine_trans_image_size
- affine_trans_pixel
- affine_trans_polygon_xld
- affine_trans_region
- projective_trans_contour_xld
- projective_trans_image
- projective_trans_image_size
- projective_trans_pixel
- projective_trans_region

A matrix representing a transformation in pixel centered coordinates can be converted to represent the same transformation (e.g., a rotation around the same point) written in edge centered coordinates, e.g., through

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
```

Note, the operators beginning with `projective_` mentioned above use a projective transformation matrix. These transformation matrices can, e.g., be obtained from a 3D camera pose. Doing so, the matrix used is written in a projection of the xy-plane within the 3D coordinate system. Accordingly, the axes have the assignment row: y coordinate, column: x coordinate and therewith the coordinates need to be converted.

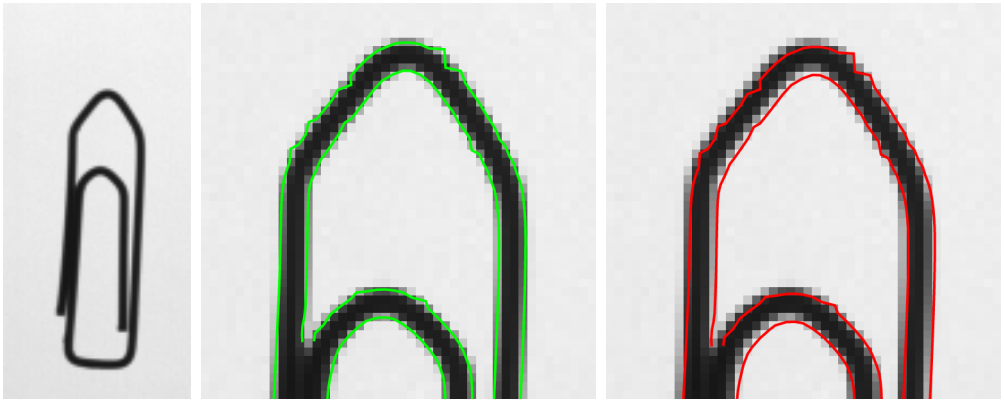
Shape-based Matching and Correlation-based Matching

Results from shape-based matching, like e.g., `find_generic_shape_model`, are given in edge centered coordinates. The returned matches are already transformed. The respective homographic transformation matrices can be retrieved using `get_generic_shape_model_result`.

Results from correlation-based matching, like e.g., `find_ncc_model` and `find_ncc_models`, are computed in edge centered coordinates as well, however the parameters for the transformation are returned separately. With these results one can create a transformation `HomMat2D` directly applicable for, e.g., `affine_trans_contour_xld` and the other operators listed in the paragraph above, entitled `Operators Expecting Parameters in Different Coordinate Systems`.

To display the results found by correlation-based matching, we highly recommend the usage of the procedure `dev_display_ncc_matching_results`.

In the following images we give an example how a displayed match may look when using the transformation matrix in the correct and the erroneous coordinate system, respectively. For the latter one, shown in image (3), the transition matrix is given in pixel centered coordinates as well and therefore the match shown by `affine_trans_contour_xld` is off by 0.5 pixels. Note, this effect is only visible when a rotation is involved.



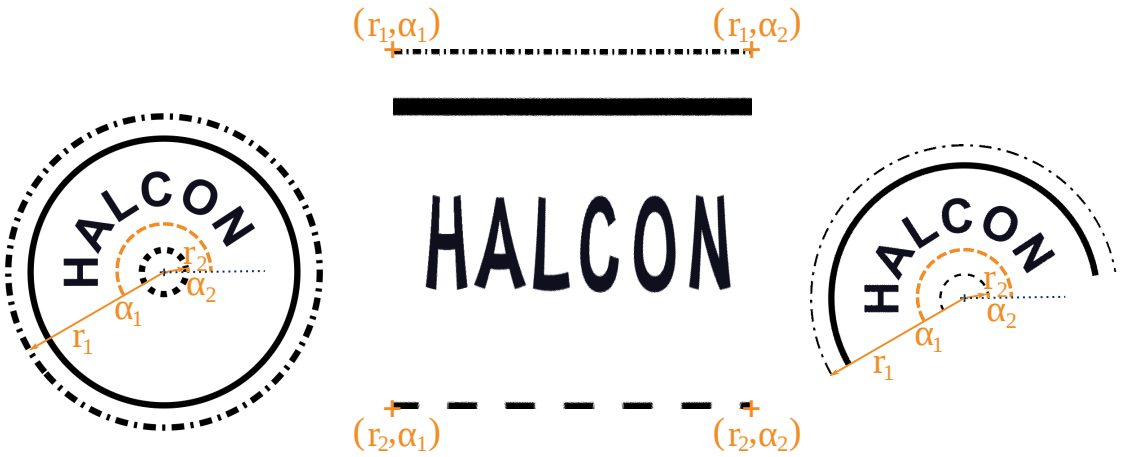
(1) (2) (3)

The original image of the paperclip (1), a part of the match where the inputs are given in the correct coordinates (2), and a match with inputs given in wrong coordinates.

Non-Cartesian Coordinate Systems

Subpixel Accurate Coordinate System: Polar Coordinates In polar coordinates, points are defined through a distance and an angle. The distance is called the radial coordinate and is given in relation to the fix point, the pole. The angular coordinate is given with respect to a defined axis, the polar axis. In HALCON, the pole is specified by (`Row`, `Column`) and the polar axis is the horizontal axis. The angular coordinate is given in radians.

After a transformation with `polar_trans_image_ext`, the upper left pixel in the output image always corresponds to the point in the input image that is specified by `RadiusStart` and `AngleStart`. Analogously, the lower right pixel in the output image corresponds to the point in the input image that is specified by `RadiusEnd` and `AngleEnd`. In the usual mode (`AngleStart` < `AngleEnd` and `RadiusStart` < `RadiusEnd`), the polar transformation is performed in the mathematically positive orientation (counterclockwise). Furthermore, points with smaller radius lie in the upper part of the output image. By suitably exchanging the values of these parameters (e.g., `AngleStart` > `AngleEnd` or `RadiusStart` > `RadiusEnd`), any desired orientation of the output image can be achieved.



As an example, we show an annular arc defined by its pole (Row,Column) (+), the polar axis (...), two angular coordinates AngleStart ($\alpha 1$), AngleEnd ($\alpha 2$) and two radial coordinates RadiusStart ($r1$), RadiusEnd ($r2$). (1) The original image and the parameters defining the annular arc. (2) The annular arc, shown in a figure where the polar coordinates form an equidistant grid obtained by polar_trans_image_ext. (3) The annular arc in the representation of the original image. The Cartesian coordinates have been obtained through polar_trans_image_inv on image (2). The origin is in the center of the pixel in the upper left corner.

Polar coordinates are used by the following operators:

- polar_trans_image_ext
- polar_trans_image_inv
- polar_trans_region
- polar_trans_region_inv
- polar_trans_contour_xld
- polar_trans_contour_xld_inv
- polar_trans_image (legacy)

Images with a reduced domain, regions, and models

In the part before we spoke about coordinates of images. When it comes to the location of the origin of the coordinate system used, images with reduced domains, regions, and models are treated differently than images.

Images with a reduced domain and regions Both images with a reduced domain and regions keep the coordinate system of the image from which they were created. This means, they inherit the origin and the points keep the coordinate values they had in the original image.

Models Models, on the other side, can have a local coordinate system. E.g., models obtained over create_generic_shape_model have their origin in the center of gravity of the ROI they are created from. For further information see the "Solution Guide II-B - Matching".

Calibrated Coordinates

While working with pixel units, we can not extract any information about real-world distances directly. When a camera is calibrated, it is possible to rectify the images. In this case one can assign world coordinates to the image. For further information we refer to the "Solution Guide III-C - 3D Vision".

```
affine_trans_pixel ( : : HomMat2D, Row, Col : RowTrans,
    ColTrans )
```

Apply an arbitrary affine 2D transformation to pixel coordinates.

affine_trans_pixel applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the input pixels (Row,Col) and returns the resulting pixels in (RowTrans,ColTrans); the input and output pixels are subpixel precise coordinates. The affine transformation is described by the homogeneous transformation matrix given in HomMat2D.

In contrast to affine_trans_point_2d, affine_trans_pixel first converts the input coordinates from HALCON's standard coordinate system (with the origin in the center of the upper left pixel) to a coordinate system with the origin in the upper left corner of the upper left pixel. After the transformation with HomMat2D the result is converted back to the standard coordinate system. This way, affine_trans_pixel is compatible with affine_trans_image, affine_trans_image_size, affine_trans_region, affine_trans_contour_xld, and affine_trans_polygon_xld.

Applying affine_trans_pixel corresponds to the following chain of transformations (input and output pixels as homogeneous vectors):

$$\begin{pmatrix} \texttt{RowTrans} \\ \texttt{ColTrans} \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & -0.5 \\ 0 & 1 & -0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D} \cdot \begin{bmatrix} 1 & 0 & +0.5 \\ 0 & 1 & +0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} \texttt{Row} \\ \texttt{Col} \\ 1 \end{pmatrix}$$

Hence, affine_trans_pixel(HomMat2D, Row, Col, RowTrans, ColTrans) corresponds to the following operator sequence:

```
affine_trans_point_2d(HomMat2D, Row+0.5, Col+0.5, RowTmp, ColTmp)
RowTrans := RowTmp-0.5
ColTrans := ColTmp-0.5
```

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

- ▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
  Input transformation matrix.
- ▷ **Row** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
  Input pixel(s) (row coordinate).
  Default: 64
  Suggested values: Row ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Col** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
  Input pixel(s) (column coordinate).
  Default: 64
  Suggested values: Col ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **RowTrans** (output_control) . . . . . point.x(-array) ⇝ *real*
  Output pixel(s) (row coordinate).
- ▷ **ColTrans** (output_control) . . . . . point.y(-array) ⇝ *real*
  Output pixel(s) (column coordinate).

Result

If the matrix HomMat2D represents an affine transformation (i.e., not a projective transformation), affine_trans_pixel returns 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

hom_mat2d_translate, hom_mat2d_translate_local, hom_mat2d_scale, hom_mat2d_scale_local, hom_mat2d_rotate, hom_mat2d_rotate_local,

hom_mat2d_slant, hom_mat2d_slant_local, hom_mat2d_reflect, hom_mat2d_reflect_local

Alternatives

affine_trans_point_2d

Module

Foundation

affine_trans_point_2d (: : HomMat2D, Px, Py : Qx, Qy)

Apply an arbitrary affine 2D transformation to points.

affine_trans_point_2d applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the input points (Px,Py) and returns the resulting points in (Qx,Qy). The affine transformation is described by the homogeneous transformation matrix given in HomMat2D. This corresponds to the following equation (input and output points as homogeneous vectors):

$$\begin{pmatrix} Qx \\ Qy \\ 1 \end{pmatrix} = HomMat2D \cdot \begin{pmatrix} Px \\ Py \\ 1 \end{pmatrix}$$

If the points to transform are specified in standard image coordinates, their *row* coordinates must be passed in Px and their *column* coordinates in Py. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

The transformation matrix can be created using the operators hom_mat2d_identity, hom_mat2d_rotate, hom_mat2d_translate, etc., or can be the result of operators like vector_angle_to_rigid.

For example, if HomMat2D corresponds to a rigid transformation, i.e., if it consists of a rotation and a translation, the points are transformed as follows:

$$\begin{pmatrix} Qx \\ Qy \\ 1 \end{pmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ 0\,0 & 1 \end{bmatrix} \cdot \begin{pmatrix} Px \\ Py \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R} \cdot \begin{pmatrix} Px \\ Py \end{pmatrix} + \mathbf{T} \\ 1 \end{pmatrix}$$

Parameters

- ▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
  Input transformation matrix.
- ▷ **Px** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
  Input point(s) (x or row coordinate).
  Default: 64
  Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Py** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
  Input point(s) (y or column coordinate).
  Default: 64
  Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Qx** (output_control) . . . . . point.x(-array) ⇝ *real*
  Output point(s) (x or row coordinate).
- ▷ **Qy** (output_control) . . . . . point.y(-array) ⇝ *real*
  Output point(s) (y or column coordinate).

Result

If the matrix HomMat2D represents an affine transformation (i.e., not a projective transformation), affine_trans_point_2d returns 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

hom_mat2d_translate, hom_mat2d_translate_local, hom_mat2d_scale, hom_mat2d_scale_local, hom_mat2d_rotate, hom_mat2d_rotate_local, hom_mat2d_slant, hom_mat2d_slant_local, hom_mat2d_reflect, hom_mat2d_reflect_local

Possible Successors

hom_mat2d_translate, hom_mat2d_translate_local, hom_mat2d_scale, hom_mat2d_scale_local, hom_mat2d_rotate, hom_mat2d_rotate_local, hom_mat2d_slant, hom_mat2d_slant_local, hom_mat2d_reflect, hom_mat2d_reflect_local

Module

Foundation

deserialize_hom_mat2d (: : SerializedItemHandle : HomMat2D)

Deserialize a serialized homogeneous 2D transformation matrix.

deserialize_hom_mat2d deserializes a homogeneous 2D transformation matrix, that was serialized by serialize_hom_mat2d (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized transformation matrix is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created transformation matrix with the handle HomMat2D.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **HomMat2D** (output_control) ..... hom_mat2d ⇝ *real*
Transformation matrix.

Result

If the parameters are valid, the operator deserialize_hom_mat2d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_hom_mat2d

Module

Foundation

hom_mat2d_compose (: : HomMat2DLeft, HomMat2DRight : HomMat2DCompose)

Multiply two homogeneous 2D transformation matrices.

hom_mat2d_compose composes a new 2D transformation matrix by multiplying the two input matrices:

$$\text{HomMat2DCompose} = \text{HomMat2DLeft} \cdot \text{HomMat2DRight}$$

For example, if the two input matrices correspond to rigid transformations, i.e., to transformations consisting of a rotation and a translation, the resulting matrix is calculated as follows:

$$\texttt{HomMat2DCompose} = \begin{bmatrix} \mathbf{R}_l & \mathbf{T}_l \\ 0\,0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}_r & \mathbf{T}_r \\ 0\,0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_l \cdot \mathbf{R}_r & \mathbf{R}_l \cdot \mathbf{T}_r + \mathbf{T}_l \\ 0\quad 0 & 1 \end{bmatrix}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

▷ **HomMat2DLeft** (input_control) ..... hom_mat2d ⇝ *real*
Left input transformation matrix.

▷ **HomMat2DRight** (input_control) ..... hom_mat2d ⇝ *real*
Right input transformation matrix.

▷ **HomMat2DCompose** (output_control) ..... hom_mat2d ⇝ *real*
Output transformation matrix.

Result

If the parameters are valid, the operator `hom_mat2d_compose` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`hom_mat2d_compose`, `hom_mat2d_translate`, `hom_mat2d_translate_local`, `hom_mat2d_scale`, `hom_mat2d_scale_local`, `hom_mat2d_rotate`, `hom_mat2d_rotate_local`, `hom_mat2d_slant`, `hom_mat2d_slant_local`, `hom_mat2d_reflect`, `hom_mat2d_reflect_local`

Possible Successors

`hom_mat2d_compose`, `hom_mat2d_translate`, `hom_mat2d_translate_local`, `hom_mat2d_scale`, `hom_mat2d_scale_local`, `hom_mat2d_rotate`, `hom_mat2d_rotate_local`, `hom_mat2d_slant`, `hom_mat2d_slant_local`, `hom_mat2d_reflect`, `hom_mat2d_reflect_local`

Module

Foundation

hom_mat2d_determinant (: : HomMat2D : Determinant)

Compute the determinant of a homogeneous 2D transformation matrix.

`hom_mat2d_determinant` computes the determinant of the homogeneous 2D transformation matrix given by `HomMat2D` and returns it in `Determinant`.

Parameters

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Determinant** (output_control) . . . . . real ⇝ *real*
Determinant of the input matrix.

Result

`hom_mat2d_determinant` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`hom_mat2d_translate`, `hom_mat2d_translate_local`, `hom_mat2d_scale`, `hom_mat2d_scale_local`, `hom_mat2d_rotate`, `hom_mat2d_rotate_local`, `hom_mat2d_slant`, `hom_mat2d_slant_local`, `hom_mat2d_reflect`, `hom_mat2d_reflect_local`

Module

Foundation

hom_mat2d_identity (: : : HomMat2DIdentity)

Generate the homogeneous transformation matrix of the identical 2D transformation.

`hom_mat2d_identity` generates the homogeneous transformation matrix `HomMat2DIdentity` describing the identical 2D transformation:

$$\texttt{HomMat2DIdentity} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. Thus, `HomMat2DIdentity` is stored as the tuple [1,0,0,0,1,0].

Parameters

▷ **HomMat2DIdentity** (output_control) . . . . . hom_mat2d ⇝ *real*
Transformation matrix.

Result

`hom_mat2d_identity` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`hom_mat2d_translate`, `hom_mat2d_translate_local`, `hom_mat2d_scale`, `hom_mat2d_scale_local`, `hom_mat2d_rotate`, `hom_mat2d_rotate_local`, `hom_mat2d_slant`, `hom_mat2d_slant_local`, `hom_mat2d_reflect`, `hom_mat2d_reflect_local`

Module

Foundation

hom_mat2d_invert (: : HomMat2D : HomMat2DInvert)

Invert a homogeneous 2D transformation matrix.

hom_mat2d_invert inverts the homogeneous 2D transformation matrix given by HomMat2D. The resulting matrix is returned in HomMat2DInvert.

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **HomMat2DInvert** (output_control) . . . . . hom_mat2d ⇝ *real*
Output transformation matrix.

Result

hom_mat2d_invert returns 2 (H_MSG_TRUE) if the parameters are valid and the input matrix is invertible. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_translate, hom_mat2d_translate_local, hom_mat2d_scale, hom_mat2d_scale_local, hom_mat2d_rotate, hom_mat2d_rotate_local, hom_mat2d_slant, hom_mat2d_slant_local, hom_mat2d_reflect, hom_mat2d_reflect_local

Possible Successors

hom_mat2d_translate, hom_mat2d_translate_local, hom_mat2d_scale, hom_mat2d_scale_local, hom_mat2d_rotate, hom_mat2d_rotate_local, hom_mat2d_slant, hom_mat2d_slant_local, hom_mat2d_reflect, hom_mat2d_reflect_local

Module

Foundation

hom_mat2d_reflect (: : HomMat2D, Px, Py, Qx, Qy : HomMat2DReflect)

Add a reflection to a homogeneous 2D transformation matrix.

hom_mat2d_reflect adds a reflection about the axis given by the two points (Px,Py) and (Qx,Qy) to the homogeneous 2D transformation matrix HomMat2D and returns the resulting matrix in HomMat2DReflect. The reflection is described by a 2×2 reflection matrix $\mathbf{M}$. It is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\text{HomMat2DReflect} = \begin{bmatrix} \mathbf{M} & & 0 \\ & & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D} \qquad \mathbf{M} = \mathbf{I} - \frac{2}{v^T v} v v^T$$

where $v = (\texttt{Py} - \texttt{Qy}, \texttt{Qx} - \texttt{Px})^T$ is the normal vector to the axis.

The axis $(\texttt{Px}, \texttt{Py}) - (\texttt{Qx}, \texttt{Qy})$ is fixed in the transformation, i.e., the points on the axis remain unchanged when transformed using HomMat2DReflect. To obtain this behavior, first a translation is added to the input transformation matrix that moves the axis onto the origin of the global coordinate system. Then, the reflection is added, and finally a translation that moves the axis back to its original position. This corresponds to the following chain of transformations:

$$\texttt{HomMat2DReflect} = \begin{bmatrix} 1 & 0 & +\texttt{Px} \\ 0 & 1 & +\texttt{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} & \mathbf{M} & 0 \\ & & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -\texttt{Px} \\ 0 & 1 & -\texttt{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D}$$

To perform the transformation in the local coordinate system, i.e., the one described by HomMat2D, use hom_mat2d_reflect_local.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

- ▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
  Input transformation matrix.
- ▷ **Px** (input_control) ..... point.x ⇝ *real* / integer
  First point of the axis (x coordinate).
  Default: 0
  Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Py** (input_control) ..... point.y ⇝ *real* / integer
  First point of the axis (y coordinate).
  Default: 0
  Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Qx** (input_control) ..... point.x ⇝ *real* / integer
  Second point of the axis (x coordinate).
  Default: 16
  Suggested values: Qx ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Qy** (input_control) ..... point.y ⇝ *real* / integer
  Second point of the axis (y coordinate).
  Default: 32
  Suggested values: Qy ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **HomMat2DReflect** (output_control) ..... hom_mat2d ⇝ *real*
  Output transformation matrix.

Result

hom_mat2d_reflect returns 2 (H_MSG_TRUE) if both points on the axis are not identical. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_slant, hom_mat2d_reflect

Possible Successors

hom_mat2d_translate, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_slant, hom_mat2d_reflect

See also

hom_mat2d_reflect_local

Module

Foundation

hom_mat2d_reflect_local (: : HomMat2D, Px, Py : HomMat2DReflect)

Add a reflection to a homogeneous 2D transformation matrix.

hom_mat2d_reflect_local adds a reflection about the axis given by the two points (0,0) and (Px,Py) to the homogeneous 2D transformation matrix HomMat2D and returns the resulting matrix in HomMat2DReflect. The reflection is described by a 2×2 reflection matrix $\mathbf{M}$. In contrast to hom_mat2d_reflect, it is performed relative to the local coordinate system, i.e., the coordinate system described by HomMat2D; this corresponds to the following chain of transformation matrices:

$$\texttt{HomMat2DReflect} = \begin{bmatrix} \mathbf{M} & \begin{matrix}0\\0\end{matrix} \\ \begin{matrix}0 & 0\end{matrix} & 1 \end{bmatrix} \cdot \texttt{HomMat2D} \qquad \mathbf{M} = \mathbf{I} - \frac{2}{v^T v} v v^T$$

where $v = (-\texttt{Py}, \texttt{Px})^T$ is the normal vector to the axis.

The axis $(0,0) - (\texttt{Px}, \texttt{Py})$ is fixed in the transformation, i.e., the points on the axis remain unchanged when transformed using HomMat2DReflect.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Px** (input_control) ..... point.x ⇝ *real* / integer
Point that defines the axis (x coordinate).
Default: 16
Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Py** (input_control) . . . . . point.y ⇝ *real* / integer
Point that defines the axis (y coordinate).
Default: 32
Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **HomMat2DReflect** (output_control) . . . . . hom_mat2d ⇝ *real*
Output transformation matrix.

Result

hom_mat2d_reflect_local returns 2 (H_MSG_TRUE) if the point (Px, Py) is not $(0, 0)$. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect_local

Possible Successors

hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect_local

See also

hom_mat2d_reflect

Module

Foundation

hom_mat2d_rotate (: : HomMat2D, Phi, Px, Py : HomMat2DRotate)

Add a rotation to a homogeneous 2D transformation matrix.

hom_mat2d_rotate adds a rotation by the angle Phi to the homogeneous 2D transformation matrix HomMat2D and returns the resulting matrix in HomMat2DRotate. The rotation is described by a 2×2 rotation matrix $\mathbf{R}$. It is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\texttt{HomMat2DRotate} = \begin{bmatrix} \mathbf{R} & & 0 \\ & & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D} \qquad \mathbf{R} = \begin{bmatrix} \cos(\texttt{Phi}) & -\sin(\texttt{Phi}) \\ \sin(\texttt{Phi}) & \cos(\texttt{Phi}) \end{bmatrix}$$

The point (Px,Py) is the fixed point of the transformation, i.e., this point remains unchanged when transformed using HomMat2DRotate. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the rotation is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations:

$$\texttt{HomMat2DRotate} = \begin{bmatrix} 1 & 0 & +\texttt{Px} \\ 0 & 1 & +\texttt{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & & 0 \\ & & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -\texttt{Px} \\ 0 & 1 & -\texttt{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D}$$

To perform the transformation in the local coordinate system, i.e., the one described by HomMat2D, use hom_mat2d_rotate_local.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in

the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Phi** (input_control) . . . . . angle.rad ⇝ *real* / integer
Rotation angle.
Default: 0.78
Suggested values: Phi ∈ {0.1, 0.2, 0.3, 0.4, 0.78, 1.57, 3.14}
Value range: 0 ≤ Phi ≤ 6.28318530718

▷ **Px** (input_control) . . . . . point.x ⇝ *real* / integer
Fixed point of the transformation (x coordinate).
Default: 0
Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Py** (input_control) . . . . . point.y ⇝ *real* / integer
Fixed point of the transformation (y coordinate).
Default: 0
Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **HomMat2DRotate** (output_control) . . . . . hom_mat2d ⇝ *real*
Output transformation matrix.

Result

If the parameters are valid, the operator hom_mat2d_rotate returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_slant, hom_mat2d_reflect

Possible Successors

hom_mat2d_translate, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_slant, hom_mat2d_reflect

See also

hom_mat2d_rotate_local

Module

Foundation

hom_mat2d_rotate_local (: : HomMat2D, Phi : HomMat2DRotate)

Add a rotation to a homogeneous 2D transformation matrix.

hom_mat2d_rotate_local adds a rotation by the angle Phi to the homogeneous 2D transformation matrix HomMat2D and returns the resulting matrix in HomMat2DRotate. The rotation is described by a 2×2 rotation matrix $\mathbf{R}$. In contrast to hom_mat2d_rotate, it is performed relative to the local coordinate system, i.e., the coordinate system described by HomMat2D; this corresponds to the following chain of transformation matrices:

$$\texttt{HomMat2DRotate} = \texttt{HomMat2D} \cdot \begin{bmatrix} \mathbf{R} & & 0 \\ & & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R} = \begin{bmatrix} \cos(\texttt{Phi}) & -\sin(\texttt{Phi}) \\ \sin(\texttt{Phi}) & \cos(\texttt{Phi}) \end{bmatrix}$$

The fixed point of the transformation is the origin of the local coordinate system, i.e., this point remains unchanged when transformed using HomMat2DRotate.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Phi** (input_control) . . . . . angle.rad ⇝ *real* / integer
Rotation angle.
Default: 0.78
Suggested values: Phi ∈ {0.1, 0.2, 0.3, 0.4, 0.78, 1.57, 3.14}
Value range: 0 ≤ Phi ≤ 6.28318530718

▷ **HomMat2DRotate** (output_control) . . . . . hom_mat2d ⇝ *real*
Output transformation matrix.

Result

If the parameters are valid, the operator hom_mat2d_rotate_local returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect_local

Possible Successors

hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect_local

See also

hom_mat2d_rotate

Module

Foundation

hom_mat2d_scale (: : HomMat2D, Sx, Sy, Px, Py : HomMat2DScale)

Add a scaling to a homogeneous 2D transformation matrix.

hom_mat2d_scale adds a scaling by the scale factors Sx and Sy to the homogeneous 2D transformation matrix HomMat2D and returns the resulting matrix in HomMat2DScale. The scaling is described by a 2×2 scaling matrix $\mathbf{S}$. It is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\mathtt{HomMat2DScale} = \begin{bmatrix} & \mathbf{S} & & 0 \\ & & & 0 \\ 0 & 0 & & 1 \end{bmatrix} \cdot \mathtt{HomMat2D} \qquad \mathbf{S} = \begin{bmatrix} \mathtt{Sx} & 0 \\ 0 & \mathtt{Sy} \end{bmatrix}$$

The point (Px,Py) is the fixed point of the transformation, i.e., this point remains unchanged when transformed using HomMat2DScale. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the scaling is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations:

$$\mathtt{HomMat2DScale} = \begin{bmatrix} 1 & 0 & +\mathtt{Px} \\ 0 & 1 & +\mathtt{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} & \mathbf{S} & & 0 \\ & & & 0 \\ 0 & 0 & & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -\mathtt{Px} \\ 0 & 1 & -\mathtt{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \mathtt{HomMat2D}$$

To perform the transformation in the local coordinate system, i.e., the one described by HomMat2D, use hom_mat2d_scale_local.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Sx** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the x-axis.
Default: 2
Suggested values: Sx ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 16}
Restriction: Sx != 0

▷ **Sy** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the y-axis.
Default: 2
Suggested values: Sy ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 16}
Restriction: Sy != 0

▷ **Px** (input_control) . . . . . point.x ⇝ *real* / integer
Fixed point of the transformation (x coordinate).
Default: 0
Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Py** (input_control) . . . . . point.y ⇝ *real* / integer
Fixed point of the transformation (y coordinate).
Default: 0
Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **HomMat2DScale** (output_control) . . . . . hom_mat2d ⇝ *real*
Output transformation matrix.

Result

hom_mat2d_scale returns 2 (H_MSG_TRUE) if both scale factors are not 0. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_slant, hom_mat2d_reflect

Possible Successors

hom_mat2d_translate, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_slant, hom_mat2d_reflect

See also

hom_mat2d_scale_local

Module

Foundation

hom_mat2d_scale_local (: : HomMat2D, Sx, Sy : HomMat2DScale)

Add a scaling to a homogeneous 2D transformation matrix.

hom_mat2d_scale_local adds a scaling by the scale factors Sx and Sy to the homogeneous 2D transformation matrix HomMat2D and returns the resulting matrix in HomMat2DScale. The scaling is described by a 2×2 scaling matrix $\mathbf{S}$. In contrast to hom_mat2d_scale, it is performed relative to the local coordinate system, i.e., the coordinate system described by HomMat2D; this corresponds to the following chain of transformation matrices:

$$\text{HomMat2DScale} = \text{HomMat2D} \cdot \begin{bmatrix} & \mathbf{S} & & 0 \\ & & & 0 \\ 0 & & 0 & 1 \end{bmatrix} \qquad \mathbf{S} = \begin{bmatrix} \text{Sx} & 0 \\ 0 & \text{Sy} \end{bmatrix}$$

The fixed point of the transformation is the origin of the local coordinate system, i.e., this point remains unchanged when transformed using HomMat2DScale.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Sx** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the x-axis.
Default: 2
Suggested values: Sx ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 16}
Restriction: Sx != 0

▷ **Sy** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the y-axis.
Default: 2
Suggested values: Sy ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 16}
Restriction: Sy != 0

▷ **HomMat2DScale** (output_control) . . . . . hom_mat2d ⇝ *real*
Output transformation matrix.

Result

hom_mat2d_scale_local returns 2 (H_MSG_TRUE) if both scale factors are not 0. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect_local

Possible Successors

hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect_local

See also

hom_mat2d_scale

Module

Foundation

hom_mat2d_slant (: : HomMat2D, Theta, Axis, Px, Py : HomMat2DSlant)

Add a slant to a homogeneous 2D transformation matrix.

hom_mat2d_slant adds a slant by the angle Theta to the homogeneous 2D transformation matrix HomMat2D and returns the resulting matrix in HomMat2DSlant. A slant is an affine transformation in which one coordinate axis remains fixed, while the other coordinate axis is rotated counterclockwise by an angle Theta. The parameter Axis determines which coordinate axis is slanted. For Axis = *'x'*, the x-axis is slanted and the y-axis remains fixed, while for Axis = *'y'* the y-axis is slanted and the x-axis remains fixed. The slanting is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chains of transformation matrices:

$$\texttt{Axis} = 'x': \quad \texttt{HomMat2DSlant} = \begin{bmatrix} \cos(\texttt{Theta}) & 0 & 0 \\ \sin(\texttt{Theta}) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D}$$

$$\texttt{Axis} = 'y': \quad \texttt{HomMat2DSlant} = \begin{bmatrix} 1 & -\sin(\texttt{Theta}) & 0 \\ 0 & \cos(\texttt{Theta}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D}$$

The point (Px,Py) is the fixed point of the transformation, i.e., this point remains unchanged when transformed using HomMat2DSlant. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the slant is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations for Axis = *'x'*:

$$\texttt{HomMat2DSlant} = \begin{bmatrix} 1 & 0 & +\texttt{Px} \\ 0 & 1 & +\texttt{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\texttt{Theta}) & 0 & 0 \\ \sin(\texttt{Theta}) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -\texttt{Px} \\ 0 & 1 & -\texttt{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D}$$

To perform the transformation in the local coordinate system, i.e., the one described by HomMat2D, use hom_mat2d_slant_local.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Theta** (input_control) ..... angle.rad ⇝ *real* / integer
Slant angle.
Default: 0.78
Suggested values: Theta ∈ {0.1, 0.2, 0.3, 0.4, 0.78, 1.57, 3.14}
Value range: 0 ≤ Theta ≤ 6.28318530718

▷ **Axis** (input_control) ..... string ⇝ *string*
Coordinate axis that is slanted.
Default: 'x'
List of values: Axis ∈ {'x', 'y'}

▷ **Px** (input_control) ..... point.x ⇝ *real* / integer
Fixed point of the transformation (x coordinate).
Default: 0
Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Py** (input_control) ..... point.y ⇝ *real* / integer
Fixed point of the transformation (y coordinate).
Default: 0
Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **HomMat2DSlant** (output_control) . . . . . hom_mat2d ⇝ *real*
Output transformation matrix.

Result

If the parameters are valid, the operator `hom_mat2d_slant` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`hom_mat2d_identity`, `hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`, `hom_mat2d_reflect`

Possible Successors

`hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`, `hom_mat2d_reflect`

See also

`hom_mat2d_slant_local`

Module

Foundation

hom_mat2d_slant_local (: : HomMat2D, Theta, Axis : HomMat2DSlant)

Add a slant to a homogeneous 2D transformation matrix.

`hom_mat2d_slant_local` adds a slant by the angle `Theta` to the homogeneous 2D transformation matrix `HomMat2D` and returns the resulting matrix in `HomMat2DSlant`. A slant is an affine transformation in which one coordinate axis remains fixed, while the other coordinate axis is rotated counterclockwise by an angle `Theta`. The parameter `Axis` determines which coordinate axis is slanted. For `Axis` = *'x'*, the x-axis is slanted and the y-axis remains fixed, while for `Axis` = *'y'* the y-axis is slanted and the x-axis remains fixed. In contrast to `hom_mat2d_slant`, the slanting is performed relative to the local coordinate system, i.e., the coordinate system described by `HomMat2D`; this corresponds to the following chains of transformation matrices:

$$\texttt{Axis} = 'x': \quad \texttt{HomMat2DSlant} = \texttt{HomMat2D} \cdot \begin{bmatrix} \cos(\texttt{Theta}) & 0 & 0 \\ \sin(\texttt{Theta}) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\texttt{Axis} = 'y': \quad \texttt{HomMat2DSlant} = \texttt{HomMat2D} \cdot \begin{bmatrix} 1 & -\sin(\texttt{Theta}) & 0 \\ 0 & \cos(\texttt{Theta}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The fixed point of the transformation is the origin of the local coordinate system, i.e., this point remains unchanged when transformed using `HomMat2DSlant`.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (`Row`,`Column`). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **Theta** (input_control) . . . . . angle.rad ⇝ *real* / integer
Slant angle.
Default: 0.78
Suggested values: Theta ∈ {0.1, 0.2, 0.3, 0.4, 0.78, 1.57, 3.14}
Value range: $0 \leq$ Theta $\leq$ 6.28318530718

▷ **Axis** (input_control) . . . . . string ⇝ *string*
Coordinate axis that is slanted.
Default: 'x'
List of values: Axis ∈ {'x', 'y'}

▷ **HomMat2DSlant** (output_control) . . . . . hom_mat2d ⇝ *real*
Output transformation matrix.

Result

If the parameters are valid, the operator hom_mat2d_slant_local returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect

Possible Successors

hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect_local

See also

hom_mat2d_slant

Module

Foundation

hom_mat2d_to_affine_par (: : HomMat2D : Sx, Sy, Phi, Theta, Tx, Ty)

Compute the affine transformation parameters from a homogeneous 2D transformation matrix.

hom_mat2d_to_affine_par computes the affine transformation parameters corresponding to the homogeneous 2D transformation matrix HomMat2D. The parameters Sx and Sy determine how the transformation scales the original x- and y-axes, respectively. The two scaling factors are always positive. The angle Theta describes whether the transformed coordinate axes are orthogonal (Theta = 0) or slanted. If $|\text{Theta}| > \pi/2$, the transformation contains a reflection. The angle Phi determines the rotation of the transformed x-axis with respect to the original x-axis. The parameters Tx and Ty determine the translation of the two coordinate systems. The matrix HomMat2D can be constructed from the six transformation parameters by the following operator sequence:

```
hom_mat2d_identity(HomMat2DIdentity)
hom_mat2d_scale(HomMat2DIdentity, Sx, Sy, 0, 0, HomMat2DScale)
hom_mat2d_slant(HomMat2DScale, Theta, 'y', 0, 0, HomMat2DSlant)
hom_mat2d_rotate(HomMat2DSlant, Phi, 0, 0, HomMat2DRotate)
hom_mat2d_translate(HomMat2DRotate, Tx, Ty, HomMat2D)
```

This is equivalent to the following chain of transformation matrices:

$$\mathtt{HomMat2D} = \begin{bmatrix} 1 & 0 & +\mathtt{Tx} \\ 0 & 1 & +\mathtt{Ty} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\mathtt{Phi}) & -\sin(\mathtt{Phi}) & 0 \\ \sin(\mathtt{Phi}) & \cos(\mathtt{Phi}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -\sin(\mathtt{Theta}) & 0 \\ 0 & \cos(\mathtt{Theta}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathtt{Sx} & 0 & 0 \\ 0 & \mathtt{Sy} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Parameters

- ▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
  Input transformation matrix.
- ▷ **Sx** (output_control) . . . . . real ⇝ *real*
  Scaling factor along the x direction.
- ▷ **Sy** (output_control) . . . . . real ⇝ *real*
  Scaling factor along the y direction.
- ▷ **Phi** (output_control) . . . . . angle.rad ⇝ *real*
  Rotation angle.
- ▷ **Theta** (output_control) . . . . . angle.rad ⇝ *real*
  Slant angle.
- ▷ **Tx** (output_control) . . . . . point.x ⇝ *real*
  Translation along the x direction.
- ▷ **Ty** (output_control) . . . . . point.y ⇝ *real*
  Translation along the y direction.

Result

If the matrix HomMat2D is non-degenerate and represents an affine transformation (i.e., not a projective transformation), hom_mat2d_to_affine_par returns 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

vector_to_hom_mat2d, vector_to_rigid, vector_to_similarity, vector_to_aniso, point_line_to_hom_mat2d

Possible Successors

hom_mat2d_translate, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_slant

Module

Foundation

hom_mat2d_translate (: : HomMat2D, Tx, Ty : HomMat2DTranslate)

Add a translation to a homogeneous 2D transformation matrix.

hom_mat2d_translate adds a translation by the vector $\mathbf{T}$ = (Tx,Ty) to the homogeneous 2D transformation matrix HomMat2D and returns the resulting matrix in HomMat2DTranslate. The translation is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\texttt{HomMat2DTranslate} = \begin{bmatrix} 1 & 0 & \\ 0 & 1 & \mathbf{T} \\ 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat2D} \qquad \mathbf{T} = \begin{pmatrix} \texttt{Tx} \\ \texttt{Ty} \end{pmatrix}$$

To perform the transformation in the local coordinate system, i.e., the one described by HomMat2D, use hom_mat2d_translate_local.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

- ▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
  Input transformation matrix.
- ▷ **Tx** (input_control) . . . . . point.x ⇝ *real* / integer
  Translation along the x-axis.
  Default: 64
  Suggested values: Tx ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Ty** (input_control) . . . . . point.y ⇝ *real* / integer
  Translation along the y-axis.
  Default: 64
  Suggested values: Ty ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **HomMat2DTranslate** (output_control) . . . . . hom_mat2d ⇝ *real*
  Output transformation matrix.

Result

If the parameters are valid, the operator hom_mat2d_translate returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_slant, hom_mat2d_reflect

Possible Successors

hom_mat2d_translate, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_slant, hom_mat2d_reflect

See also

hom_mat2d_translate_local

Module

Foundation

hom_mat2d_translate_local (: : HomMat2D, Tx, Ty : HomMat2DTranslate)

Add a translation to a homogeneous 2D transformation matrix.

hom_mat2d_translate_local adds a translation by the vector $\mathbf{T}$ = (Tx,Ty) to the homogeneous 2D transformation matrix HomMat2D and returns the resulting matrix in HomMat2DTranslate. In contrast to hom_mat2d_translate, the translation is performed relative to the local coordinate system, i.e., the coordinate system described by HomMat2D; this corresponds to the following chain of transformation matrices:

$$\text{HomMat2DTranslate} = \text{HomMat2D} \cdot \begin{bmatrix} 1 & 0 & \mathbf{T} \\ 0 & 1 & \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{T} = \begin{pmatrix} \text{Tx} \\ \text{Ty} \end{pmatrix}$$

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf]. However, it is also possible to process full 3×3 matrices, which represent a projective 2D transformation.

Parameters

- ▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
  Input transformation matrix.
- ▷ **Tx** (input_control) ..... point.x ⇝ *real* / integer
  Translation along the x-axis.
  Default: 64
  Suggested values: Tx ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Ty** (input_control) ..... point.y ⇝ *real* / integer
  Translation along the y-axis.
  Default: 64
  Suggested values: Ty ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **HomMat2DTranslate** (output_control) ..... hom_mat2d ⇝ *real*
  Output transformation matrix.

Result

If the parameters are valid, the operator hom_mat2d_translate_local returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect_local

Possible Successors

hom_mat2d_translate_local, hom_mat2d_scale_local, hom_mat2d_rotate_local, hom_mat2d_slant_local, hom_mat2d_reflect_local

See also

hom_mat2d_translate

Module

Foundation

hom_mat2d_transpose (: : HomMat2D : HomMat2DTranspose)

Transpose a homogeneous 2D transformation matrix.

hom_mat2d_transpose transposes the homogeneous 2D transformation matrix given by HomMat2D. The result matrix HomMat2DTranspose is always a 3×3 matrix, even if the input matrix is represented by a 2×3 matrix.

Parameters

▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
Input transformation matrix.

▷ **HomMat2DTranspose** (output_control) ..... hom_mat2d ⇝ *real*
Output transformation matrix.

Result

hom_mat2d_transpose always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat2d_translate, hom_mat2d_translate_local, hom_mat2d_scale, hom_mat2d_scale_local, hom_mat2d_rotate, hom_mat2d_rotate_local, hom_mat2d_slant, hom_mat2d_slant_local, hom_mat2d_reflect, hom_mat2d_reflect_local

Possible Successors

hom_mat2d_compose, hom_mat2d_invert

Module

Foundation

hom_mat3d_project (: : HomMat3D, PrincipalPointRow, PrincipalPointCol, Focus : HomMat2D)

Project an affine 3D transformation matrix to a 2D projective transformation matrix.

hom_mat3d_project calculates a homogeneous projection matrix from a homogeneous 3×4 transformation matrix describing an affine transformation in 3D. The result can be used to project a plane, in particular a plane containing an image. The projection matrix defines a projective transformation between two (two-dimensional) planes.

This can be used to create perspective distortions, which occur in a projection of a plane rotated around an axis other than the z axis. Usually, however, projective transformations are determined from point correspondences (see vector_to_proj_hom_mat2d, hom_vector_to_proj_hom_mat2d, and proj_match_points_ransac).

Matrices for rotation, scale, and translation can be constructed using the operators hom_mat3d_identity, hom_mat3d_scale, hom_mat3d_rotate, hom_mat3d_translate and pose_to_hom_mat3d.

Note that for 3D transformations the x-axis represents the column axis while the y-axis represents the row axis (see also Calibration) while in `projective_trans_image`, the first row of `HomMat2D` contains the transformation of the row axis and the second row contains the transformation of the column axis of the image.

The point (`PrincipalPointRow`, `PrincipalPointCol`) is the principal point of the projection and the point (`PrincipalPointRow`, `PrincipalPointCol`, *0*) can thus be interpreted as the position of the camera in a virtual three-dimensional space. The direction of view is along the positive z-axis.

In this virtual space the plane containing the input image as well as the image plane are located at $z =$ `Focus`, which is `Focus` pixels away form the camera. As a result, using the identity matrix as the input matrix `HomMat3D` leads to a matrix `HomMat2D` which also represents the identity in 2D.

Consequently, the parameter `Focus` is the "focal distance" of the virtual camera used and its unit is pixels. Its value influences the degree of perspective distortions. The same input matrix at a bigger focal distance results in weaker distortions than at a low focal distance.

Let H be the affine 3D matrix with elements h_{ij}, $(r, c) =$ (`PrincipalPointRow`, `PrincipalPointCol`) and $f =$ `Focus`.

Then the projective transformation matrix is calculated as follows: First, a 3×4 projection matrix is calculated as:

$$Q = \begin{pmatrix} f & 0 & c \\ 0 & f & r \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -c \\ 0 & 1 & 0 & -r \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ h_{31} & h_{32} & h_{33} & h_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Since the image of a plane containing points $(x, y, f, 1)^T$ is to be calculated the last two columns of Q can be joined:

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} = \begin{pmatrix} q_{11} & q_{12} & f \cdot q_{13} + q_{14} \\ q_{21} & q_{22} & f \cdot r_{23} + q_{24} \\ q_{31} & q_{32} & f \cdot r_{33} + q_{34} \end{pmatrix} = Q \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & f \\ 0 & 0 & 1 \end{pmatrix}$$

Finally, the columns and rows of R are swapped in a way that the first row of P contains the transformation of the row coordinates and the second row contains the transformation of the column coordinates so that P can be used directly in `projective_trans_image`:

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot R \cdot \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The overall transformation can be written as:

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} f & 0 & c \\ 0 & f & r \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -c \\ 0 & 1 & 0 & -r \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot H \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & f \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Parameters

▷ **`HomMat3D`** (input_control) ..... hom_mat3d ⇝ *real*
3 × 4 3D transformation matrix.

▷ **`PrincipalPointRow`** (input_control) ..... point.y ⇝ *real* / integer
Row coordinate of the principal point.
Default: 256
Suggested values: `PrincipalPointRow` ∈ {16, 32, 64, 128, 240, 256, 512}

▷ **`PrincipalPointCol`** (input_control) ..... point.x ⇝ *real* / integer
Column coordinate of the principal point.
Default: 256
Suggested values: `PrincipalPointCol` ∈ {16, 32, 64, 128, 256, 320, 512}

▷ **Focus** (input_control) ..... number ⇝ *real* / integer
Focal length in pixels.
Default: 256
Suggested values: Focus ∈ {1, 2, 5, 256, 32768}

▷ **HomMat2D** (output_control) ..... hom_mat2d ⇝ *real*
Homogeneous projective transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_identity, hom_mat3d_rotate, hom_mat3d_translate, hom_mat3d_scale

Possible Successors

projective_trans_image, projective_trans_point_2d, projective_trans_region, projective_trans_contour_xld, hom_mat2d_invert

Module

Foundation

hom_vector_to_proj_hom_mat2d (: : Px, Py, Pw, Qx, Qy, Qw, Method : HomMat2D)

Compute a homogeneous transformation matrix using given point correspondences.

hom_vector_to_proj_hom_mat2d determines the homogeneous projective transformation matrix HomMat2D that optimally fulfills the following equations given by at least 4 point correspondences

$$\mathtt{HomMat2D} \cdot \begin{pmatrix} \mathtt{Px} \\ \mathtt{Py} \\ \mathtt{Pw} \end{pmatrix} = \begin{pmatrix} \mathtt{Qx} \\ \mathtt{Qy} \\ \mathtt{Qw} \end{pmatrix}$$

If fewer than 4 pairs of points (Px, Py, Pw), (Qx, Qy, Qw) are given, there exists no unique solution, if exactly 4 pairs are supplied the matrix HomMat2D transforms them in exactly the desired way, and if there are more than 4 point pairs given, hom_vector_to_proj_hom_mat2d seeks to minimize the transformation error. To achieve such a minimization, two different algorithms are available. The algorithm to use can be chosen using the parameter Method. For conventional geometric problems Method=*'normalized_dlt'* usually yields better results. However, if one of the coordinates Qw or Pw equals *0*, Method=*'dlt'* must be chosen.

In contrast to vector_to_proj_hom_mat2d, hom_vector_to_proj_hom_mat2d uses homogeneous coordinates for the points, and hence points at infinity (Pw = *0* or Qw = *0*) can be used to determine the transformation. If finite points are used, typically Pw and Qw are set to *1*. In this case, vector_to_proj_hom_mat2d can also be used. vector_to_proj_hom_mat2d has the advantage that one additional optimization method can be used and that the covariances of the points can be taken into account. If the correspondence between the points has not been determined, proj_match_points_ransac should be used to determine the correspondence as well as the transformation.

If the points to transform are specified in standard image coordinates, their *row* coordinates must be passed in Px and their *column* coordinates in Py. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain

a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Furthermore, it should be noted that if a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, it is assumed that the origin of the coordinate system of the homogeneous transformation matrix lies in the upper left corner of a pixel. The image processing operators that return point coordinates, however, assume a coordinate system in which the origin lies in the center of a pixel. Therefore, to obtain a consistent homogeneous transformation matrix, 0.5 must be added to the point coordinates before computing the transformation.

Parameters

▷ **Px** (input_control) . . . . . number-array ⇝ *real* / integer
Input points 1 (x coordinate).

▷ **Py** (input_control) . . . . . number-array ⇝ *real* / integer
Input points 1 (y coordinate).

▷ **Pw** (input_control) . . . . . number-array ⇝ *real* / integer
Input points 1 (w coordinate).

▷ **Qx** (input_control) . . . . . number-array ⇝ *real*
Input points 2 (x coordinate).

▷ **Qy** (input_control) . . . . . number-array ⇝ *real*
Input points 2 (y coordinate).

▷ **Qw** (input_control) . . . . . number-array ⇝ *real*
Input points 2 (w coordinate).

▷ **Method** (input_control) . . . . . string ⇝ *string*
Estimation algorithm.
Default: 'normalized_dlt'
List of values: `Method` ∈ {'normalized_dlt', 'dlt'}

▷ **HomMat2D** (output_control) . . . . . hom_mat2d ⇝ *real*
Homogeneous projective transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`proj_match_points_ransac`, `proj_match_points_ransac_guided`, `points_foerstner`, `points_harris`

Possible Successors

`projective_trans_image`, `projective_trans_image_size`, `projective_trans_region`, `projective_trans_contour_xld`, `projective_trans_point_2d`, `projective_trans_pixel`

Alternatives

`vector_to_proj_hom_mat2d`, `proj_match_points_ransac`, `proj_match_points_ransac_guided`

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2000.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

Calibration

point_line_to_hom_mat2d (: : TransformationType, Px, Py, L1x, L1y, L2x, L2y : HomMat2D)

Approximate an affine transformation from point-to-line correspondences.

point_line_to_hom_mat2d approximates an affine transformation from point-to-line correspondences and returns it as the homogeneous transformation matrix HomMat2D (see hom_mat2d_to_affine_par for the content of the homogeneous transformation matrix).

The points are passed in the tuples (Px,Py). Their corresponding lines are specified as two points on the line, which are passed in (L1x,L1y) and (L2x,L2y). Corresponding points and lines must be at the same index positions in these tuples.

The type of transformation to be approximated is determined with TransformationType. Each type of transformation requires a certain minimum number of point-to-line correspondences, as described in the following table:

TransformationType	Type of transformation	Minimum number of correspondences
'translation'	translation	2
'rigid'	rigid transformation	3
'similarity'	similarity transformation	4
'aniso'	anisotropic similarity transformation	5
'affine'	general affine transformation	6

The types of transformations that are supported can be described as follows:

'translation': A translation, i.e., a transformation that can be obtained as follows:

```
hom_mat2d_identity(HomMat2D)
hom_mat2d_translate(HomMat2D, Tx, Ty, HomMat2D)
```

This means that

$$\begin{aligned} \texttt{HomMat2D} &= \begin{bmatrix} 1 & 0 & \\ 0 & 1 & \mathbf{T} \\ 0 & 0 & 1 \end{bmatrix} \\ &= \mathbf{H}(\mathbf{T}) \end{aligned}$$

where $\mathbf{T}$ is the translation vector.

'rigid': A rigid transformation, i.e., a transformation that can be obtained as follows:

```
hom_mat2d_identity(HomMat2D)
hom_mat2d_rotate(HomMat2D, Phi, 0, 0, HomMat2D)
hom_mat2d_translate(HomMat2D, Tx, Ty, HomMat2D)
```

This means that

$$\begin{aligned} \texttt{HomMat2D} &= \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ 0\,0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & \\ 0 & 1 & \mathbf{T} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & 0 \\ & 0 \\ 0\,0 & 1 \end{bmatrix} \\ &= \mathbf{H}(\mathbf{T}) \cdot \mathbf{H}(\mathbf{R}) \end{aligned}$$

where $\mathbf{R}$ is a rotation matrix corresponding to the rotation angle Phi and $\mathbf{T}$ is the translation vector.

'similarity': A similarity transformation, i.e., a transformation that can be obtained as follows (note the identical scale factors in hom_mat2d_scale):

```
hom_mat2d_identity(HomMat2D)
hom_mat2d_scale(HomMat2D, S, S, 0, 0, HomMat2D)
hom_mat2d_rotate(HomMat2D, Phi, 0, 0, HomMat2D)
hom_mat2d_translate(HomMat2D, Tx, Ty, HomMat2D)
```

This means that

$$\begin{aligned}\texttt{HomMat2D} &= \begin{bmatrix} \mathbf{R}\cdot\mathbf{S} & \mathbf{T} \\ 0\;0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & \\ 0 & 1 & \mathbf{T} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & & 0 \\ & & 0 \\ 0\;0 & & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{S} & & 0 \\ & & 0 \\ 0\;0 & & 1 \end{bmatrix} \\ &= \mathbf{H}(\mathbf{T}) \cdot \mathbf{H}(\mathbf{R}) \cdot \mathbf{H}(\mathbf{S})\end{aligned}$$

where $\mathbf{S}$ is a scaling matrix with identical scaling `S` in the x and y directions, $\mathbf{R}$ is a rotation matrix corresponding to the rotation angle `Phi`, and $\mathbf{T}$ is the translation vector.

'aniso': An anisotropic similarity transformation, i.e., a transformation that can be obtained as follows (note the different scale factors in `hom_mat2d_scale`):

```
hom_mat2d_identity(HomMat2D)
hom_mat2d_scale(HomMat2D, Sx, Sy, 0, 0, HomMat2D)
hom_mat2d_rotate(HomMat2D, Phi, 0, 0, HomMat2D)
hom_mat2d_translate(HomMat2D, Tx, Ty, HomMat2D)
```

This means that

$$\begin{aligned}\texttt{HomMat2D} &= \begin{bmatrix} \mathbf{R}\cdot\mathbf{S} & \mathbf{T} \\ 0\;0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & \\ 0 & 1 & \mathbf{T} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & & 0 \\ & & 0 \\ 0\;0 & & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \mathbf{H}(\mathbf{T}) \cdot \mathbf{H}(\mathbf{R}) \cdot \mathbf{H}(\mathbf{S})\end{aligned}$$

where $\mathbf{S}$ is a scaling matrix with non-identical scaling `Sx` and `Sy` in the x and y directions, $\mathbf{R}$ is a rotation matrix corresponding to the rotation angle `Phi`, and $\mathbf{T}$ is the translation vector.

'affine': A general affine transformation, i.e., a transformation that can be obtained as follows:

```
hom_mat2d_identity(HomMat2D)
hom_mat2d_scale(HomMat2D, Sx, Sy, 0, 0, HomMat2D)
hom_mat2d_slant(HomMat2D, Theta, 'y', 0, 0, HomMat2D)
hom_mat2d_rotate(HomMat2D, Phi, 0, 0, HomMat2D)
hom_mat2d_translate(HomMat2D, Tx, Ty, HomMat2D)
```

This means that

$$\texttt{HomMat2D} = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ 0 & 0 & 1 \end{bmatrix}$$

where $h_{1,1}$–$h_{2,3}$ are arbitrary numbers.

The transformation is computed by minimizing the sum of the squared distances of the points (`Px`,`Py`) transformed with the computed transformation to the lines given by (`L1x`,`L1y`) and (`L2x`,`L2y`). The lines are regarded as lines with infinite extent. This means that the points (`Px`,`Py`) transformed with the transformation `HomMat2D` may not lie "between" (`L1x`,`L1y`) and (`L2x`,`L2y`). An exception to this rule occurs for rigid and anisotropic similarity transformations that are determined from the minimum number of point-to-line correspondences (3 and 5, respectively). In this case, there are in general two possible solutions that both yield an error of 0. The algorithm returns the solution for which the transformed points lie as close as possible to the line *segments* given by (`L1x`,`L1y`) and (`L2x`,`L2y`). If a unique solution is desired, one additional point-to-line correspondence should be used (i.e., 4 or 6, respectively).

`HomMat2D` can be used directly with operators that transform data using affine transformations, e.g., `affine_trans_image`.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or

any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Furthermore, it should be noted that if a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, it is assumed that the origin of the coordinate system of the homogeneous transformation matrix lies in the upper left corner of a pixel. The image processing operators that return point coordinates, however, assume a coordinate system in which the origin lies in the center of a pixel. Therefore, to obtain a consistent homogeneous transformation matrix, 0.5 must be added to the point coordinates before computing the transformation.

Parameters

- ▷ **TransformationType** (input_control) ..... string ⇝ *string*
  Type of the transformation to compute.
  Default: 'rigid'
  List of values: TransformationType ∈ {'translation', 'rigid', 'similarity', 'aniso', 'affine'}
- ▷ **Px** (input_control) ..... point.x-array ⇝ *real*
  X coordinates of the original points.
- ▷ **Py** (input_control) ..... point.y-array ⇝ *real*
  Y coordinates of the original points.
- ▷ **L1x** (input_control) ..... point.x-array ⇝ *real*
  X coordinates of the first point on the corresponding line.
- ▷ **L1y** (input_control) ..... point.y-array ⇝ *real*
  Y coordinates of the first point on the corresponding line.
- ▷ **L2x** (input_control) ..... point.x-array ⇝ *real*
  X coordinates of the second point on the corresponding line.
- ▷ **L2y** (input_control) ..... point.y-array ⇝ *real*
  Y coordinates of the second point on the corresponding line.
- ▷ **HomMat2D** (output_control) ..... hom_mat2d ⇝ *real*
  Output transformation matrix.

Example

```
* Use point_line_to_hom_mat2d for alignment.
* Read the reference image.
read_image (Image, ReferenceFileName)
* Set up the metrology model with four lines.  Four lines are used
* since this is the minimum number of point-to-line correspondences
* that results in a unique rigid transformation.
Row1 := [RowB1,RowB2,RowB3,RowB4]
Col1 := [ColB1,ColB2,ColB3,ColB4]
Row2 := [RowE1,RowE2,RowE3,RowE4]
Col2 := [ColE1,ColE2,ColE3,ColE4]
create_metrology_model (MetrologyHandle)
add_metrology_object_line_measure (MetrologyHandle, Row1, Col1, \
                                   Row2, Col2, 40, 5, 1, 30, \
                                   [], [], Index)
* Apply the metrology model to the reference image and read out
* the results.
apply_metrology_model (Image, MetrologyHandle)
get_metrology_object_result (MetrologyHandle, 'all', 'all', \
                             'result_type', 'row_begin', \
                             RowBegin)
get_metrology_object_result (MetrologyHandle, 'all', 'all', \
                             'result_type', 'column_begin', \
                             ColBegin)
get_metrology_object_result (MetrologyHandle, 'all', 'all', \
```

```
                              'result_type', 'row_end', \
                              RowEnd)
get_metrology_object_result (MetrologyHandle, 'all', 'all', \
                              'result_type', 'column_end', \
                              ColEnd)
* The reference points of the model are the center points of the
* detected line segments.  They will be used to compute the
* transformation from the current image to the reference image
* using point_line_to_hom_mat2d below.
RowRef := 0.5*(RowBegin+RowEnd)
ColRef := 0.5*(ColBegin+ColEnd)
for I := 1 to |FileNames|-1 by 1
    read_image (Image, FileNames[I])
    * Apply the metrology model to the current image and read out
    * the line segment coordinates.
    apply_metrology_model (Image, MetrologyHandle)
    get_metrology_object_result (MetrologyHandle, 'all', 'all', \
                                 'result_type', 'row_begin', \
                                 RowBegin)
    get_metrology_object_result (MetrologyHandle, 'all', 'all', \
                                 'result_type', 'column_begin', \
                                 ColBegin)
    get_metrology_object_result (MetrologyHandle, 'all', 'all', \
                                 'result_type', 'row_end', \
                                 RowEnd)
    get_metrology_object_result (MetrologyHandle, 'all', 'all', \
                                 'result_type', 'column_end', \
                                 ColEnd)
    * Determine a rigid transformation based on the point-to-line
    * correspondences from the reference points to the extracted
    * lines.  Note that this determines a transformation from the
    * reference points to the lines in the current image.
    * Therefore, we must invert this transformation to obtain the
    * transformation from the current image to the rerefence image.
    point_line_to_hom_mat2d ('rigid', RowRef+0.5, ColRef+0.5, \
                             RowBegin+0.5, ColBegin+0.5, \
                             RowEnd+0.5, ColEnd+0.5, HomMat2D)
    hom_mat2d_invert (HomMat2D, HomMat2DInvert)
    affine_trans_image (Image, ImageTrans, HomMat2DInvert, \
                        'constant', 'false')
    * Now that the current image has been aligned with the
    * reference image, we can do some processing based on the
    * aligned image ImageTrans.
    * [...]
endfor
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

affine_trans_image, affine_trans_image_size, affine_trans_region, affine_trans_contour_xld, affine_trans_polygon_xld, affine_trans_point_2d

See also

vector_to_hom_mat2d, vector_to_aniso, vector_to_similarity, vector_to_rigid

Module

Foundation

projective_trans_pixel (: : HomMat2D, Row, Col : RowTrans, ColTrans)

Project pixel coordinates using a homogeneous projective transformation matrix.

projective_trans_pixel applies the homogeneous projective transformation matrix HomMat2D to all input pixels (Row,Col) and returns an array of output pixels (RowTrans,ColTrans). The transformation is described by the homogeneous transformation matrix given in HomMat2D.

The difference between projective_trans_pixel and projective_trans_point_2d lies in the used coordinate system: projective_trans_pixel uses a coordinate system with origin in the upper left corner of the image, while projective_trans_point_2d uses the standard image coordinate system, whose origin lies in the middle of the upper left pixel and which is also used by operators like area_center.

projective_trans_pixel corresponds to the following steps (input and output points as homogeneous vectors):

$$\begin{pmatrix} \mathtt{RTrans} \\ \mathtt{CTrans} \\ \mathtt{WTrans} \end{pmatrix} = \mathtt{HomMat2D} \cdot \begin{pmatrix} \mathtt{Row} \\ \mathtt{Col} \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} \mathtt{RowTrans} \\ \mathtt{ColTrans} \end{pmatrix} = \begin{pmatrix} \frac{\mathtt{RTrans}}{\mathtt{WTrans}} \\ \frac{\mathtt{CTrans}}{\mathtt{WTrans}} \end{pmatrix}$$

If a point at infinity (WTrans $= 0$) is created by the transformation, an error is returned.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

- ▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
  Homogeneous projective transformation matrix.
- ▷ **Row** (input_control) . . . . . point.x(-array) ⇝ *real* / integer
  Input pixel(s) (row coordinate).
  Default: 64
  Suggested values: Row ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Col** (input_control) . . . . . point.y(-array) ⇝ *real* / integer
  Input pixel(s) (column coordinate).
  Default: 64
  Suggested values: Col ∈ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **RowTrans** (output_control) . . . . . point.x(-array) ⇝ *real*
  Output pixel(s) (row coordinate).
- ▷ **ColTrans** (output_control) . . . . . point.y(-array) ⇝ *real*
  Output pixel(s) (column coordinate).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

vector_to_proj_hom_mat2d, hom_vector_to_proj_hom_mat2d, proj_match_points_ransac, proj_match_points_ransac_guided, hom_mat3d_project

See also

projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_point_2d

Module

Foundation

projective_trans_point_2d (: : HomMat2D, Px, Py, Pw : Qx, Qy, Qw)

Project a homogeneous 2D point using a projective transformation matrix.

projective_trans_point_2d applies the homogeneous projective transformation matrix HomMat2D to all homogeneous input points (Px,Py,Pw) and returns an array of homogeneous output points (Qx,Qy,Qw). The transformation is described by the homogeneous transformation matrix given in HomMat2D. This corresponds to the following equation (input and output points as homogeneous vectors):

$$\begin{pmatrix} Qx \\ Qy \\ Qw \end{pmatrix} = HomMat2D \cdot \begin{pmatrix} Px \\ Py \\ Pw \end{pmatrix}$$

To transform the homogeneous coordinates to Euclidean coordinates, they have to be divided by Qw:

$$\begin{pmatrix} Ex \\ Ey \end{pmatrix} = \begin{pmatrix} \frac{Qx}{Qw} \\ \frac{Qy}{Qw} \end{pmatrix}$$

If the points to transform are specified in standard image coordinates, their *row* coordinates must be passed in Px and their *column* coordinates in Py. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

Parameters

- ▷ **HomMat2D** (input_control) . . . hom_mat2d ⇝ *real*
  Homogeneous projective transformation matrix.
- ▷ **Px** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (x coordinate).
- ▷ **Py** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (y coordinate).
- ▷ **Pw** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (w coordinate).
- ▷ **Qx** (output_control) . . . number(-array) ⇝ *real*
  Output point (x coordinate).
- ▷ **Qy** (output_control) . . . number(-array) ⇝ *real*
  Output point (y coordinate).
- ▷ **Qw** (output_control) . . . number(-array) ⇝ *real*
  Output point (w coordinate).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

vector_to_proj_hom_mat2d, hom_vector_to_proj_hom_mat2d, proj_match_points_ransac, proj_match_points_ransac_guided, hom_mat3d_project

See also

projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_pixel

Module

Foundation

serialize_hom_mat2d (: : HomMat2D : SerializedItemHandle)

Serialize a homogeneous 2D transformation matrix.

serialize_hom_mat2d serializes the data of a homogeneous 2D transformation matrix (see fwrite_serialized_item for an introduction of the basic principle of serialization). The transformation matrix is defined by the handle HomMat2D. The serialized transformation matrix is returned by the handle SerializedItemHandle and can be deserialized by deserialize_hom_mat2d.

Parameters

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Transformation matrix.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_hom_mat2d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_hom_mat2d

Module

Foundation

vector_angle_to_rigid (: : Row1, Column1, Angle1, Row2, Column2, Angle2 : HomMat2D)

Compute a rigid affine transformation from points and angles.

vector_angle_to_rigid computes a rigid affine transformation, i.e., a transformation consisting of a rotation and a translation, from a point correspondence and two corresponding angles and returns it as the homogeneous transformation matrix HomMat2D. The matrix consists of 2 components: a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{T}$ (also see hom_mat2d_rotate and hom_mat2d_translate):

$$\mathtt{HomMat2D} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ 0\,0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \\ 0 & 1 & \mathbf{T} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & 0 \\ 0\,0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{T}) \cdot \mathbf{H}(\mathbf{R})$$

The coordinates of the original point are passed in (Row1,Column1), while the corresponding angle is passed in Angle1. The coordinates of the transformed point are passed in (Row2,Column2), while the corresponding angle is passed in Angle2. The following equation describes the transformation of the point using homogeneous vectors:

$$\begin{pmatrix} \mathtt{Row2} \\ \mathtt{Column2} \\ 1 \end{pmatrix} = \mathtt{HomMat2D} \cdot \begin{pmatrix} \mathtt{Row1} \\ \mathtt{Column1} \\ 1 \end{pmatrix}$$

In particular, the operator vector_angle_to_rigid is useful to construct a rigid affine transformation from the results of the matching operators (e.g., find_ncc_model), which transforms a reference image to the current image or (if the parameters are passed in reverse order) from the current image to the reference image.

HomMat2D can be used directly with operators that transform data using affine transformations, e.g., affine_trans_image.

Parameters

- ▷ **Row1** (input_control) . . . . . point.y ⇝ *real* / integer
  Row coordinate of the original point.
- ▷ **Column1** (input_control) . . . . . point.x ⇝ *real* / integer
  Column coordinate of the original point.
- ▷ **Angle1** (input_control) . . . . . angle.rad ⇝ *real* / integer
  Angle of the original point.
- ▷ **Row2** (input_control) . . . . . point.y ⇝ *real* / integer
  Row coordinate of the transformed point.
- ▷ **Column2** (input_control) . . . . . point.x ⇝ *real* / integer
  Column coordinate of the transformed point.
- ▷ **Angle2** (input_control) . . . . . angle.rad ⇝ *real* / integer
  Angle of the transformed point.
- ▷ **HomMat2D** (output_control) . . . . . hom_mat2d ⇝ *real*
  Output transformation matrix.

Example

```
read_image (Image, 'face_masks/face_mask_01')
* Prepare NCC matching.
gen_rectangle2 (ROI, 616.5, 708.5, rad(-82.4054), 50, 35)
reduce_domain (Image, ROI, ImageReduced)
create_ncc_model (ImageReduced, 'auto', rad(0), rad(360), 'auto',\
                  'use_polarity', ModelID)
read_image (SearchImage, 'face_masks/face_mask_02')
find_ncc_model (SearchImage, ModelID, rad(0), rad(360), 0.7, 1, 0.5,\
                'true', 0, Row, Column, Angle, Score)
get_ncc_model_region (ModelRegion, ModelID)
gen_contour_region_xld (ModelRegion, ModelContours, 'border_holes')
* Create transformation matrix for found match.
vector_angle_to_rigid (0, 0, 0, Row, Column, Angle, HomMat2D)
affine_trans_contour_xld (ModelContours, ContoursAffineTrans, HomMat2D)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_ncc_model_region

Possible Successors

hom_mat2d_invert, affine_trans_image, affine_trans_region, affine_trans_contour_xld, affine_trans_polygon_xld, affine_trans_point_2d

Alternatives

vector_to_rigid

See also

vector_field_to_hom_mat2d

Module

Foundation

vector_field_to_hom_mat2d (VectorField : : : HomMat2D)

Approximate an affine map from a displacement vector field.

vector_field_to_hom_mat2d approximates an affine map from the displacement vector field VectorField. The affine map is returned in HomMat2D.

If the displacement vector field has been computed from the original image I_{orig} and the second image I_{res}, the internally stored transformation matrix (see affine_trans_image) contains a map that describes how to transform the first image I_{orig} to the second image I_{res}. Note that the VectorField must be in relative coordinates as returned by optical_flow_mg.

Parameters

▷ **VectorField** (input_object) . . . . . singlechannelimage ⇝ *object :* vector_field
Input image.

▷ **HomMat2D** (output_control) . . . . . hom_mat2d ⇝ *real*
Output transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

optical_flow_mg

Possible Successors

affine_trans_image

Alternatives

vector_to_hom_mat2d

Module

Foundation

vector_to_aniso (: : Px, Py, Qx, Qy : HomMat2D)

Approximate an anisotropic similarity transformation from point correspondences.

vector_to_aniso approximates an anisotropic similarity transformation, i.e., a transformation consisting of a rotation, a non-uniform scaling, and a translation, from at least three point correspondences and returns it as the homogeneous transformation matrix HomMat2D. The matrix consists of 3 components: a scaling matrix $\mathbf{S}$ with non-identical scaling in the x and y directions, a rotation matrix $\mathbf{R}$, and a translation vector $\mathbf{T}$ (also see hom_mat2d_scale, hom_mat2d_rotate, and hom_mat2d_translate):

$$\texttt{HomMat2D} = \begin{bmatrix} \mathbf{R}\cdot\mathbf{S} & \mathbf{T} \\ 0\ \ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \mathbf{T} \\ 0 & 1 & \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & & 0 \\ & & 0 \\ 0\,0 & & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{T})\cdot\mathbf{H}(\mathbf{R})\cdot\mathbf{H}(\mathbf{S})$$

The point correspondences are passed in the tuples (Px,Py) and (Qx,Qy), where corresponding points must be at the same index positions in the tuples. The transformation is always overdetermined. Therefore, the returned transformation is the transformation that minimizes the distances between the original points (Px,Py) and the transformed points (Qx,Qy), as described in the following equation (points as homogeneous vectors):

$$\sum_i \left\| \begin{pmatrix} \texttt{Qx}[i] \\ \texttt{Qy}[i] \\ 1 \end{pmatrix} - \texttt{HomMat2D} \cdot \begin{pmatrix} \texttt{Px}[i] \\ \texttt{Py}[i] \\ 1 \end{pmatrix} \right\|^2 = \text{minimum}$$

HomMat2D can be used directly with operators that transform data using affine transformations, e.g., affine_trans_image.

In an anisotropic similarity transformation, as defined above, the points are first scaled and then rotated. Sometimes, a transformation in which the points are first rotated and then scaled is useful. This kind of transformation can be computed with vector_to_aniso by passing the point correspondences in the opposite order, i.e., the points (Px,Py) are passed in the parameters (Qx,Qy) and vice versa. The resulting transformation must then be inverted with hom_mat2d_invert.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Furthermore, it should be noted that if a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, it is assumed that the origin of the coordinate system of the homogeneous transformation matrix lies in the upper left corner of a pixel. The image processing operators that return point coordinates, however, assume a coordinate system in which the origin lies in the center of a pixel. Therefore, to obtain a consistent homogeneous transformation matrix, 0.5 must be added to the point coordinates before computing the transformation.

Parameters

- ▷ **Px** (input_control) ..... point.x-array ⇝ *real*
  X coordinates of the original points.
- ▷ **Py** (input_control) ..... point.y-array ⇝ *real*
  Y coordinates of the original points.
- ▷ **Qx** (input_control) ..... point.x-array ⇝ *real*
  X coordinates of the transformed points.
- ▷ **Qy** (input_control) ..... point.y-array ⇝ *real*
  Y coordinates of the transformed points.
- ▷ **HomMat2D** (output_control) ..... hom_mat2d ⇝ *real*
  Output transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

hom_mat2d_invert, affine_trans_image, affine_trans_image_size, affine_trans_region, affine_trans_contour_xld, affine_trans_polygon_xld, affine_trans_point_2d

Alternatives

vector_to_hom_mat2d, vector_to_similarity, vector_to_rigid

See also

vector_field_to_hom_mat2d, point_line_to_hom_mat2d

Module

Foundation

vector_to_hom_mat2d (: : Px, Py, Qx, Qy : HomMat2D)

Approximate an affine transformation from point correspondences.

vector_to_hom_mat2d approximates an affine transformation from at least three point correspondences and returns it as the homogeneous transformation matrix HomMat2D (see hom_mat2d_to_affine_par for the content of the homogeneous transformation matrix).

The point correspondences are passed in the tuples (Px,Py) and (Qx,Qy), where corresponding points must be at the same index positions in the tuples. If more than three point correspondences are passed, the transformation is overdetermined. In this case, the returned transformation is the transformation that minimizes the distances between the input points (Px,Py) and the transformed points (Qx,Qy), as described in the following equation (points as homogeneous vectors):

$$\sum_i \left\| \begin{pmatrix} \mathtt{Qx}[i] \\ \mathtt{Qy}[i] \\ 1 \end{pmatrix} - \mathtt{HomMat2D} \cdot \begin{pmatrix} \mathtt{Px}[i] \\ \mathtt{Py}[i] \\ 1 \end{pmatrix} \right\|^2 = \text{minimum}$$

HomMat2D can be used directly with operators that transform data using affine transformations, e.g., affine_trans_image.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Furthermore, it should be noted that if a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, it is assumed that the origin of the coordinate system of the homogeneous transformation matrix lies in the upper left corner of a pixel. The image processing operators that return point coordinates, however, assume a coordinate system in which the origin lies in the center of a pixel. Therefore, to obtain a consistent homogeneous transformation matrix, 0.5 must be added to the point coordinates before computing the transformation.

Parameters

- ▷ **Px** (input_control) . . . point.x-array ⇝ *real*
  X coordinates of the original points.
- ▷ **Py** (input_control) . . . point.y-array ⇝ *real*
  Y coordinates of the original points.
- ▷ **Qx** (input_control) . . . point.x-array ⇝ *real*
  X coordinates of the transformed points.
- ▷ **Qy** (input_control) . . . point.y-array ⇝ *real*
  Y coordinates of the transformed points.
- ▷ **HomMat2D** (output_control) . . . hom_mat2d ⇝ *real*
  Output transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

affine_trans_image, affine_trans_image_size, affine_trans_region, affine_trans_contour_xld, affine_trans_polygon_xld, affine_trans_point_2d

Alternatives

vector_to_aniso, vector_to_similarity, vector_to_rigid

See also

vector_field_to_hom_mat2d, point_line_to_hom_mat2d

Module

Foundation

vector_to_proj_hom_mat2d (: : Px, Py, Qx, Qy, Method, CovXX1, CovYY1, CovXY1, CovXX2, CovYY2, CovXY2 : HomMat2D, Covariance)

Compute a projective transformation matrix using given point correspondences.

vector_to_proj_hom_mat2d determines the homogeneous projective transformation matrix HomMat2D that optimally fulfills the following equations given by at least 4 point correspondences

$$\mathtt{HomMat2D} \cdot \begin{pmatrix} \mathtt{Px} \\ \mathtt{Py} \\ 1 \end{pmatrix} = \begin{pmatrix} \mathtt{Qx} \\ \mathtt{Qy} \\ 1 \end{pmatrix}.$$

If fewer than 4 pairs of points (Px,Py), (Qx,Qy) are given, there exists no unique solution, if exactly 4 pairs are supplied the matrix HomMat2D transforms them in exactly the desired way, and if there are more than 4 point pairs given, vector_to_proj_hom_mat2d seeks to minimize the transformation error. To achieve such a minimization, several different algorithms are available. The algorithm to use can be chosen using the parameter Method. Method=*'dlt'* uses a fast and simple, but also rather inaccurate error estimation algorithm while Method=*'normalized_dlt'* offers a good compromise between speed and accuracy. Finally, Method=*'gold_standard'* performs a mathematically optimal but slower optimization.

If *'gold_standard'* is used and the input points have been obtained from an operator like points_foerstner, which provides a covariance matrix for each of the points, which specifies the accuracy of the points, this can be taken into account by using the input parameters CovYY1, CovXX1, CovXY1 for the points in the first image and CovYY2, CovXX2, CovXY2 for the points in the second image. The covariances are symmetric 2×2 matrices. CovXX1/CovXX2 and CovYY1/CovYY2 are a list of diagonal entries while CovXY1/CovXY2 contains the non-diagonal entries which appear twice in a symmetric matrix. If a different Method than *'gold_standard'* is used or the covariances are unknown the covariance parameters can be left empty.

In contrast to hom_vector_to_proj_hom_mat2d, points at infinity cannot be used to determine the transformation in vector_to_proj_hom_mat2d. If this is necessary, hom_vector_to_proj_hom_mat2d must be used. If the correspondence between the points has not been determined, proj_match_points_ransac should be used to determine the correspondence as well as the transformation.

If the points to transform are specified in standard image coordinates, their *row* coordinates must be passed in Px and their *column* coordinates in Py. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Furthermore, it should be noted that if a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, it is assumed that the origin of the coordinate system of the homogeneous transformation matrix lies in the upper left corner of a pixel. The image processing operators that return point coordinates, however, assume a coordinate system in which the origin lies in the center of a pixel. Therefore, to obtain a consistent homogeneous transformation matrix, 0.5 must be added to the point coordinates before computing the transformation.

Parameters

- ▷ **Px** (input_control) ..... point.x-array ⇝ *real* / integer
  Input points in image 1 (row coordinate).
- ▷ **Py** (input_control) ..... point.y-array ⇝ *real* / integer
  Input points in image 1 (column coordinate).
- ▷ **Qx** (input_control) ..... point.x-array ⇝ *real*
  Input points in image 2 (row coordinate).
- ▷ **Qy** (input_control) ..... point.y-array ⇝ *real*
  Input points in image 2 (column coordinate).

▷ **Method** (input_control) . . . . . string ⇝ *string*
Estimation algorithm.
Default: 'normalized_dlt'
List of values: Method ∈ {'normalized_dlt', 'gold_standard', 'dlt'}

▷ **CovXX1** (input_control) . . . . . real-array ⇝ *real*
Row coordinate variance of the points in image 1.
Default: []

▷ **CovYY1** (input_control) . . . . . real-array ⇝ *real*
Column coordinate variance of the points in image 1.
Default: []

▷ **CovXY1** (input_control) . . . . . real-array ⇝ *real*
Covariance of the points in image 1.
Default: []

▷ **CovXX2** (input_control) . . . . . real-array ⇝ *real*
Row coordinate variance of the points in image 2.
Default: []

▷ **CovYY2** (input_control) . . . . . real-array ⇝ *real*
Column coordinate variance of the points in image 2.
Default: []

▷ **CovXY2** (input_control) . . . . . real-array ⇝ *real*
Covariance of the points in image 2.
Default: []

▷ **HomMat2D** (output_control) . . . . . hom_mat2d ⇝ *real*
Homogeneous projective transformation matrix.

▷ **Covariance** (output_control) . . . . . real-array ⇝ *real*
9×9 covariance matrix of the projective transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

proj_match_points_ransac, proj_match_points_ransac_guided, points_foerstner, points_harris

Possible Successors

projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_point_2d, projective_trans_pixel

Alternatives

hom_vector_to_proj_hom_mat2d, proj_match_points_ransac, proj_match_points_ransac_guided

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2000.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

Calibration

vector_to_proj_hom_mat2d_distortion (: : Points1Row, Points1Col, Points2Row, Points2Col, CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2, ImageWidth, ImageHeight, Method : HomMat2D, Kappa, Error)

Compute a projective transformation matrix and the radial distortion coefficient using given image point correspondences.

vector_to_proj_hom_mat2d_distortion determines the projective transformation matrix HomMat2D and the radial distortion coefficient Kappa (κ) from given point correspondences (Points1Row,Points1Col), (Points2Row,Points2Col) that optimally fulfill the following equation:

$$\begin{pmatrix} r_2 \\ c_2 \\ 1 \end{pmatrix} = \text{HomMat2D} \cdot \begin{pmatrix} r_1 \\ c_1 \\ 1 \end{pmatrix}.$$

Here, (r_1, c_1) and (r_2, c_2) denote image points that are obtained by undistorting the input image points with the division model (see Calibration):

$$r = \frac{\tilde{r}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)} \qquad c = \frac{\tilde{c}}{1 + \kappa(\tilde{r}^2 + \tilde{c}^2)}$$

Here, $(\tilde{r}_1, \tilde{c}_1) = (\text{Points1Row} - 0.5(\text{ImageHeight} - 1), \text{Points1Col} - 0.5(\text{ImageWidth} - 1))$ and $(\tilde{r}_2, \tilde{c}_2) = (\text{Points2Row} - 0.5(\text{ImageHeight} - 1), \text{Points2Col} - 0.5(\text{ImageWidth} - 1))$ denote the distorted image points, specified relative to the image center. Thus, vector_to_proj_hom_mat2d_distortion assumes that the principal point of the camera, i.e., the center of the radial distortions, lies at the center of the image. Note that the images from which the points (Points1Row,Points1Col) and (Points2Row,Points2Col) are extracted have to be obtained with the same camera.

The returned Kappa can be used to construct camera parameters that can be used to rectify images or points (see change_radial_distortion_cam_par, change_radial_distortion_image, and change_radial_distortion_points):

$$\begin{aligned} \text{CamPar} = [&'\textit{area\_scan\_telecentric\_division}', 1.0, \text{Kappa}, 1.0, 1.0, 0.5(\text{ImageWidth} - 1), \\ &0.5(\text{ImageHeight} - 1), \text{ImageWidth}, \text{ImageHeight}] \end{aligned}$$

The minimum number of required point correspondences is five. vector_to_proj_hom_mat2d_distortion seeks to minimize the transformation error between the point correspondences based on the above equations. To achieve such a minimization, two different algorithms are available. The algorithm to use can be chosen using the parameter Method. For Method = *'linear'*, a linear algorithm that minimizes an algebraic error based on the above equations is used. This algorithm is very fast and returns accurate results for small to moderate noise of the point coordinates and for most distortions (except for small distortions). For Method = *'gold_standard'*, a mathematically optimal but slower optimization is used, which minimizes the geometric reprojection error. In general, it is preferable to use Method = *'gold_standard'*.

If *'gold_standard'* is used, the covariances of the image points (CovRR1, CovRC1, CovCC1, CovRR2, CovRC2, CovCC2) can be incorporated into the computation. They can be provided, for example, by the operator points_foerstner. If the point covariances are unknown, which is the default, empty tuples are passed. In this case, the optimization algorithm internally assumes uniform and equal covariances for all points. If *'linear'* is used, the covariances are ignored and the covariance parameters can be left empty.

The value Error indicates the overall quality of the optimization procedure and is the mean symmetric euclidean distance in pixels between the points and their corresponding transformed points.

If the correspondence between the points has not been determined, proj_match_points_distortion_ransac or proj_match_points_distortion_ransac_guided should be used to determine the correspondence as well as the transformation.

Attention

It should be noted that if a homogeneous transformation matrix is used to transform images, regions, XLD contours,

or any other data that has been extracted from images, it is assumed that the origin of the coordinate system of the homogeneous transformation matrix lies in the upper left corner of a pixel. The image processing operators that return point coordinates, however, assume a coordinate system in which the origin lies in the center of a pixel. Therefore, to obtain a consistent homogeneous transformation matrix, 0.5 must be added to the point coordinates before computing the transformation.

Parameters

▷ **Points1Row** (input_control) . . . . . point.y-array ⇝ *real* / integer
Input points in image 1 (row coordinate).
Restriction: length(Points1Row) >= 5

▷ **Points1Col** (input_control) . . . . . point.x-array ⇝ *real* / integer
Input points in image 1 (column coordinate).
Restriction: length(Points1Col) == length(Points1Row)

▷ **Points2Row** (input_control) . . . . . point.y-array ⇝ *real* / integer
Input points in image 2 (row coordinate).
Restriction: length(Points2Row) == length(Points1Row)

▷ **Points2Col** (input_control) . . . . . point.x-array ⇝ *real* / integer
Input points in image 2 (column coordinate).
Restriction: length(Points2Col) == length(Points1Row)

▷ **CovRR1** (input_control) . . . . . number-array ⇝ *real* / integer
Row coordinate variance of the points in image 1.
Default: []

▷ **CovRC1** (input_control) . . . . . number-array ⇝ *real* / integer
Covariance of the points in image 1.
Default: []

▷ **CovCC1** (input_control) . . . . . number-array ⇝ *real* / integer
Column coordinate variance of the points in image 1.
Default: []

▷ **CovRR2** (input_control) . . . . . number-array ⇝ *real* / integer
Row coordinate variance of the points in image 2.
Default: []

▷ **CovRC2** (input_control) . . . . . number-array ⇝ *real* / integer
Covariance of the points in image 2.
Default: []

▷ **CovCC2** (input_control) . . . . . number-array ⇝ *real* / integer
Column coordinate variance of the points in image 2.
Default: []

▷ **ImageWidth** (input_control) . . . . . integer ⇝ *integer*
Width of the images from which the points were extracted.
Restriction: ImageWidth > 0

▷ **ImageHeight** (input_control) . . . . . integer ⇝ *integer*
Height of the images from which the points were extracted.
Restriction: ImageHeight > 0

▷ **Method** (input_control) . . . . . string ⇝ *string*
Estimation algorithm.
Default: 'gold_standard'
List of values: `Method` ∈ {'linear', 'gold_standard'}

▷ **HomMat2D** (output_control) . . . . . hom_mat2d ⇝ *real*
Homogeneous projective transformation matrix.

▷ **Kappa** (output_control) . . . . . real ⇝ *real*
Computed radial distortion coefficient.

▷ **Error** (output_control) . . . . . real ⇝ *real*
Root-Mean-Square transformation error.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

proj_match_points_distortion_ransac, proj_match_points_distortion_ransac_guided, points_foerstner, points_harris

Possible Successors

change_radial_distortion_cam_par, change_radial_distortion_image, change_radial_distortion_points, projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_contour_xld, projective_trans_point_2d, projective_trans_pixel

Alternatives

vector_to_proj_hom_mat2d, hom_vector_to_proj_hom_mat2d

See also

calibrate_cameras

References

Richard Hartley, Andrew Zisserman: "Multiple View Geometry in Computer Vision"; Cambridge University Press, Cambridge; 2003.

Olivier Faugeras, Quang-Tuan Luong: "The Geometry of Multiple Images: The Laws That Govern the Formation of Multiple Images of a Scene and Some of Their Applications"; MIT Press, Cambridge, MA; 2001.

Module

Calibration

vector_to_rigid (: : Px, Py, Qx, Qy : HomMat2D)

Approximate a rigid affine transformation from point correspondences.

vector_to_rigid approximates a rigid affine transformation, i.e., a transformation consisting of a rotation and a translation, from at least two point correspondences and returns it as the homogeneous transformation matrix HomMat2D. The matrix consists of 2 components: a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{T}$ (also see hom_mat2d_rotate and hom_mat2d_translate):

$$\texttt{HomMat2D} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ 0\,0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \mathbf{T} \\ 0 & 1 & \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & 0 \\ & 0 \\ 0\,0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{T}) \cdot \mathbf{H}(\mathbf{R})$$

The point correspondences are passed in the tuples (Px, Py) and (Qx,Qy), where corresponding points must be at the same index positions in the tuples. The transformation is always overdetermined. Therefore, the returned transformation is the transformation that minimizes the distances between the original points (Px,Py) and the transformed points (Qx,Qy), as described in the following equation (points as homogeneous vectors):

$$\sum_i \left\| \begin{pmatrix} \texttt{Qx}[i] \\ \texttt{Qy}[i] \\ 1 \end{pmatrix} - \texttt{HomMat2D} \cdot \begin{pmatrix} \texttt{Px}[i] \\ \texttt{Py}[i] \\ 1 \end{pmatrix} \right\|^2 = \text{minimum}$$

HomMat2D can be used directly with operators that transform data using affine transformations, e.g., affine_trans_image.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain

a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Furthermore, it should be noted that if a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, it is assumed that the origin of the coordinate system of the homogeneous transformation matrix lies in the upper left corner of a pixel. The image processing operators that return point coordinates, however, assume a coordinate system in which the origin lies in the center of a pixel. Therefore, to obtain a consistent homogeneous transformation matrix, 0.5 must be added to the point coordinates before computing the transformation.

Parameters

- ▷ **Px** (input_control) . . . . . point.x-array ⇝ *real*
  X coordinates of the original points.
- ▷ **Py** (input_control) . . . . . point.y-array ⇝ *real*
  Y coordinates of the original points.
- ▷ **Qx** (input_control) . . . . . point.x-array ⇝ *real*
  X coordinates of the transformed points.
- ▷ **Qy** (input_control) . . . . . point.y-array ⇝ *real*
  Y coordinates of the transformed points.
- ▷ **HomMat2D** (output_control) . . . . . hom_mat2d ⇝ *real*
  Output transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

affine_trans_image, affine_trans_image_size, affine_trans_region, affine_trans_contour_xld, affine_trans_polygon_xld, affine_trans_point_2d

Alternatives

vector_to_hom_mat2d, vector_to_aniso, vector_to_similarity

See also

vector_field_to_hom_mat2d, point_line_to_hom_mat2d

Module

Foundation

vector_to_similarity (: : Px, Py, Qx, Qy : HomMat2D)

Approximate an similarity transformation from point correspondences.

vector_to_similarity approximates a similarity transformation, i.e., a transformation consisting of a uniform scaling, a rotation, and a translation, from at least two point correspondences and returns it as the homogeneous transformation matrix HomMat2D. The matrix consists of 3 components: a scaling matrix $\mathbf{S}$ with identical scaling in the x and y directions, a rotation matrix $\mathbf{R}$, and a translation vector $\mathbf{T}$ (also see hom_mat2d_scale, hom_mat2d_rotate, and hom_mat2d_translate):

$$\mathtt{HomMat2D} = \begin{bmatrix} \mathbf{R}\cdot\mathbf{S} & \mathbf{T} \\ 0\ \ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \mathbf{T} \\ 0 & 1 & \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & 0 \\ & 0 \\ 0\ \ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{S} & 0 \\ & 0 \\ 0\ \ 0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{T}) \cdot \mathbf{H}(\mathbf{R}) \cdot \mathbf{H}(\mathbf{S})$$

The point correspondences are passed in the tuples (Px, Py) and (Qx,Qy), where corresponding points must be at the same index positions in the tuples. If more than two point correspondences are passed, the transformation is overdetermined. In this case, the returned transformation is the transformation that minimizes the distances between the original points (Px,Py) and the transformed points (Qx,Qy), as described in the following equation (points as homogeneous vectors):

$$\sum_i \left\| \begin{pmatrix} \texttt{Qx}[i] \\ \texttt{Qy}[i] \\ 1 \end{pmatrix} - \texttt{HomMat2D} \cdot \begin{pmatrix} \texttt{Px}[i] \\ \texttt{Py}[i] \\ 1 \end{pmatrix} \right\|^2 = \text{minimum}$$

HomMat2D can be used directly with operators that transform data using affine transformations, e.g., affine_trans_image.

Attention

It should be noted that homogeneous transformation matrices refer to a general right-handed mathematical coordinate system. If a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, the row coordinates of the transformation must be passed in the x coordinates, while the column coordinates must be passed in the y coordinates. Consequently, the order of passing row and column coordinates follows the usual order (Row,Column). This convention is essential to obtain a right-handed coordinate system for the transformation of iconic data, and consequently to ensure in particular that rotations are performed in the correct mathematical direction.

Furthermore, it should be noted that if a homogeneous transformation matrix is used to transform images, regions, XLD contours, or any other data that has been extracted from images, it is assumed that the origin of the coordinate system of the homogeneous transformation matrix lies in the upper left corner of a pixel. The image processing operators that return point coordinates, however, assume a coordinate system in which the origin lies in the center of a pixel. Therefore, to obtain a consistent homogeneous transformation matrix, 0.5 must be added to the point coordinates before computing the transformation.

Parameters

- ▷ **Px** (input_control) ..... point.x-array ⇝ *real*
  X coordinates of the original points.
- ▷ **Py** (input_control) ..... point.y-array ⇝ *real*
  Y coordinates of the original points.
- ▷ **Qx** (input_control) ..... point.x-array ⇝ *real*
  X coordinates of the transformed points.
- ▷ **Qy** (input_control) ..... point.y-array ⇝ *real*
  Y coordinates of the transformed points.
- ▷ **HomMat2D** (output_control) ..... hom_mat2d ⇝ *real*
  Output transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

affine_trans_image, affine_trans_image_size, affine_trans_region, affine_trans_contour_xld, affine_trans_polygon_xld, affine_trans_point_2d

Alternatives

vector_to_hom_mat2d, vector_to_aniso, vector_to_rigid

See also

vector_field_to_hom_mat2d, point_line_to_hom_mat2d

Module

Foundation

27.2 3D Transformations

affine_trans_point_3d (: : HomMat3D, Px, Py, Pz : Qx, Qy, Qz)

Apply an arbitrary affine 3D transformation to points.

affine_trans_point_3d applies an arbitrary affine 3D transformation, i.e., scaling, rotation, and translation, to the input points (Px,Py,Pz) and returns the resulting points in (Qx, Qy,Qz). The affine transformation is described by the homogeneous transformation matrix given in HomMat3D. This corresponds to the following equation (input and output points as homogeneous vectors):

$$\begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = HomMat3D \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix}$$

The transformation matrix can be created using the operators hom_mat3d_identity, hom_mat3d_scale, hom_mat3d_rotate, hom_mat3d_translate, etc., or be the result of pose_to_hom_mat3d.

For example, if HomMat3D corresponds to a rigid transformation, i.e., if it consists of a rotation and a translation, the points are transformed as follows:

$$\begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ 0\,0\,0 & 1 \end{bmatrix} \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R} \cdot \begin{pmatrix} Px \\ Py \\ Pz \end{pmatrix} + \mathbf{T} \\ 1 \end{pmatrix}$$

Parameters

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Input transformation matrix.

▷ **Px** (input_control) . . . . . point3d.x(-array) ⇝ *real* / integer
Input point(s) (x coordinate).
Default: 64
Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Py** (input_control) . . . . . point3d.y(-array) ⇝ *real* / integer
Input point(s) (y coordinate).
Default: 64
Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Pz** (input_control) . . . . . point3d.z(-array) ⇝ *real* / integer
Input point(s) (z coordinate).
Default: 64
Suggested values: Pz ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Qx** (output_control) . . . . . point3d.x(-array) ⇝ *real*
Output point(s) (x coordinate).

▷ **Qy** (output_control) . . . . . point3d.y(-array) ⇝ *real*
Output point(s) (y coordinate).

▷ **Qz** (output_control) . . . . . point3d.z(-array) ⇝ *real*
Output point(s) (z coordinate).

Result

If the parameters are valid, the operator affine_trans_point_3d returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

hom_mat3d_translate, hom_mat3d_translate_local, hom_mat3d_scale, hom_mat3d_scale_local, hom_mat3d_rotate, hom_mat3d_rotate_local

Possible Successors

hom_mat3d_translate, hom_mat3d_translate_local, hom_mat3d_scale, hom_mat3d_scale_local, hom_mat3d_rotate, hom_mat3d_rotate_local

Module

Foundation

deserialize_hom_mat3d (: : SerializedItemHandle : HomMat3D)

Deserialize a serialized homogeneous 3D transformation matrix.

deserialize_hom_mat3d deserializes a homogeneous 3D transformation matrix, that was serialized by serialize_hom_mat3d (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized transformation matrix is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created transformation matrix with the handle HomMat3D.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **HomMat3D** (output_control) ..... hom_mat3d ⇝ *real*
Transformation matrix.

Result

If the parameters are valid, the operator deserialize_hom_mat3d returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_hom_mat3d

Module

Foundation

hom_mat3d_compose (: : HomMat3DLeft,
HomMat3DRight : HomMat3DCompose)

Multiply two homogeneous 3D transformation matrices.

hom_mat3d_compose composes a new 3D transformation matrix by multiplying the two input matrices:

$$\texttt{HomMat3DCompose} = \texttt{HomMat3DLeft} \cdot \texttt{HomMat3DRight}$$

For example, if the two input matrices correspond to rigid transformations, i.e., to transformations consisting of a rotation and a translation, the resulting matrix is calculated as follows:

$$\texttt{HomMat3DCompose} = \begin{bmatrix} \mathbf{R}_l & \mathbf{T}_l \\ 0\,0\,0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}_r & \mathbf{T}_r \\ 0\,0\,0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_l \cdot \mathbf{R}_r & \mathbf{R}_l \cdot \mathbf{T}_r + \mathbf{T}_l \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl]. However, it is also possible to process full 4×4 matrices, which represent a projective 4D transformation.

Parameters

▷ **HomMat3DLeft** (input_control) . . . . . hom_mat3d ⇝ *real*
Left input transformation matrix.

▷ **HomMat3DRight** (input_control) . . . . . hom_mat3d ⇝ *real*
Right input transformation matrix.

▷ **HomMat3DCompose** (output_control) . . . . . hom_mat3d ⇝ *real*
Output transformation matrix.

Result

If the parameters are valid, the operator `hom_mat3d_compose` returns `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`hom_mat3d_compose`, `hom_mat3d_translate`, `hom_mat3d_translate_local`, `hom_mat3d_scale`, `hom_mat3d_scale_local`, `hom_mat3d_rotate`, `hom_mat3d_rotate_local`, `pose_to_hom_mat3d`

Possible Successors

`hom_mat3d_compose`, `hom_mat3d_translate`, `hom_mat3d_translate_local`, `hom_mat3d_scale`, `hom_mat3d_scale_local`, `hom_mat3d_rotate`, `hom_mat3d_rotate_local`

Alternatives

`pose_compose`, `dual_quat_compose`

See also

`affine_trans_point_3d`, `hom_mat3d_identity`, `hom_mat3d_rotate`, `hom_mat3d_translate`, `pose_to_hom_mat3d`, `hom_mat3d_to_pose`

Module

Foundation

hom_mat3d_determinant (: : HomMat3D : Determinant)

Compute the determinant of a homogeneous 3D transformation matrix.

`hom_mat3d_determinant` computes the determinant of the homogeneous 3D transformation matrix given by `HomMat3D` and returns it in `Determinant`.

Parameters

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Input transformation matrix.

▷ **Determinant** (output_control) . . . . . real ⇝ *real*
Determinant of the input matrix.

Result

`hom_mat3d_determinant` always returns `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_translate, hom_mat3d_translate_local, hom_mat3d_scale, hom_mat3d_scale_local, hom_mat3d_rotate, hom_mat3d_rotate_local, vector_to_hom_mat3d

Module

Foundation

hom_mat3d_identity (: : : HomMat3DIdentity)

Generate the homogeneous transformation matrix of the identical 3D transformation.

hom_mat3d_identity generates the homogeneous transformation matrix HomMat3DIdentity describing the identical 3D transformation:

$$\text{HomMat3DIdentity} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. Thus, HomMat3DIdentity is stored as the tuple [1,0,0,0,0,1,0,0,0,0,1,0].

Parameters

▷ **HomMat3DIdentity** (output_control) ..... hom_mat3d ⇝ *real*
Transformation matrix.

Result

hom_mat3d_identity always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

hom_mat3d_translate, hom_mat3d_translate_local, hom_mat3d_scale, hom_mat3d_scale_local, hom_mat3d_rotate, hom_mat3d_rotate_local

Alternatives

pose_to_hom_mat3d

Module

Foundation

hom_mat3d_invert (: : HomMat3D : HomMat3DInvert)

Invert a homogeneous 3D transformation matrix.

hom_mat3d_invert inverts the homogeneous 3D transformation matrix given by HomMat3D. The resulting matrix is returned in HomMat3DInvert.

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl]. However, it is also possible to process full 4×4 matrices, which represent a projective 4D transformation.

Parameters

▷ **HomMat3D** (input_control) ..... hom_mat3d ⇝ *real*
Input transformation matrix.

▷ **HomMat3DInvert** (output_control) ..... hom_mat3d ⇝ *real*
Output transformation matrix.

Result

hom_mat3d_invert returns 2 (H_MSG_TRUE) if the parameters are valid and the input matrix is invertible. Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_translate, hom_mat3d_translate_local, hom_mat3d_scale, hom_mat3d_scale_local, hom_mat3d_rotate, hom_mat3d_rotate_local, pose_to_hom_mat3d

Possible Successors

hom_mat3d_translate, hom_mat3d_translate_local, hom_mat3d_scale, hom_mat3d_scale_local, hom_mat3d_rotate, hom_mat3d_rotate_local, hom_mat3d_to_pose

Alternatives

pose_invert, dual_quat_conjugate

See also

affine_trans_point_3d, hom_mat3d_identity, hom_mat3d_rotate, hom_mat3d_translate, pose_to_hom_mat3d, hom_mat3d_to_pose, hom_mat3d_compose

Module

Foundation

hom_mat3d_rotate (: : HomMat3D, Phi, Axis, Px, Py, Pz : HomMat3DRotate)

Add a rotation to a homogeneous 3D transformation matrix.

hom_mat3d_rotate adds a rotation by the angle Phi around the axis passed in the parameter Axis to the homogeneous 3D transformation matrix HomMat3D and returns the resulting matrix in HomMat3DRotate. The axis can be specified by passing the strings 'x', 'y', or 'z', or by passing a vector [x,y,z] as a tuple.

The rotation is described by a 3×3 rotation matrix $\mathbf{R}$. It is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

Axis = *'x'*:

$$\text{HomMat3DRotate} = \begin{bmatrix} & & & 0 \\ & \mathbf{R}_x & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat3D} \qquad \mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\text{Phi}) & -\sin(\text{Phi}) \\ 0 & \sin(\text{Phi}) & \cos(\text{Phi}) \end{bmatrix}$$

Axis = *'y'*:

$$\texttt{HomMat3DRotate} = \begin{bmatrix} & & & 0 \\ & \mathbf{R}_y & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat3D} \qquad \mathbf{R}_y = \begin{bmatrix} \cos(\texttt{Phi}) & 0 & \sin(\texttt{Phi}) \\ 0 & 1 & 0 \\ -\sin(\texttt{Phi}) & 0 & \cos(\texttt{Phi}) \end{bmatrix}$$

Axis = *'z'*:

$$\texttt{HomMat3DRotate} = \begin{bmatrix} & & & 0 \\ & \mathbf{R}_z & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat3D} \qquad \mathbf{R}_z = \begin{bmatrix} \cos(\texttt{Phi}) & -\sin(\texttt{Phi}) & 0 \\ \sin(\texttt{Phi}) & \cos(\texttt{Phi}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Axis = *[x,y,z]*:

$$\texttt{HomMat3DRotate} = \begin{bmatrix} & & & 0 \\ & \mathbf{R}_a & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat3D} \qquad \mathbf{R}_a = \mathbf{u} \cdot \mathbf{u}^{\mathrm{T}} + cos(\texttt{Phi}) \cdot (\mathbf{I} - \mathbf{u} \cdot \mathbf{u}^{\mathrm{T}}) + sin(\texttt{Phi}) \cdot \mathbf{S}$$

$$\mathbf{u} = \frac{\texttt{Axis}}{\|\texttt{Axis}\|} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{S} = \begin{bmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix}$$

The point (Px,Py,Pz) is the fixed point of the transformation, i.e., this point remains unchanged when transformed using HomMat3DRotate. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the rotation is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations:

$$\texttt{HomMat3DRotate} = \begin{bmatrix} 1 & 0 & 0 & +\texttt{Px} \\ 0 & 1 & 0 & +\texttt{Py} \\ 0 & 0 & 1 & +\texttt{Pz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} & & & 0 \\ & \mathbf{R} & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -\texttt{Px} \\ 0 & 1 & 0 & -\texttt{Py} \\ 0 & 0 & 1 & -\texttt{Pz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \texttt{HomMat3D}$$

To perform the transformation in the local coordinate system, i.e., the one described by HomMat3D, use hom_mat3d_rotate_local.

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl]. However, it is also possible to process full 4×4 matrices, which represent a projective 4D transformation.

Parameters

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Input transformation matrix.

▷ **Phi** (input_control) . . . . . angle.rad ⇝ *real* / integer
Rotation angle.
Default: 0.78
Suggested values: Phi ∈ {0.1, 0.2, 0.3, 0.4, 0.78, 1.57, 3.14}
Value range: 0 ≤ Phi ≤ 6.28318530718

▷ **Axis** (input_control) . . . . . string(-array) ⇝ *string* / real / integer
Axis, to be rotated around.
Default: 'x'
Suggested values: Axis ∈ {'x', 'y', 'z'}

▷ **Px** (input_control) . . . . . point3d.x ⇝ *real* / integer
Fixed point of the transformation (x coordinate).
Default: 0
Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Py** (input_control) . . . . . point3d.y ⇝ *real* / integer
Fixed point of the transformation (y coordinate).
Default: 0
Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Pz** (input_control) . . . . . point3d.z ⇝ *real* / integer
Fixed point of the transformation (z coordinate).
Default: 0
Suggested values: Pz ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **HomMat3DRotate** (output_control) . . . . . hom_mat3d ⇝ *real*
Output transformation matrix.

Result

If the parameters are valid, the operator hom_mat3d_rotate returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_identity, hom_mat3d_translate, hom_mat3d_scale, hom_mat3d_rotate

Possible Successors

hom_mat3d_translate, hom_mat3d_scale, hom_mat3d_rotate

See also

hom_mat3d_invert, hom_mat3d_identity, hom_mat3d_rotate_local, pose_to_hom_mat3d, hom_mat3d_to_pose, hom_mat3d_compose

Module

Foundation

hom_mat3d_rotate_local (: : HomMat3D, Phi, Axis : HomMat3DRotate)

Add a rotation to a homogeneous 3D transformation matrix.

hom_mat3d_rotate_local adds a rotation by the angle Phi around the axis passed in the parameter Axis to the homogeneous 3D transformation matrix HomMat3D and returns the resulting matrix in HomMat3DRotate. The axis can be specified by passing the strings 'x', 'y', or 'z', or by passing a vector [x,y,z] as a tuple.

The rotation is described by a 3×3 rotation matrix $\mathbf{R}$. In contrast to hom_mat3d_rotate, it is performed relative to the local coordinate system, i.e., the coordinate system described by HomMat3D; this corresponds to the following chain of transformation matrices:

Axis = *'x'*:

$$\text{HomMat3DRotate} = \text{HomMat3D} \cdot \begin{bmatrix} & & & 0 \\ & \mathbf{R}_x & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\text{Phi}) & -\sin(\text{Phi}) \\ 0 & \sin(\text{Phi}) & \cos(\text{Phi}) \end{bmatrix}$$

Axis = 'y':

$$\texttt{HomMat3DRotate} = \texttt{HomMat3D} \cdot \begin{bmatrix} & & & 0 \\ & \mathbf{R}_y & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R}_y = \begin{bmatrix} \cos(\texttt{Phi}) & 0 & \sin(\texttt{Phi}) \\ 0 & 1 & 0 \\ -\sin(\texttt{Phi}) & 0 & \cos(\texttt{Phi}) \end{bmatrix}$$

Axis = 'z':

$$\texttt{HomMat3DRotate} = \texttt{HomMat3D} \cdot \begin{bmatrix} & & & 0 \\ & \mathbf{R}_z & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R}_z = \begin{bmatrix} \cos(\texttt{Phi}) & -\sin(\texttt{Phi}) & 0 \\ \sin(\texttt{Phi}) & \cos(\texttt{Phi}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

@

Axis = [x,y,z]:

$$\texttt{HomMat3DRotate} = \texttt{HomMat3D} \cdot \begin{bmatrix} & & & 0 \\ & \mathbf{R}_a & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{R}_a = \mathbf{u} \cdot \mathbf{u}^{\mathrm{T}} + cos(\texttt{Phi}) \cdot (\mathbf{I} - \mathbf{u} \cdot \mathbf{u}^{\mathrm{T}}) + sin(\texttt{Phi}) \cdot \mathbf{S}$$

$$\mathbf{u} = \frac{\texttt{Axis}}{\|\texttt{Axis}\|} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \qquad \mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{S} = \begin{bmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix}$$

The fixed point of the transformation is the origin of the local coordinate system, i.e., this point remains unchanged when transformed using HomMat3DRotate.

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl]. However, it is also possible to process full 4×4 matrices, which represent a projective 4D transformation.

Parameters

- ▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
  Input transformation matrix.
- ▷ **Phi** (input_control) . . . . . angle.rad ⇝ *real* / integer
  Rotation angle.
  Default: 0.78
  Suggested values: Phi ∈ {0.1, 0.2, 0.3, 0.4, 0.78, 1.57, 3.14}
  Value range: 0 ≤ Phi ≤ 6.28318530718
- ▷ **Axis** (input_control) . . . . . string(-array) ⇝ *string* / real / integer
  Axis, to be rotated around.
  Default: 'x'
  Suggested values: Axis ∈ {'x', 'y', 'z'}
- ▷ **HomMat3DRotate** (output_control) . . . . . hom_mat3d ⇝ *real*
  Output transformation matrix.

Result

If the parameters are valid, the operator hom_mat3d_rotate_local returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_identity, hom_mat3d_translate_local, hom_mat3d_scale_local, hom_mat3d_rotate_local

Possible Successors

hom_mat3d_translate_local, hom_mat3d_scale_local, hom_mat3d_rotate_local

See also

hom_mat3d_invert, hom_mat3d_identity, hom_mat3d_rotate, pose_to_hom_mat3d, hom_mat3d_to_pose, hom_mat3d_compose

Module

Foundation

hom_mat3d_scale (: : HomMat3D, Sx, Sy, Sz, Px, Py, Pz : HomMat3DScale)

Add a scaling to a homogeneous 3D transformation matrix.

hom_mat3d_scale adds a scaling by the scale factors Sx, Sy, and Sz to the homogeneous 3D transformation matrix HomMat3D and returns the resulting matrix in HomMat3DScale. The scaling is described by a 3×3 scaling matrix $\mathbf{S}$. It is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\mathtt{HomMat3DScale} = \begin{bmatrix} & & 0 \\ & \mathbf{S} & 0 \\ & & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \mathtt{HomMat3D} \qquad \mathbf{S} = \begin{bmatrix} \mathtt{Sx} & 0 & 0 \\ 0 & \mathtt{Sy} & 0 \\ 0 & 0 & \mathtt{Sz} \end{bmatrix}$$

The point (Px,Py,Pz) is the fixed point of the transformation, i.e., this point remains unchanged when transformed using HomMat3DScale. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the scaling is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations:

$$\mathtt{HomMat3DScale} = \begin{bmatrix} 1 & 0 & 0 & +\mathtt{Px} \\ 0 & 1 & 0 & +\mathtt{Py} \\ 0 & 0 & 1 & +\mathtt{Pz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} & & 0 \\ & \mathbf{S} & 0 \\ & & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -\mathtt{Px} \\ 0 & 1 & 0 & -\mathtt{Py} \\ 0 & 0 & 1 & -\mathtt{Pz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \mathtt{HomMat3D}$$

To perform the transformation in the local coordinate system, i.e., the one described by HomMat3D, use hom_mat3d_scale_local.

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl]. However, it is also possible to process full 4×4 matrices, which represent a projective 4D transformation.

Parameters

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Input transformation matrix.

▷ **Sx** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the x-axis.
Default: 2
Suggested values: Sx ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 112}
Restriction: Sx != 0

▷ **Sy** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the y-axis.
Default: 2
Suggested values: Sy ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 112}
Restriction: Sy != 0

▷ **Sz** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the z-axis.
Default: 2
Suggested values: Sz ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 112}
Restriction: Sz != 0

▷ **Px** (input_control) . . . . . point3d.x ⇝ *real* / integer
Fixed point of the transformation (x coordinate).
Default: 0
Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Py** (input_control) . . . . . point3d.y ⇝ *real* / integer
Fixed point of the transformation (y coordinate).
Default: 0
Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Pz** (input_control) . . . . . point3d.z ⇝ *real* / integer
Fixed point of the transformation (z coordinate).
Default: 0
Suggested values: Pz ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **HomMat3DScale** (output_control) . . . . . hom_mat3d ⇝ *real*
Output transformation matrix.

Result

hom_mat3d_scale returns 2 (H_MSG_TRUE) if all three scale factors are not 0. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_identity, hom_mat3d_translate, hom_mat3d_scale, hom_mat3d_rotate

Possible Successors

hom_mat3d_translate, hom_mat3d_scale, hom_mat3d_rotate

See also

hom_mat3d_invert, hom_mat3d_identity, hom_mat3d_scale_local, pose_to_hom_mat3d, hom_mat3d_to_pose, hom_mat3d_compose

Module

Foundation

hom_mat3d_scale_local (: : HomMat3D, Sx, Sy, Sz : HomMat3DScale)

Add a scaling to a homogeneous 3D transformation matrix.

hom_mat3d_scale_local adds a scaling by the scale factors Sx, Sy, and Sz to the homogeneous 3D transformation matrix HomMat3D and returns the resulting matrix in HomMat3DScale. The scaling is described by a 3×3 scaling matrix $\mathbf{S}$. In contrast to hom_mat3d_scale, it is performed relative to the local coordinate system, i.e., the coordinate system described by HomMat3D; this corresponds to the following chain of transformation matrices:

$$\texttt{HomMat3DScale} = \texttt{HomMat3D} \cdot \begin{bmatrix} & & & 0 \\ & \mathbf{S} & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{S} = \begin{bmatrix} \texttt{Sx} & 0 & 0 \\ 0 & \texttt{Sy} & 0 \\ 0 & 0 & \texttt{Sz} \end{bmatrix}$$

The fixed point of the transformation is the origin of the local coordinate system, i.e., this point remains unchanged when transformed using HomMat3DScale.

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl]. However, it is also possible to process full 4×4 matrices, which represent a projective 4D transformation.

Parameters

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Input transformation matrix.

▷ **Sx** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the x-axis.
Default: 2
Suggested values: Sx ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 112}
Restriction: Sx != 0

▷ **Sy** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the y-axis.
Default: 2
Suggested values: Sy ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 112}
Restriction: Sy != 0

▷ **Sz** (input_control) . . . . . number ⇝ *real* / integer
Scale factor along the z-axis.
Default: 2
Suggested values: Sz ∈ {0.125, 0.25, 0.5, 1, 2, 4, 8, 112}
Restriction: Sz != 0

▷ **HomMat3DScale** (output_control) . . . . . hom_mat3d ⇝ *real*
Output transformation matrix.

Result

hom_mat3d_scale_local returns 2 (H_MSG_TRUE) if all three scale factors are not 0. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_identity, hom_mat3d_translate_local, hom_mat3d_scale_local, hom_mat3d_rotate_local

Possible Successors

hom_mat3d_translate_local, hom_mat3d_scale_local, hom_mat3d_rotate_local

See also

hom_mat3d_invert, hom_mat3d_identity, hom_mat3d_scale, pose_to_hom_mat3d, hom_mat3d_to_pose, hom_mat3d_compose

Module

Foundation

hom_mat3d_to_pose (: : HomMat3D : Pose)

Convert a homogeneous transformation matrix into a 3D pose.

hom_mat3d_to_pose converts a homogeneous transformation matrix into the corresponding 3D pose with type code 0. For details about 3D poses and the corresponding transformation matrices please refer to create_pose.

A typical application of hom_mat3d_to_pose is that a 3D pose was converted into a homogeneous transformation matrix to further transform it, e.g., with hom_mat3d_rotate or hom_mat3d_translate, and now must be converted back into a pose to use it as input for operators like image_points_to_world_plane.

Attention

hom_mat3d_to_pose only supports rigid transformations in HomMat3D. For non-rigid transformations, the operator attempts to return a pose that is close to HomMat3D. If HomMat3D deviates significantly from a rigid transformation, the generated pose may also deviate significantly from HomMat3D.

Parameters

▷ **HomMat3D** (input_control) ..... hom_mat3d ⇝ *real*
Homogeneous transformation matrix.

▷ **Pose** (output_control) ..... pose ⇝ *real* / integer
Equivalent 3D pose.
Number of elements: 7

Example

```
* Calibrate camera.
calibrate_cameras (CalibDataID,Error)
* Get reference pose (pose 2 of calibration object 0).
get_calib_data (CalibDataID, 'calib_obj_pose',\
                [0,2], 'pose', ObjInCameraPose)
*  Convert pose to homogeneous transformation matrix.
pose_to_hom_mat3d(ObjInCameraPose, cam_H_cal)
*  Rotate it 90 degrees around its y-axis to obtain a world coordinate system
*  whose y- and z-axis lie in the plane of the calibration plate while the
*  x-axis point 'upwards': cam_H_w = cam_H_cal * RotY(90).
hom_mat3d_identity(HomMat3DIdent)
hom_mat3d_rotate(HomMat3DIdent, rad(90), 'y', 0, 0, 0, \
                 HomMat3DRotateY)
hom_mat3d_compose(cam_H_cal, HomMat3DRotateY, cam_H_w)
* Convert transformed matrix back to pose.
hom_mat3d_to_pose (cam_H_w, Pose)
```

Result

hom_mat3d_to_pose returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

hom_mat3d_rotate, hom_mat3d_translate, hom_mat3d_invert

Possible Successors

camera_calibration, write_pose, disp_caltab, sim_caltab

See also

create_pose, camera_calibration, disp_caltab, sim_caltab, write_pose, read_pose, pose_to_hom_mat3d, project_3d_point, get_line_of_sight, hom_mat3d_rotate, hom_mat3d_translate, hom_mat3d_invert, affine_trans_point_3d

Module

Foundation

hom_mat3d_translate (: : HomMat3D, Tx, Ty, Tz : HomMat3DTranslate)

Add a translation to a homogeneous 3D transformation matrix.

hom_mat3d_translate adds a translation by the vector $\mathbf{T} = (\mathtt{Tx},\mathtt{Ty},\mathtt{Tz})$ to the homogeneous 3D transformation matrix HomMat3D and returns the resulting matrix in HomMat3DTranslate. The translation is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\mathtt{HomMat3DTranslate} = \begin{bmatrix} 1 & 0 & 0 & \\ 0 & 1 & 0 & \mathbf{T} \\ 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \mathtt{HomMat3D} \qquad \mathbf{T} = \begin{pmatrix} \mathtt{Tx} \\ \mathtt{Ty} \\ \mathtt{Tz} \end{pmatrix}$$

To perform the transformation in the local coordinate system, i.e., the one described by HomMat3D, use hom_mat3d_translate_local.

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl]. However, it is also possible to process full 4×4 matrices, which represent a projective 4D transformation.

Parameters

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Input transformation matrix.

▷ **Tx** (input_control) . . . . . point3d.x ⇝ *real* / integer
Translation along the x-axis.
Default: 64
Suggested values: Tx ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Ty** (input_control) . . . . . point3d.y ⇝ *real* / integer
Translation along the y-axis.
Default: 64
Suggested values: Ty ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Tz** (input_control) . . . . . point3d.z ⇝ *real* / integer
Translation along the z-axis.
Default: 64
Suggested values: Tz ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **HomMat3DTranslate** (output_control) . . . . . hom_mat3d ⇝ *real*
Output transformation matrix.

Result

If the parameters are valid, the operator hom_mat3d_translate returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_identity, hom_mat3d_translate, hom_mat3d_scale, hom_mat3d_rotate

Possible Successors

hom_mat3d_translate, hom_mat3d_scale, hom_mat3d_rotate

See also

hom_mat3d_invert, hom_mat3d_identity, hom_mat3d_translate_local, pose_to_hom_mat3d, hom_mat3d_to_pose, hom_mat3d_compose

Module

Foundation

hom_mat3d_translate_local (: : HomMat3D, Tx, Ty, Tz : HomMat3DTranslate)

Add a translation to a homogeneous 3D transformation matrix.

hom_mat3d_translate_local adds a translation by the vector $\mathbf{T}$ = (Tx,Ty,Tz) to the homogeneous 3D transformation matrix HomMat3D and returns the resulting matrix in HomMat3DTranslate. In contrast to hom_mat3d_translate, the translation is performed relative to the local coordinate system, i.e., the coordinate system described by HomMat3D; this corresponds to the following chain of transformation matrices:

$$\mathtt{HomMat3DTranslate} = \mathtt{HomMat3D} \cdot \begin{bmatrix} 1 & 0 & 0 & \\ 0 & 1 & 0 & \mathbf{T} \\ 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \mathbf{T} = \begin{pmatrix} \mathtt{Tx} \\ \mathtt{Ty} \\ \mathtt{Tz} \end{pmatrix}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is usually not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl]. However, it is also possible to process full 4×4 matrices, which represent a projective 4D transformation.

Parameters

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Input transformation matrix.

▷ **Tx** (input_control) . . . . . point3d.x ⇝ *real* / integer
Translation along the x-axis.
Default: 64
Suggested values: Tx ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Ty** (input_control) . . . . . point3d.y ⇝ *real* / integer
Translation along the y-axis.
Default: 64
Suggested values: Ty ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Tz** (input_control) . . . . . point3d.z ⇝ *real* / integer
Translation along the z-axis.
Default: 64
Suggested values: Tz ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **HomMat3DTranslate** (output_control) . . . . . hom_mat3d ⇝ *real*
Output transformation matrix.

Result

If the parameters are valid, the operator hom_mat3d_translate_local returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_identity, hom_mat3d_translate_local, hom_mat3d_scale_local, hom_mat3d_rotate_local

Possible Successors

hom_mat3d_translate_local, hom_mat3d_scale_local, hom_mat3d_rotate_local

See also

hom_mat3d_invert, hom_mat3d_identity, hom_mat3d_translate, pose_to_hom_mat3d, hom_mat3d_to_pose, hom_mat3d_compose

Module

Foundation

hom_mat3d_transpose (: : HomMat3D : HomMat3DTranspose)

Transpose a homogeneous 3D transformation matrix.

hom_mat3d_transpose transposes the homogeneous 3D transformation matrix given by HomMat3D. The result matrix HomMat3DTranspose is always a 4×4 matrix, even if the input matrix is represented by a 3×4 matrix.

Parameters

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Input transformation matrix.

▷ **HomMat3DTranspose** (output_control) . . . . . hom_mat3d ⇝ *real*
Output transformation matrix.

Result

hom_mat3d_transpose always returns 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

hom_mat3d_translate, hom_mat3d_translate_local, hom_mat3d_scale,

hom_mat3d_scale_local, hom_mat3d_rotate, hom_mat3d_rotate_local, vector_to_hom_mat3d

Possible Successors

hom_mat3d_compose, hom_mat3d_invert

Module

Foundation

point_pluecker_line_to_hom_mat3d (: : TransformationType, PointX, PointY, PointZ, LineDirectionX, LineDirectionY, LineDirectionZ, LineMomentX, LineMomentY, LineMomentZ : HomMat3D)

Approximate a 3D affine transformation from 3D point-to-line correspondences.

point_pluecker_line_to_hom_mat3d approximates a 3D affine transformation from 3D point-to-line correspondences and returns it as the homogeneous transformation matrix HomMat3D. The input tuples must be of same length and determine at least 3 point-to-line correspondences. The corresponding points and lines must be at the same index positions in the tuples. The points are given by (PointX, PointY, PointZ). The lines are given by their Plücker coordinates $\mathbf{L}$ (LineDirectionX, LineDirectionY, LineDirectionZ) and $\mathbf{M}$ (LineMomentX, LineMomentY, LineMomentZ) For the definition of Plücker coordinates, see "Solution Guide III-C - 3D Vision".

The type of the 3D transformation to compute is specified with TransformationType. For TransformationType = *'rigid'*, a rigid 3D transformation (a rotation and a translation) is computed. This is the only transformation type that is supported at the moment.

Note that for the minimum number of 3 correspondences, up to 8 valid solutions are possible. point_pluecker_line_to_hom_mat3d returns an arbitrarily chosen valid solution in this case. If a unique solution is desired, at least 4 correspondences should be specified. Furthermore, note that no well-defined solution exists if all lines are parallel. In this case, point_pluecker_line_to_hom_mat3d returns an error.

The returned transformation minimizes the sum of the squared orthogonal distances of the points transformed with HomMat3D from their corresponding lines (see distance_point_pluecker_line).

HomMat3D can be used directly with operators that transform 3D data using affine transformations, e.g., affine_trans_point_3d. To transform lines given in Plücker coordinates, the operator dual_quat_trans_line_3d can be used. Here, HomMat3D must be converted into a pose using hom_mat3d_to_pose, the pose must be inverted using pose_invert and then converted into a dual quaternion using pose_to_dual_quat.

Parameters

▷ **TransformationType** (input_control) ..... string ⇝ *string*
Type of the transformation to compute.
Default: 'rigid'
List of values: TransformationType ∈ {'rigid'}

▷ **PointX** (input_control) ..... point3d.x-array ⇝ *real*
X coordinate of the original points.

▷ **PointY** (input_control) ..... point3d.y-array ⇝ *real*
Y coordinate of the original points.

▷ **PointZ** (input_control) ..... point3d.z-array ⇝ *real*
Z coordinate of the original points.

▷ **LineDirectionX** (input_control) ..... point3d.x-array ⇝ *real* / integer
X component of the direction vector of the corresponding line.

▷ **LineDirectionY** (input_control) ..... point3d.y-array ⇝ *real* / integer
Y component of the direction vector of the corresponding line.

▷ **LineDirectionZ** (input_control) ..... point3d.z-array ⇝ *real* / integer
Z component of the direction vector of the corresponding line.

▷ **LineMomentX** (input_control) ..... point3d.x-array ⇝ *real* / integer
X component of the moment vector of the corresponding line.

▷ **LineMomentY** (input_control) ..... point3d.y-array ⇝ *real* / integer
Y component of the moment vector of the corresponding line.

▷ **LineMomentZ** (input_control) . . . . . point3d.z-array ⇝ *real* / integer
Z component of the moment vector of the corresponding line.

▷ **HomMat3D** (output_control) . . . . . hom_mat3d ⇝ *real*
Output transformation matrix.

Example

```
* In this example, we assume that correspondences between points
* (PX, PY, PZ) and Plücker lines (LX, LY, LZ) - (MX, MY, MZ) have
* already been computed.
point_pluecker_line_to_hom_mat3d ('rigid', PX, PY, PZ, LX, LY, LZ, \
                                  MX, MY, MZ, HomMat3D)
hom_mat3d_to_pose (HomMat3D, Pose)
* Compute the residual errors in 3D space, i.e., the distance of the
* points (PX, PY, PZ) transformed by HomMat3D from the lines.
affine_trans_point_3d (HomMat3D, PX, PY, PZ, TPX, TPY, TPZ)
distance_point_pluecker_line (TPX, TPY, TPZ, LX, LY, LZ, \
                              MX, MY, MZ, D)
* The same distance can also be computed by transforming the lines
* instead of the points.  Here, the inverse pose must be used.
pose_invert (Pose, PoseInv)
pose_to_dual_quat (PoseInv, DualQuat)
dual_quat_trans_line_3d (DualQuat, 'moment', LX, LY, LZ, MX, MY, MZ, \
                         TLX, TLY, TLZ, TMX, TMY, TMZ)
distance_point_pluecker_line (PX, PY, PZ, TLX, TLY, TLZ, \
                              TMX, TMY, TMZ, D)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_line_of_sight, points_to_pluecker_line, point_direction_to_pluecker_line

Possible Successors

hom_mat3d_to_pose, affine_trans_point_3d, pose_invert, pose_to_dual_quat, dual_quat_trans_line_3d, distance_point_pluecker_line

See also

vector_to_hom_mat3d

Module

Foundation

pose_to_hom_mat3d (: : Pose : HomMat3D)

Convert a 3D pose into a homogeneous transformation matrix.

pose_to_hom_mat3d converts a 3D pose Pose, e.g., the external camera parameters, into the equivalent homogeneous transformation matrix HomMat3D. For details about 3D poses and the corresponding transformation matrices please refer to create_pose.

A typical application of pose_to_hom_mat3d is that you want to further transform the pose, e.g., rotate or translate it using hom_mat3d_rotate or hom_mat3d_translate. In case of the external camera parameters, this can be necessary if the calibration plate cannot be placed such that its coordinate system coincides with the desired world coordinate system.

Parameters

▷ **Pose** (input_control) . . . . . pose ⇝ *real* / integer
3D pose.
Number of elements: 7

▷ **HomMat3D** (output_control) . . . . . hom_mat3d ⇝ *real*
Equivalent homogeneous transformation matrix.

Example

```
* Calibrate camera.
calibrate_cameras (CalibDataID,Error)
* Get reference pose (pose 2 of calibration object 0).
get_calib_data (CalibDataID, 'calib_obj_pose',\
                [0,2], 'pose', ObjInCameraPose)
*  Convert pose to homogeneous transformation matrix.
pose_to_hom_mat3d(ObjInCameraPose, cam_H_cal)
*  Rotate it 90 degrees around its y-axis to obtain a world coordinate system
*  whose y- and z-axis lie in the plane of the calibration plate while the
*  x-axis point 'upwards': cam_H_w = cam_H_cal * RotY(90).
hom_mat3d_identity(HomMat3DIdent)
hom_mat3d_rotate(HomMat3DIdent, rad(90), 'y', 0, 0, 0, \
                 HomMat3DRotateY)
hom_mat3d_compose(cam_H_cal, HomMat3DRotateY, cam_H_w)
* Convert transformed matrix back to pose.
hom_mat3d_to_pose (cam_H_w, Pose)
```

Result

`pose_to_hom_mat3d` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`camera_calibration`, `read_pose`

Possible Successors

`affine_trans_point_3d`, `hom_mat3d_invert`, `hom_mat3d_translate`, `hom_mat3d_rotate`, `hom_mat3d_to_pose`

Alternatives

`pose_to_dual_quat`

See also

`create_pose`, `camera_calibration`, `write_pose`, `read_pose`, `hom_mat3d_to_pose`, `project_3d_point`, `get_line_of_sight`, `hom_mat3d_rotate`, `hom_mat3d_translate`, `hom_mat3d_invert`, `affine_trans_point_3d`

Module

Foundation

projective_trans_hom_point_3d (: : HomMat3D, Px, Py, Pz, Pw : Qx, Qy, Qz, Qw)

Project a homogeneous 3D point using a projective transformation matrix.

`projective_trans_hom_point_3d` applies the homogeneous projective transformation matrix `HomMat3D` to all homogeneous input points (`Px`,`Py`,`Pz`,`Pw`) and returns an array of homogeneous output

points (Qx,Qy,Qz,Qw). The transformation is described by the homogeneous transformation matrix given in HomMat3D. This corresponds to the following equation (input and output points as homogeneous vectors):

$$\begin{pmatrix} Qx \\ Qy \\ Qz \\ Qw \end{pmatrix} = HomMat3D \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ Pw \end{pmatrix}$$

To transform the homogeneous coordinates to Euclidean coordinates, they must be divided by Qw:

$$\begin{pmatrix} Ex \\ Ey \\ Ez \end{pmatrix} = \begin{pmatrix} \frac{Qx}{Qw} \\ \frac{Qy}{Qw} \\ \frac{Qz}{Qw} \end{pmatrix}$$

This can be achieved directly by calling projective_trans_point_3d. Thus, projective_trans_hom_point_3d is primarily useful for transforming points or point sets for which the resulting points might lie in the plane at infinity, i.e., points that potentially have Qw = 0, for which the above division cannot be performed.

Parameters

- ▷ **HomMat3D** (input_control) . . . hom_mat3d ⇝ *real*
  Homogeneous projective transformation matrix.
- ▷ **Px** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (x coordinate).
- ▷ **Py** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (y coordinate).
- ▷ **Pz** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (z coordinate).
- ▷ **Pw** (input_control) . . . number(-array) ⇝ *real* / integer
  Input point (w coordinate).
- ▷ **Qx** (output_control) . . . number(-array) ⇝ *real*
  Output point (x coordinate).
- ▷ **Qy** (output_control) . . . number(-array) ⇝ *real*
  Output point (y coordinate).
- ▷ **Qz** (output_control) . . . number(-array) ⇝ *real*
  Output point (z coordinate).
- ▷ **Qw** (output_control) . . . number(-array) ⇝ *real*
  Output point (w coordinate).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

vector_to_hom_mat3d

Alternatives

projective_trans_point_3d

Module

Foundation

projective_trans_point_3d (: : HomMat3D, Px, Py, Pz : Qx, Qy, Qz)

Project a 3D point using a projective transformation matrix.

projective_trans_point_3d applies the homogeneous projective transformation matrix HomMat3D to all input points (Px,Py,Pz) and returns an array of output points (Qx,Qy,Qz). The transformation is described by the homogeneous transformation matrix given in HomMat3D. This corresponds to the following equations (input and output points as homogeneous vectors):

$$\begin{pmatrix} Tx \\ Ty \\ Tz \\ Tw \end{pmatrix} = \texttt{HomMat3D} \cdot \begin{pmatrix} \texttt{Px} \\ \texttt{Py} \\ \texttt{Pz} \\ 1 \end{pmatrix}$$

projective_trans_point_3d then transforms the homogeneous coordinates to Euclidean coordinates by dividing them by Tw:

$$\begin{pmatrix} \texttt{Qx} \\ \texttt{Qy} \\ \texttt{Qz} \end{pmatrix} = \begin{pmatrix} \frac{Tx}{Tw} \\ \frac{Ty}{Tw} \\ \frac{Tz}{Tw} \end{pmatrix}$$

If a point in the plane at infinity ($Tw = 0$) is created by the transformation, an error is returned. If this is undesired, projective_trans_hom_point_3d can be used.

Parameters

- ▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
  Homogeneous projective transformation matrix.
- ▷ **Px** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input point (x coordinate).
- ▷ **Py** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input point (y coordinate).
- ▷ **Pz** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input point (z coordinate).
- ▷ **Qx** (output_control) . . . . . number(-array) ⇝ *real*
  Output point (x coordinate).
- ▷ **Qy** (output_control) . . . . . number(-array) ⇝ *real*
  Output point (y coordinate).
- ▷ **Qz** (output_control) . . . . . number(-array) ⇝ *real*
  Output point (z coordinate).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

vector_to_hom_mat3d

Alternatives

projective_trans_hom_point_3d

Module

Foundation

serialize_hom_mat3d (: : HomMat3D : SerializedItemHandle)

Serialize a homogeneous 3D transformation matrix.

serialize_hom_mat3d serializes the data of a homogeneous 3D transformation matrix (see fwrite_serialized_item for an introduction of the basic principle of serialization). The transformation matrix is defined by the handle HomMat3D. The serialized transformation matrix is returned by the handle SerializedItemHandle and can be deserialized by deserialize_hom_mat3d.

Parameters

▷ **HomMat3D** (input_control) . . . . . hom_mat3d ⇝ *real*
Transformation matrix.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator `serialize_hom_mat3d` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`fwrite_serialized_item`, `send_serialized_item`, `deserialize_hom_mat3d`

Module

Foundation

vector_to_hom_mat3d (: : TransformationType, Px, Py, Pz, Qx, Qy, Qz : HomMat3D)

Approximate a 3D transformation from point correspondences.

`vector_to_hom_mat3d` approximates an affine or projective 3D transformation from point correspondences and returns it as the homogeneous transformation matrix `HomMat3D`.

The type of the 3D transformation to compute is specified with `TransformationType`. For `TransformationType` = *'rigid'*, a rigid 3D transformation (a rotation and a translation), for `TransformationType` = *'similarity'*, a 3D similarity transformation (a uniform scaling, a rotation, and a translation), for `TransformationType` = *'affine'* a general affine 3D transformation, and for `TransformationType` = *'projective'* a projective 3D transformation is computed.

The minimum required number of point correspondences is 3 for `TransformationType` = *'rigid'*, 3 for `TransformationType` = *'similarity'*, 4 for `TransformationType` = *'affine'*, and 5 for `TransformationType` = *'projective'*.

The point correspondences are passed in the tuples (`Px`,`Py`,`Pz`) and (`Qx`,`Qy`,`Qz`), where corresponding points must be at the same index positions in the tuples. If more than the minimum number of point correspondences are passed, the transformation is overdetermined. In this case, the returned transformation is the transformation that minimizes the distances between the transformed input points (`Px`,`Py`,`Pz`) and the points (`Qx`,`Qy`,`Qz`), as described in the following equation (points as homogeneous vectors):

$$\sum_i \left\| \begin{pmatrix} \mathtt{Qx}[i] \\ \mathtt{Qy}[i] \\ \mathtt{Qz}[i] \\ 1 \end{pmatrix} - \mathtt{HomMat3D} \cdot \begin{pmatrix} \mathtt{Px}[i] \\ \mathtt{Py}[i] \\ \mathtt{Pz}[i] \\ 1 \end{pmatrix} \right\|^2 = \text{minimum}$$

`HomMat3D` can be used directly with operators that transform 3D data using affine transformations, e.g., `affine_trans_point_3d`.

Parameters

▷ **TransformationType** (input_control) . . . . . string ⇝ *string*
Type of the transformation to compute.
Default: 'rigid'
List of values: `TransformationType` ∈ {'rigid', 'similarity', 'affine', 'projective'}

▷ **Px** (input_control) . . . . . point3d.x-array ⇝ *real*
X coordinates of the original points.

▷ **Py** (input_control) . . . . . point3d.y-array ⇝ *real*
Y coordinates of the original points.
▷ **Pz** (input_control) . . . . . point3d.z-array ⇝ *real*
Z coordinates of the original points.
▷ **Qx** (input_control) . . . . . point3d.x-array ⇝ *real*
X coordinates of the transformed points.
▷ **Qy** (input_control) . . . . . point3d.x-array ⇝ *real*
Y coordinates of the transformed points.
▷ **Qz** (input_control) . . . . . point3d.z-array ⇝ *real*
Z coordinates of the transformed points.
▷ **HomMat3D** (output_control) . . . . . hom_mat3d ⇝ *real*
Output transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

hom_mat3d_to_pose, affine_trans_point_3d

See also

point_pluecker_line_to_hom_mat3d

Module

Foundation

27.3 Dual Quaternions

This chapter contains operators for handling dual quaternions.

Introduction to Dual Quaternions

A dual quaternion $\hat{q} = q_r + \varepsilon q_d$ consists of the two quaternions q_r and q_d, where q_r is the real part, q_d is the dual part, and ε is the dual unit number ($\varepsilon^2 = 0$). Each quaternion $q = w + ix + jy + kz$ consists of the scalar part w and the vector part $\mathbf{v} = (x, y, z)$, where $(1, i, j, k)$ are the basis elements of the quaternion vector space.

For information how dual quaternions can be used for the description of rigid 3D transformations and their relation to Plücker coordinates, see "Solution Guide III-C - 3D Vision".

Representing Dual Quaternions in HALCON

In HALCON, a dual quaternion is represented by a tuple with eight values $[w_r, x_r, y_r, z_r, w_d, x_d, y_d, z_d]$, where w_r and $\mathbf{v}_r = (x_r, y_r, z_r)$ are the scalar and the vector part of the real part and w_d and $\mathbf{v}_d = (x_d, y_d, z_d)$ are the scalar and the vector part of the dual part.

deserialize_dual_quat (: : SerializedItemHandle : DualQuaternion)

Deserialize a serialized dual quaternion.

deserialize_dual_quat deserializes a dual quaternion that was serialized by serialize_dual_quat (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized dual quaternion is defined by the handle SerializedItemHandle. The deserialized values are stored in DualQuaternion.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **DualQuaternion** (output_control) ..... dual_quaternion ⇝ *real* / integer
Dual quaternion.

Result

If the parameters are valid, the operator `deserialize_dual_quat` returns the value `2` (`H_MSG_TRUE`). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fread_serialized_item`, `receive_serialized_item`, `serialize_quat`

Possible Successors

`dual_quat_to_hom_mat3d`, `dual_quat_to_pose`, `dual_quat_compose`

Module

Foundation

dual_quat_compose (: : DualQuaternionLeft,
DualQuaternionRight : DualQuaternionComposed)

Multiply two dual quaternions.

The operator `dual_quat_compose` multiplies the two dual quaternions `DualQuaternionLeft` and `DualQuaternionRight` and returns the product in `DualQuaternionComposed`.

For a brief introduction to dual quaternions, the used notation, and the relationship between dual quaternions and screws, see "`Solution Guide III-C - 3D Vision`".

The multiplication of the dual quaternions $\hat{p}$ and $\hat{q}$ is $\hat{p}\hat{q} = p_r q_r + \varepsilon(p_r q_d + p_d q_r)$.

For the multiplication of quaternions see `quat_compose`.

Note that the multiplication of two dual quaternions is not commutative, i.e., $\hat{p}\hat{q} \neq \hat{q}\hat{p}$.

If `DualQuaternionLeft` and `DualQuaternionRight` are unit dual quaternions and, hence, represent 3D rigid transformations, their multiplication corresponds to the multiplication of their corresponding homogeneous transformation matrices. Consequently, `dual_quat_compose` can be used to concatenate two 3D rigid transformations analogously to `pose_compose` or `hom_mat3d_compose`.

Parameters

▷ **DualQuaternionLeft** (input_control) ..... dual_quaternion(-array) ⇝ *real*
Left dual quaternion.

▷ **DualQuaternionRight** (input_control) ..... dual_quaternion(-array) ⇝ *real*
Right dual quaternion.

▷ **DualQuaternionComposed** (output_control) ..... dual_quaternion(-array) ⇝ *real*
Product of the dual quaternions.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

screw_to_dual_quat

Possible Successors

dual_quat_to_hom_mat3d, dual_quat_to_pose, dual_quat_to_screw

Alternatives

pose_compose, hom_mat3d_compose

See also

dual_quat_interpolate, pose_to_dual_quat, dual_quat_normalize, dual_quat_conjugate, serialize_dual_quat, deserialize_dual_quat, dual_quat_trans_line_3d, dual_quat_trans_point_3d, quat_compose

Module

Foundation

dual_quat_conjugate (
: : DualQuaternion : DualQuaternionConjugate)

Conjugate a dual quaternion.

The operator dual_quat_conjugate computes the conjugation DualQuaternionConjugate of the input dual quaternion DualQuaternion.

For a brief introduction to dual quaternions, the used notation, and the relationship between dual quaternions and screws, see "Solution Guide III-C - 3D Vision".

The conjugation of a dual quaternion $\hat{q} = q_r + \varepsilon q_d$ is given by $\bar{\hat{q}} = \bar{q}_r + \varepsilon \bar{q}_d$, where $\bar{q}_r$ and $\bar{q}_d$ are the conjugations of the quaternions q_r and q_d.

For the conjugation of quaternions see quat_conjugate.

If DualQuaternion is a unit dual quaternion and, hence, represents a 3D rigid transformation, its inverse is its conjugate, i.e., $\hat{q}^{-1} = \bar{\hat{q}}$. Consequently, DualQuaternionConjugate represents the inverse 3D rigid transformation of DualQuaternion. Therefore, dual_quat_conjugate can be used to invert a 3D rigid transformation analogously to pose_invert or hom_mat3d_invert.

Parameters

▷ **DualQuaternion** (input_control) ..... dual_quaternion(-array) ⇝ *real*
Dual quaternion.

▷ **DualQuaternionConjugate** (output_control) ..... dual_quaternion(-array) ⇝ *real*
Conjugate of the dual quaternion.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

dual_quat_compose, pose_to_dual_quat, screw_to_dual_quat

Possible Successors

dual_quat_compose, dual_quat_to_hom_mat3d, dual_quat_to_screw, dual_quat_interpolate, dual_quat_to_pose

Alternatives

pose_invert, hom_mat3d_invert

See also

dual_quat_normalize, serialize_dual_quat, deserialize_dual_quat, dual_quat_trans_line_3d, dual_quat_trans_point_3d, quat_conjugate

Module

Foundation

dual_quat_interpolate (: : DualQuaternionStart, DualQuaternionEnd, InterpPos : DualQuaternionInterpolated)

Interpolate two dual quaternions.

The operator dual_quat_interpolate interpolates the two dual quaternions DualQuaternionStart and DualQuaternionEnd at the interpolation position InterpPos.

For a brief introduction to dual quaternions, the used notation, and the relationship between dual quaternions and screws, see "Solution Guide III-C - 3D Vision".

For interpolations, the position InterpPos must lie within the interval $[0, 1]$. However, values outside this interval are also possible, which then correspond to an extrapolation. For InterpPos $= 0$, the interpolated dual quaternion DualQuaternionInterpolated corresponds to DualQuaternionStart. For InterpPos $= 1$, the interpolated dual quaternion DualQuaternionInterpolated corresponds to DualQuaternionEnd.

The interpolation is performed by using screw linear interpolation (ScLERP). If both DualQuaternionStart and DualQuaternionEnd are unit dual quaternions, DualQuaternionInterpolated will be a unit dual quaternion as well. Hence, dual_quat_interpolate can be used to smoothly interpolate between two 3D rigid transformations.

Note that the interpolation of the rotation parts of DualQuaternionStart and DualQuaternionEnd is performed identically to the interpolation of quaternions, i.e., the screw angle is interpolated linearly (see quat_interpolate). The interpolation of the translation part is done by linearly interpolating the screw translation.

It is possible to pass a tuple of values for InterpPos. In this case, a tuple of interpolated DualQuaternionInterpolated is returned, one for each value in InterpPos. This is more efficient than calling dual_quat_interpolate multiple times with a single value for InterpPos.

Parameters

▷ **DualQuaternionStart** (input_control) ..... dual_quaternion ⇝ *real*
Dual quaternion as the start point of the interpolation.

▷ **DualQuaternionEnd** (input_control) ..... dual_quaternion ⇝ *real*
Dual quaternion as the end point of the interpolation.

▷ **InterpPos** (input_control) ..... real(-array) ⇝ *real*
Interpolation parameter.
Default: 0.5
Suggested values: InterpPos ∈ {0.0, 0.25, 0.5, 0.75, 1.0}

▷ **DualQuaternionInterpolated** (output_control) ..... dual_quaternion(-array) ⇝ *real*
Interpolated dual quaternion.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

pose_to_dual_quat, screw_to_dual_quat, dual_quat_normalize

Possible Successors

dual_quat_to_hom_mat3d, dual_quat_to_pose, dual_quat_to_screw

See also

dual_quat_compose, dual_quat_conjugate, serialize_dual_quat, deserialize_dual_quat, dual_quat_trans_line_3d, dual_quat_trans_point_3d, quat_interpolate

Module

Foundation

dual_quat_normalize (: : DualQuaternion : DualQuaternionNormalized)

Normalize a dual quaternion.

The operator dual_quat_normalize normalizes the input dual quaternion DualQuaternion and returns the normalized dual quaternion, which is also called a unit dual quaternion, in DualQuaternionNormalized.

For a brief introduction to dual quaternions, the used notation, and the relationship between dual quaternions and screws, see "Solution Guide III-C - 3D Vision".

The norm of a unit dual quaternion $\hat{q} = q_r + \varepsilon q_d$ is 1, i.e., $\hat{q}\bar{\hat{q}} = 1$. This is equivalent to the following two conditions:

$$\begin{aligned} \|q_r\| = q_r \bar{q}_r &= 1 \\ q_r \bar{q}_d + \bar{q}_r q_d &= 0, \end{aligned}$$

where $\bar{q}$ represents the conjugate quaternion of q (see quat_conjugate).

A 3D rigid transformation can be represented by a unit dual quaternion.

Attention

If the norm of the real part of DualQuaternion is 0, dual_quat_normalize returns the error code 9310 (H_ERR_DQ_ZERO_NORM) because in this case no normalization is possible.

Parameters

- ▷ **DualQuaternion** (input_control) ..... dual_quaternion(-array) ⇝ *real*
  Unit dual quaternion.
- ▷ **DualQuaternionNormalized** (output_control) ..... dual_quaternion(-array) ⇝ *real*
  Normalized dual quaternion.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

dual_quat_compose

Possible Successors

dual_quat_to_hom_mat3d

See also

dual_quat_interpolate, pose_to_dual_quat, screw_to_dual_quat, dual_quat_to_screw, dual_quat_to_pose, dual_quat_conjugate, serialize_dual_quat, deserialize_dual_quat, dual_quat_trans_line_3d, dual_quat_trans_point_3d, quat_normalize

Module

Foundation

dual_quat_to_hom_mat3d (: : DualQuaternion : HomMat3D)

Convert a unit dual quaternion into a homogeneous transformation matrix.

The operator dual_quat_to_hom_mat3d converts a unit dual quaternion DualQuaternion, which represents a 3D rigid transformation, into its corresponding homogeneous transformation matrix HomMat3D.

For a brief introduction to dual quaternions, the used notation, and the relationship between dual quaternions and screws, see "Solution Guide III-C - 3D Vision".

The rotation part of HomMat3D is computed from the real part of the dual quaternion, as described in quat_to_hom_mat3d. The translation part $\mathbf{T}$ of HomMat3D is computed from the real and dual part of $\hat{q} = q_r + \varepsilon q_d$:

$$\begin{aligned} p &= q_d q_r \\ \mathbf{T} &= 2.0\mathbf{v}_p, \end{aligned}$$

where $\mathbf{v}_p$ is the vector part of the quaternion p.

Attention

HomMat3D will only be a valid rigid transformation matrix if DualQuaternion is a unit dual quaternion.

Parameters

▷ **DualQuaternion** (input_control) ..... dual_quaternion ⇝ *real*
Unit dual quaternion.

▷ **HomMat3D** (output_control) ..... hom_mat3d ⇝ *real*
Transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

screw_to_dual_quat, dual_quat_interpolate, dual_quat_compose, dual_quat_normalize

Possible Successors

affine_trans_point_3d

Alternatives

dual_quat_to_pose, dual_quat_to_screw

See also

dual_quat_compose, pose_to_dual_quat, dual_quat_conjugate, serialize_dual_quat, deserialize_dual_quat, dual_quat_trans_line_3d, dual_quat_trans_point_3d, quat_to_hom_mat3d

Module

Foundation

dual_quat_to_screw (: : DualQuaternion, ScrewFormat : AxisDirectionX, AxisDirectionY, AxisDirectionZ, AxisMomentOrPointX, AxisMomentOrPointY, AxisMomentOrPointZ, Rotation, Translation)

Convert a unit dual quaternion into a screw.

The operator dual_quat_to_screw converts the input unit dual quaternion DualQuaternion, which represents a 3D rigid transformation, into the parameters of the corresponding screw.

For a brief introduction to dual quaternions, the used notation, and the relationship between dual quaternions and screws, see "Solution Guide III-C - 3D Vision".

If ScrewFormat is set to *'moment'*, these parameters are returned in the corresponding parameters AxisDirectionX, AxisDirectionY, AxisDirectionZ, AxisMomentOrPointX, AxisMomentOrPointY, AxisMomentOrPointZ, Rotation, and Translation.

For convenience reasons, it is also possible to query the point on the screw axis that is closest to the origin instead of the moment of the screw axis. For this, ScrewFormat must be set to *'point'*. In this case, the coordinates of the point are returned in AxisMomentOrPointX, AxisMomentOrPointY, and AxisMomentOrPointZ.

Attention

dual_quat_to_screw assumes that the input DualQuaternion is a unit dual quaternion, and hence represents a 3D rigid transformation. Otherwise the returned screw parameters are not meaningful. Further note that

the screw axis for an identity transformation, i.e., no rotation and no translation, is undefined. In this case L is arbitrarily set to $(1, 0, 0)^T$.

Parameters

▷ **DualQuaternion** (input_control) . . . . . dual_quaternion ⇝ *real*
Unit dual quaternion.

▷ **ScrewFormat** (input_control) . . . . . string ⇝ *string*
Format of the screw parameters.
Default: 'moment'
List of values: ScrewFormat ∈ {'moment', 'point'}

▷ **AxisDirectionX** (output_control) . . . . . point3d.x ⇝ *real*
X component of the direction vector of the screw axis.

▷ **AxisDirectionY** (output_control) . . . . . point3d.y ⇝ *real*
Y component of the direction vector of the screw axis.

▷ **AxisDirectionZ** (output_control) . . . . . point3d.z ⇝ *real*
Z component of the direction vector of the screw axis.

▷ **AxisMomentOrPointX** (output_control) . . . . . point3d.x ⇝ *real*
X component of the moment vector or a point on the screw axis.

▷ **AxisMomentOrPointY** (output_control) . . . . . point3d.y ⇝ *real*
Y component of the moment vector or a point on the screw axis.

▷ **AxisMomentOrPointZ** (output_control) . . . . . point3d.z ⇝ *real*
Z component of the moment vector or a point on the screw axis.

▷ **Rotation** (output_control) . . . . . angle.rad ⇝ *real*
Rotation angle in radians.

▷ **Translation** (output_control) . . . . . real ⇝ *real*
Translation.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

dual_quat_compose, dual_quat_conjugate, dual_quat_interpolate

Alternatives

dual_quat_to_pose, dual_quat_to_hom_mat3d

See also

pose_to_dual_quat, dual_quat_normalize, serialize_dual_quat, deserialize_dual_quat, dual_quat_trans_line_3d, dual_quat_trans_point_3d, screw_to_dual_quat

Module

Foundation

dual_quat_trans_line_3d (: : DualQuaternion, LineFormat, LineDirectionX, LineDirectionY, LineDirectionZ, LineMomentOrPointX, LineMomentOrPointY, LineMomentOrPointZ : TransLineDirectionX, TransLineDirectionY, TransLineDirectionZ, TransLineMomentOrPointX, TransLineMomentOrPointY, TransLineMomentOrPointZ)

Transform a 3D line with a unit dual quaternion.

The operator dual_quat_trans_line_3d transforms a 3D line that is represented by its Plücker coordinates by a 3D rigid transformation that is given by the unit dual quaternion DualQuaternion.

For a brief introduction to dual quaternions, Plücker coordinates and the used notation, see "Solution Guide III-C - 3D Vision".

If LineFormat is set to *'moment'*, the moment of the line must be passed in LineMomentOrPointX, LineMomentOrPointY, and LineMomentOrPointZ.

If LineFormat is set to *'point'*, instead of the moment an arbitrary point on the line can be passed in LineMomentOrPointX, LineMomentOrPointY, and LineMomentOrPointZ.

The parameters of the transformed line are returned in TransLineDirectionX, TransLineDirectionY, TransLineDirectionZ, TransLineMomentOrPointX, TransLineMomentOrPointY, and TransLineMomentOrPointZ in the format that was specified in LineFormat. If LineFormat is set to *'point'*, the point on the line that is closest to the origin of the coordinate system is returned.

Attention

dual_quat_trans_line_3d returns meaningful results only if DualQuaternion is a unit dual quaternion and the length of the line direction is 1.

Parameters

- ▷ **DualQuaternion** (input_control) . . . . . dual_quaternion ⇝ *real* / integer
  Unit dual quaternion representing the transformation.
- ▷ **LineFormat** (input_control) . . . . . string ⇝ *string*
  Format of the line parameters.
  Default: 'moment'
  List of values: LineFormat ∈ {'moment', 'point'}
- ▷ **LineDirectionX** (input_control) . . . . . point3d.x(-array) ⇝ *real* / integer
  X component of the direction vector of the line.
- ▷ **LineDirectionY** (input_control) . . . . . point3d.y(-array) ⇝ *real* / integer
  Y component of the direction vector of the line.
- ▷ **LineDirectionZ** (input_control) . . . . . point3d.z(-array) ⇝ *real* / integer
  Z component of the direction vector of the line.
- ▷ **LineMomentOrPointX** (input_control) . . . . . point3d.x(-array) ⇝ *real* / integer
  X component of the moment vector or a point on the line.
- ▷ **LineMomentOrPointY** (input_control) . . . . . point3d.y(-array) ⇝ *real* / integer
  Y component of the moment vector or a point on the line.
- ▷ **LineMomentOrPointZ** (input_control) . . . . . point3d.z(-array) ⇝ *real* / integer
  Z component of the moment vector or a point on the line.
- ▷ **TransLineDirectionX** (output_control) . . . . . point3d.x(-array) ⇝ *real* / integer
  X component of the direction vector of the transformed line.
- ▷ **TransLineDirectionY** (output_control) . . . . . point3d.y(-array) ⇝ *real* / integer
  Y component of the direction vector of the transformed line.
- ▷ **TransLineDirectionZ** (output_control) . . . . . point3d.z(-array) ⇝ *real* / integer
  Z component of the direction vector of the transformed line.
- ▷ **TransLineMomentOrPointX** (output_control) . . . . . point3d.x(-array) ⇝ *real* / integer
  X component of the moment vector or a point on the transformed line.
- ▷ **TransLineMomentOrPointY** (output_control) . . . . . point3d.y(-array) ⇝ *real* / integer
  Y component of the moment vector or a point on the transformed line.
- ▷ **TransLineMomentOrPointZ** (output_control) . . . . . point3d.z(-array) ⇝ *real* / integer
  Z component of the moment vector or a point on the transformed line.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_to_pluecker_line, point_direction_to_pluecker_line

Possible Successors

dual_quat_compose, dual_quat_conjugate, pose_to_dual_quat

Alternatives

affine_trans_point_3d, dual_quat_trans_point_3d

See also

dual_quat_to_hom_mat3d, dual_quat_to_screw, dual_quat_to_pose, dual_quat_normalize, serialize_dual_quat, deserialize_dual_quat, quat_rotate_point_3d

Module

Foundation

dual_quat_trans_point_3d (: : DualQuaternion, Px, Py, Pz : Tx, Ty, Tz)

Transform a 3D point with a unit dual quaternion.

The operator dual_quat_trans_point_3d transforms a 3D point (Px, Py, Pz) by a 3D rigid transformation that is given by the unit dual quaternion DualQuaternion and returns the transformed 3D point in (Tx, Ty, Tz).

For a brief introduction to dual quaternions, see "Solution Guide III-C - 3D Vision".

Attention

dual_quat_trans_point_3d returns meaningful results only if DualQuaternion is a unit dual quaternion.

Parameters

▷ **DualQuaternion** (input_control) . . . . . dual_quaternion ⇝ *real* / integer
Unit dual quaternion representing the transformation.

▷ **Px** (input_control) . . . . . point3d.x(-array) ⇝ *real* / integer
Input point(s) (x coordinate).
Default: 64
Suggested values: Px ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Py** (input_control) . . . . . point3d.y(-array) ⇝ *real* / integer
Input point(s) (y coordinate).
Default: 64
Suggested values: Py ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Pz** (input_control) . . . . . point3d.z(-array) ⇝ *real* / integer
Input point(s) (z coordinate).
Default: 64
Suggested values: Pz ∈ {0, 16, 32, 64, 128, 256, 512, 1024}

▷ **Tx** (output_control) . . . . . point3d.x(-array) ⇝ *real*
Output point(s) (x coordinate).

▷ **Ty** (output_control) . . . . . point3d.y(-array) ⇝ *real*
Output point(s) (y coordinate).

▷ **Tz** (output_control) . . . . . point3d.z(-array) ⇝ *real*
Output point(s) (z coordinate).

Example

```
* dual_quat_trans_point_3d (DualQuat, Px, Py, Pz, Tx, Ty, Tz)
* is equivalent to the following code:
dual_quat_to_hom_mat3d (DualQuat, HomMat3D)
affine_trans_point_3d (HomMat3D, Px, Py, Pz, Tx, Ty, Tz)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Possible Predecessors

dual_quat_compose, pose_to_dual_quat, screw_to_dual_quat

Alternatives

dual_quat_to_hom_mat3d, affine_trans_point_3d

See also

dual_quat_to_pose, dual_quat_normalize, serialize_dual_quat, deserialize_dual_quat, quat_rotate_point_3d, dual_quat_trans_line_3d

Module

Foundation

screw_to_dual_quat (: : ScrewFormat, AxisDirectionX, AxisDirectionY, AxisDirectionZ, AxisMomentOrPointX, AxisMomentOrPointY, AxisMomentOrPointZ, Rotation, Translation : DualQuaternion)

Convert a screw into a dual quaternion.

The operator screw_to_dual_quat converts the screw parameters to the unit dual quaternion DualQuaternion, which represents a 3D rigid transformation.

For a brief introduction to dual quaternions, the used notation, and the relationship between dual quaternions and screws, see "Solution Guide III-C - 3D Vision".

A screw is described by the direction of the screw axis $\mathbf{L} = (L_x, L_y, L_z)^T$ with $\|\mathbf{L}\| = 1$, the moment of the screw axis $\mathbf{M} = (M_x, M_y, M_z)^T$ with $\mathbf{L} \cdot \mathbf{M} = 0$, the screw angle θ, and the screw translation d.

If ScrewFormat is set to *'moment'*, these parameters can be passed in the corresponding parameters AxisDirectionX, AxisDirectionY, AxisDirectionZ, AxisMomentOrPointX, AxisMomentOrPointY, AxisMomentOrPointZ, Rotation, and Translation.

For convenience reasons, it is also possible to specify an arbitrary point on the screw axis instead of the moment of the screw axis. For this, ScrewFormat must be set to *'point'* and the coordinates of the points must be passed in AxisMomentOrPointX, AxisMomentOrPointY, and AxisMomentOrPointZ.

Attention

screw_to_dual_quat assumes that the direction vector of the screw axis has length 1, i.e., $\sqrt{L_x^2 + L_y^2 + L_z^2} = 1$. Otherwise the returned dual quaternion is not meaningful.

Parameters

- ▷ **ScrewFormat** (input_control) ..... string ⇝ *string*
  Format of the screw parameters.
  Default: 'moment'
  List of values: ScrewFormat ∈ {'moment', 'point'}
- ▷ **AxisDirectionX** (input_control) ..... point3d.x ⇝ *real* / integer
  X component of the direction vector of the screw axis.
- ▷ **AxisDirectionY** (input_control) ..... point3d.y ⇝ *real* / integer
  Y component of the direction vector of the screw axis.
- ▷ **AxisDirectionZ** (input_control) ..... point3d.z ⇝ *real* / integer
  Z component of the direction vector of the screw axis.
- ▷ **AxisMomentOrPointX** (input_control) ..... point3d.x ⇝ *real* / integer
  X component of the moment vector or a point on the screw axis.
- ▷ **AxisMomentOrPointY** (input_control) ..... point3d.y ⇝ *real* / integer
  Y component of the moment vector or a point on the screw axis.
- ▷ **AxisMomentOrPointZ** (input_control) ..... point3d.z ⇝ *real* / integer
  Z component of the moment vector or a point on the screw axis.
- ▷ **Rotation** (input_control) ..... angle.rad ⇝ *real* / integer
  Rotation angle in radians.
- ▷ **Translation** (input_control) ..... real ⇝ *real* / integer
  Translation.

▷ **DualQuaternion** (output_control) . . . . . dual_quaternion ⇝ *real* / integer
Dual quaternion.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

dual_quat_compose, dual_quat_conjugate, dual_quat_interpolate

Alternatives

pose_to_dual_quat

See also

dual_quat_to_hom_mat3d, pose_to_dual_quat, dual_quat_to_screw, dual_quat_to_pose, dual_quat_normalize, serialize_dual_quat, deserialize_dual_quat, dual_quat_trans_line_3d, dual_quat_trans_point_3d, axis_angle_to_quat

Module

Foundation

serialize_dual_quat (: : DualQuaternion : SerializedItemHandle)

Serialize a dual quaternion.

serialize_dual_quat serializes the data of the DualQuaternion (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized dual quaternion is returned by the handle SerializedItemHandle and can be deserialized by deserialize_dual_quat.

Parameters

▷ **DualQuaternion** (input_control) . . . . . dual_quaternion ⇝ *real* / integer
Dual quaternion.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_dual_quat returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

axis_angle_to_quat

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_quat

Module

Foundation

27.4 Misc

convert_point_3d_cart_to_spher (: : X, Y, Z, EquatPlaneNormal, ZeroMeridian : Longitude, Latitude, Radius)

Convert Cartesian coordinates of a 3D point to spherical coordinates.

The operator convert_point_3d_cart_to_spher converts Cartesian coordinates of a 3D point, which are given in X, Y, and Z, into spherical coordinates. The spherical coordinates are returned in Longitude, Latitude, and Radius. The Longitude is returned in the range $[-\pi, +\pi]$ while the Latitude is returned in the range $[-\pi/2, +\pi/2]$. Furthermore, the latitude of the north pole is $\pi/2$, and hence, the latitude of the south pole is $-\pi/2$.

The orientation of the spherical coordinate system with respect to the Cartesian coordinate system can be specified with the parameters EquatPlaneNormal and ZeroMeridian.

EquatPlaneNormal determines the normal of the equatorial plane (longitude == 0) pointing to the north pole (positive latitude) and may take the following values:

'x': The equatorial plane is the yz plane. The positive x axis points to the north pole.

'-x': The equatorial plane is the yz plane. The positive x axis points to the south pole.

'y': The equatorial plane is the xz plane. The positive y axis points to the north pole.

'-y': The equatorial plane is the xz plane. The positive y axis points to the south pole.

'z': The equatorial plane is the xy plane. The positive z axis points to the north pole.

'-z': The equatorial plane is the xy plane. The positive z axis points to the south pole.

The position of the zero meridian can be specified with the parameter ZeroMeridian. For this, the coordinate axis (lying in the equatorial plane) that points to the zero meridian must be passed. The following values for ZeroMeridian are valid:

'x': The *positive* x axis points in the direction of the zero meridian.

'-x': The *negative* x axis points in the direction of the zero meridian.

'y': The *positive* y axis points in the direction of the zero meridian.

'-y': The *negative* y axis points in the direction of the zero meridian.

'z': The *positive* z axis points in the direction of the zero meridian.

'-z': The *negative* z axis points in the direction of the zero meridian.

Only reasonable combinations of EquatPlaneNormal and ZeroMeridian are permitted, i.e., the normal of the equatorial plane must not be parallel to the direction of the zero meridian. For example, the combination EquatPlaneNormal='y' and ZeroMeridian='-y' is not permitted.

Note that in order to guarantee a consistent conversion back from spherical to Cartesian coordinates by using convert_point_3d_spher_to_cart, the same values must be passed for EquatPlaneNormal and ZeroMeridian as were passed to convert_point_3d_cart_to_spher.

The operator convert_point_3d_cart_to_spher can be used, for example, to convert a given camera position into spherical coordinates. If multiple camera positions are converted in this way, one obtains a pose range (in spherical coordinates), which can be passed to create_shape_model_3d in order to create a 3D shape model.

Parameters

▷ **X** (input_control) ..... real(-array) ⇝ *real*
X coordinate of the 3D point.

▷ **Y** (input_control) ..... real(-array) ⇝ *real*
Y coordinate of the 3D point.

▷ **Z** (input_control) ..... real(-array) ⇝ *real*
Z coordinate of the 3D point.

▷ **EquatPlaneNormal** (input_control) ..... string ⇝ *string*
Normal vector of the equatorial plane (points to the north pole).
Default: '-y'
List of values: EquatPlaneNormal ∈ {'x', 'y', 'z', '-x', '-y', '-z'}

▷ **ZeroMeridian** (input_control) ..... string ⇝ *string*
Coordinate axis in the equatorial plane that points to the zero meridian.
Default: '-z'
List of values: ZeroMeridian ∈ {'x', 'y', 'z', '-x', '-y', '-z'}

▷ **Longitude** (output_control) ..... angle.rad(-array) ⇝ *real*
Longitude of the 3D point.

▷ **Latitude** (output_control) ..... angle.rad(-array) ⇝ *real*
Latitude of the 3D point.

▷ **Radius** (output_control) ..... real(-array) ⇝ *real*
Radius of the 3D point.

Result

If the parameters are valid, the operator convert_point_3d_cart_to_spher returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

create_shape_model_3d, find_shape_model_3d

See also

convert_point_3d_spher_to_cart

Module

3D Metrology

convert_point_3d_spher_to_cart (: : Longitude, Latitude, Radius, EquatPlaneNormal, ZeroMeridian : X, Y, Z)

Convert spherical coordinates of a 3D point to Cartesian coordinates.

The operator convert_point_3d_spher_to_cart converts the spherical coordinates of a 3D point, which are given in Longitude, Latitude, and Radius, into the Cartesian coordinates X, Y, and Z. The spherical coordinates Longitude and Latitude must be specified in radians. Furthermore, the Latitude must be within the range $[-\pi/2, +\pi/2]$, where the latitude of the north pole is $\pi/2$, and hence, the latitude of the south pole is $-\pi/2$.

The orientation of the spherical coordinate system with respect to the Cartesian coordinate system can be specified with the parameters EquatPlaneNormal and ZeroMeridian.

EquatPlaneNormal determines the normal of the equatorial plane (longitude == 0) pointing to the north pole (positive latitude) and may take the following values:

'x': The equatorial plane is the yz plane. The positive x axis points to the north pole.

'-x': The equatorial plane is the yz plane. The positive x axis points to the south pole.

'y': The equatorial plane is the xz plane. The positive y axis points to the north pole.

'-y': The equatorial plane is the xz plane. The positive y axis points to the south pole.

'z': The equatorial plane is the xy plane. The positive z axis points to the north pole.

'-z': The equatorial plane is the xy plane. The positive z axis points to the south pole.

The position of the zero meridian can be specified with the parameter ZeroMeridian. For this, the coordinate axis (lying in the equatorial plane) that points to the zero meridian must be passed. The following values for ZeroMeridian are valid:

'x': The *positive* x axis points in the direction of the zero meridian.

'-x': The *negative* x axis points in the direction of the zero meridian.

'y': The *positive* y axis points in the direction of the zero meridian.

'-y': The *negative* y axis points in the direction of the zero meridian.

'z': The *positive* z axis points in the direction of the zero meridian.

'-z': The *negative* z axis points in the direction of the zero meridian.

Only reasonable combinations of `EquatPlaneNormal` and `ZeroMeridian` are permitted, i.e., the normal of the equatorial plane must not be parallel to the direction of the zero meridian. For example, the combination `EquatPlaneNormal`='*y*' and `ZeroMeridian`='*-y*' is not permitted.

Note that in order to guarantee a consistent conversion back from Cartesian to spherical coordinates by using `convert_point_3d_cart_to_spher`, the same values must be passed for `EquatPlaneNormal` and `ZeroMeridian` as were passed to `convert_point_3d_spher_to_cart`.

The operator `convert_point_3d_spher_to_cart` can be used, for example, to convert a camera position that is given in spherical coordinates into Cartesian coordinates. The result can then be utilized to create a complete camera pose by passing the Cartesian coordinates to `create_cam_pose_look_at_point`.

Parameters

- ▷ **`Longitude`** (input_control) ..... angle.rad(-array) ⇝ *real*
  Longitude of the 3D point.
- ▷ **`Latitude`** (input_control) ..... angle.rad(-array) ⇝ *real*
  Latitude of the 3D point.
  Restriction: - pi / 2 <= Latitude && Latitude <= pi / 2
- ▷ **`Radius`** (input_control) ..... real(-array) ⇝ *real*
  Radius of the 3D point.
- ▷ **`EquatPlaneNormal`** (input_control) ..... string ⇝ *string*
  Normal vector of the equatorial plane (points to the north pole).
  Default: '-y'
  List of values: `EquatPlaneNormal` ∈ {'x', 'y', 'z', '-x', '-y', '-z'}
- ▷ **`ZeroMeridian`** (input_control) ..... string ⇝ *string*
  Coordinate axis in the equatorial plane that points to the zero meridian.
  Default: '-z'
  List of values: `ZeroMeridian` ∈ {'x', 'y', 'z', '-x', '-y', '-z'}
- ▷ **`X`** (output_control) ..... real(-array) ⇝ *real*
  X coordinate of the 3D point.
- ▷ **`Y`** (output_control) ..... real(-array) ⇝ *real*
  Y coordinate of the 3D point.
- ▷ **`Z`** (output_control) ..... real(-array) ⇝ *real*
  Z coordinate of the 3D point.

Result

If the parameters are valid, the operator `convert_point_3d_spher_to_cart` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_shape_model_3d_params`

See also

`convert_point_3d_cart_to_spher`

Module

3D Metrology

27.5 Poses

A pose describes a rigid 3D transformation, i.e., a transformation consisting of an arbitrary translation and rotation. In HALCON, a pose is a tuple with 7 parameters: 3 parameters specifying the translation (`TransX`, `TransY`, and `TransZ`), 3 parameters describing the rotation (`RotX`, `RotY`, and `RotZ`). The last parameter codes the order of the translations and the rotations (as well as the direction of rotation). Further information about these parameters can be found in the documentation of `create_pose` and in the "`Solution Guide III-C - 3D Vision`".

3D poses can be interpreted in two ways: First, to describe the position and orientation of one coordinate system relative to another (e.g., the pose of a part's coordinate system relative to the camera coordinate system - in short: the pose of the part relative to the camera). Following the second interpretation, a pose describes how coordinates can be transformed between two coordinate systems (e.g., to transform points from part coordinates into camera coordinates).

A pose that transforms point coordinates from coordinate system 1 ($cs1$) to coordinate system 2 ($cs2$) is denoted as ${}^{cs2}\mathbf{P}_{cs1}$. The corresponding transformation of a point given in $cs1$ ($\mathbf{p}^{cs1}$) into $cs2$ ($\mathbf{p}^{cs2}$) is denoted as

$$\mathbf{p}^{cs2} = {}^{cs2}\mathbf{P}_{cs1} \cdot \mathbf{p}^{cs1}.$$

It should be noted that not the pose (as a tuple) but the transformation described by this pose is used for the multiplication and the above notation is only used for readability. Hence, the pose ${}^{cs2}\mathbf{P}_{cs1}$ describes the rigid transformation that is represented by the homogeneous transformation matrix $\mathbf{H}({}^{cs2}\mathbf{P}_{cs1}) = {}^{cs2}\mathbf{H}_{cs1}$ (see, e.g., the documentation of `create_pose` for further details)

$$\begin{pmatrix} \mathbf{p}^{cs2} \\ 1 \end{pmatrix} = {}^{cs2}\mathbf{H}_{cs1} \cdot \begin{pmatrix} \mathbf{p}^{cs1} \\ 1 \end{pmatrix}.$$

Consequently, the pose ${}^{cs2}\mathbf{P}_{cs1}$ describes the transformation of points from $cs1$ into $cs2$. Furthermore, as mentioned above, it also describes the transformation of the coordinate system itself, however, in reverse order: Thus, ${}^{cs2}\mathbf{P}_{cs1}$ describes how coordinate system 2 must be transformed to obtain coordinate system 1, and hence, the pose of coordinate system 2 relative to system 1.

With this notation, poses can easily concatenated like homogeneous matrices, e.g.,

$${}^{cs2}\mathbf{P}_{cs0} = {}^{cs2}\mathbf{P}_{cs1} \cdot {}^{cs1}\mathbf{P}_{cs0}.$$

Such a concatenation can be done using e.g., `pose_compose`.

`convert_pose_type` (: : `PoseIn`, `OrderOfTransform`, `OrderOfRotation`, `ViewOfTransform` : `PoseOut`)

Change the representation type of a 3D pose.

`convert_pose_type` converts the 3D pose `PoseIn` into a 3D pose `PoseOut` with a different representation type. See `create_pose` for details about 3D poses, their representation types, and the meaning of the parameters `OrderOfTransform`, `OrderOfRotation`, and `ViewOfTransform`.

Note that `convert_pose_type` only changes the representation of a 3D pose, but not the rigid transformation described by the pose.

Parameters

▷ **`PoseIn`** (input_control) ..... pose ⇝ *real* / integer
Original 3D pose.
Number of elements: 7

▷ **`OrderOfTransform`** (input_control) ..... string ⇝ *string*
Order of rotation and translation.
Default: 'Rp+T'
Suggested values: `OrderOfTransform` ∈ {'Rp+T', 'R(p-T)'}

▷ **OrderOfRotation** (input_control) ..... string ⇝ *string*
Meaning of the rotation values.
Default: 'gba'
Suggested values: OrderOfRotation ∈ {'gba', 'abg', 'rodriguez'}

▷ **ViewOfTransform** (input_control) ..... string ⇝ *string*
View of transformation.
Default: 'point'
Suggested values: ViewOfTransform ∈ {'point', 'coordinate_system'}

▷ **PoseOut** (output_control) ..... pose ⇝ *real* / integer
3D transformation.
Number of elements: 7

Example

```
* Define a pose.
create_pose (0.1, 0.1, 0.1, 90, 90, 90, 'Rp+T', 'gba', 'point', Pose)
* Convert pose to a pose with desired semantic.
convert_pose_type (Pose, 'Rp+T', 'abg', 'point', Pose2)
```

Result

convert_pose_type returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_pose, hom_mat3d_to_pose, camera_calibration, hand_eye_calibration

Possible Successors

write_pose

See also

create_pose, get_pose_type, write_pose, read_pose

Module

Foundation

create_pose (: : TransX, TransY, TransZ, RotX, RotY, RotZ, OrderOfTransform, OrderOfRotation, ViewOfTransform : Pose)

Create a 3D pose.

create_pose creates the 3D pose Pose. A pose describes a rigid 3D transformation, i.e., a transformation consisting of an arbitrary translation and rotation, with 6 parameters: TransX, TransY, and TransZ specify the translation along the x-, y-, and z-axis, respectively, while RotX, RotY, and RotZ describe the rotation.

3D poses are typically used in two ways: First, to describe the position and orientation of one coordinate system relative to another (e.g., the pose of a part's coordinate system relative to the camera coordinate system - in short: the pose of the part relative to the camera) and secondly, to describe how coordinates can be transformed between two coordinate systems (e.g., to transform points from part coordinates into camera coordinates).

Representation of orientation (rotation)

A 3D rotation around an arbitrary axis can be represented by 3 parameters in multiple ways. HALCON lets you choose between three of them with the parameter OrderOfRotation: If you pass the value *'gba'*, the rotation is described by the following chain of rotations around the three axes (see hom_mat3d_rotate for the content for the rotation matrices $\mathbf{R}_x$, $\mathbf{R}_y$, and $\mathbf{R}_z$):

$$\mathbf{R}_{gba} = \mathbf{R}_x(\texttt{RotX}) \cdot \mathbf{R}_y(\texttt{RotY}) \cdot \mathbf{R}_z(\texttt{RotZ})$$

$\mathbf{R}_{gba}$ is referred to as the Yaw-Pitch-Roll convention in the literature. Please note that you can "read" this chain in two ways: If you start from the right, the rotations are always performed relative to the global (i.e., fixed or "old") coordinate system. Thus, $\mathbf{R}_{gba}$ can be read as follows: First rotate around the z-axis, then around the "old" y-axis, and finally around the "old" x-axis. In contrast, if you read from the left to the right, the rotations are performed relative to the local (i.e., "new") coordinate system. Then, $\mathbf{R}_{gba}$ corresponds to the following: First rotate around the x-axis, the around the "new" y-axis, and finally around the "new(est)" z-axis.

Reading $\mathbf{R}_{gba}$ from right to left corresponds to the following sequence of operator calls:

```
hom_mat3d_identity(HomMat3DIdent)
hom_mat3d_rotate(HomMat3DIdent, RotZ, 'z', 0, 0, 0, HomMat3DRotZ)
hom_mat3d_rotate(HomMat3DRotZ, RotY, 'y', 0, 0, 0, HomMat3DRotYZ)
hom_mat3d_rotate(HomMat3DRotYZ, RotX, 'x', 0, 0, 0, HomMat3DXYZ)
```

In contrast, reading from left to right corresponds to the following operator sequence:

```
hom_mat3d_identity(HomMat3DIdent)
hom_mat3d_rotate_local(HomMat3DIdent, RotX, 'x', HomMat3DRotX)
hom_mat3d_rotate_local(HomMat3DRotX, RotY, 'y', HomMat3DRotXY)
hom_mat3d_rotate_local(HomMat3DRotXY, RotZ, 'z', HomMat3DXYZ)
```

When passing *'abg'* in `OrderOfRotation`, the rotation corresponds to the following chain:

$$\mathbf{R}_{abg} = \mathbf{R}_z(\texttt{RotZ}) \cdot \mathbf{R}_y(\texttt{RotY}) \cdot \mathbf{R}_x(\texttt{RotX})$$

$\mathbf{R}_{abg}$ is referred to as the Roll-Pitch-Yaw convention in the literature.

If you pass *'rodriguez'* in `OrderOfRotation`, the rotation parameters `RotX`, `RotY`, and `RotZ` are interpreted as the x-, y-, and z-component of the so-called Rodriguez rotation vector. The direction of the vector defines the (arbitrary) axis of rotation. The length of the vector usually defines the rotation angle with positive orientation. Here, a variation of the Rodriguez vector is used, where the length of the vector defines the tangent of half the rotation angle:

$$\mathbf{R}_{rodriguez} \quad = \quad rotate\ around \quad \begin{pmatrix} \texttt{RotX} \\ \texttt{RotY} \\ \texttt{RotZ} \end{pmatrix} \quad by \quad 2 \cdot \arctan(\sqrt{\texttt{RotX}^2 + \texttt{RotY}^2 + \texttt{RotZ}^2})$$

Please note that these 3D poses can be ambiguous, meaning a homogeneous transformation matrix can have several pose representations. For example, for $\mathbf{R}_{gba}$ with $b = \pm 90$ the following poses correspond to the same homogeneous transformation matrix:

```
create_pose(0, 0, 0, 30, 90, 54, 'Rp+T', 'gba', 'point', Pose1)
create_pose(0, 0, 0, 17, 90, 67, 'Rp+T', 'gba', 'point', Pose2)
```

If this leads to problems, you can instead use homogeneous transformation matrices or quaternions (`axis_angle_to_quat`) to represent rotations.

Corresponding homogeneous transformation matrix

You can obtain the homogeneous transformation matrix corresponding to a pose with the operator `pose_to_hom_mat3d`. In the standard definition, this is the following homogeneous transformation matrix which can be split into two separate matrices, one for the translation ($\mathbf{H}(\mathbf{T})$) and one for the rotation ($\mathbf{H}(\mathbf{R})$):

$$\mathbf{H}_{pose} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ 0\,0\,0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}(\texttt{RotX},\texttt{RotY},\texttt{RotZ}) & \begin{matrix}\texttt{TransX}\\ \texttt{TransY}\\ \texttt{TransZ}\end{matrix} \\ \begin{matrix}0 & 0 & 0\end{matrix} & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} 1 & 0 & 0 & \texttt{TransX} \\ 0 & 1 & 0 & \texttt{TransY} \\ 0 & 0 & 1 & \texttt{TransZ} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}(\texttt{RotX},\texttt{RotY},\texttt{RotZ}) & \begin{matrix}0\\0\\0\end{matrix} \\ \begin{matrix}0 & 0 & 0\end{matrix} & 1 \end{bmatrix} = \mathbf{H}(\mathbf{T}) \cdot \mathbf{H}(\mathbf{R})$$

Transformation of coordinates

The following equation describes how a point can be transformed from coordinate system 1 (*cs*1) into coordinate system 2 (*cs*2) with a pose, or more exactly, with the corresponding homogeneous transformation matrix ${}^{cs2}\mathbf{H}_{cs1}$ (input and output points as homogeneous vectors, see also `affine_trans_point_3d`). Note that to transform points from *cs*1 into *cs*2, you use the transformation matrix that describes the pose of *cs*1 relative to *cs*2.

$$\begin{pmatrix} \mathbf{p}^{cs2} \\ 1 \end{pmatrix} = {}^{cs2}\mathbf{H}_{cs1} \cdot \begin{pmatrix} \mathbf{p}^{cs1} \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}(\texttt{RotX},\texttt{RotY},\texttt{RotZ}) \cdot \mathbf{p}^{cs1} + \begin{pmatrix} \texttt{TransX} \\ \texttt{TransY} \\ \texttt{TransZ} \end{pmatrix} \\ 1 \end{pmatrix}$$

This corresponds to the following operator calls:

```
pose_to_hom_mat3d(PoseOf1In2, HomMat3DFrom1In2)
affine_trans_point_3d(HomMat3DFrom1In2, P1X, P1Y, P1Z, P2X, P2Y,
P2Z)
```

Non-standard pose definitions

So far, we described the standard pose definition. To create such poses, you select the (default) values *'Rp+T'* for the parameter `OrderOfTransform` and *'point'* for `ViewOfTransform`. By specifying other values for these parameters, you can create non-standard poses types which we describe briefly below. Please note that these representation types are only supported for backwards compatibility; we strongly recommend to use the standard types.

If you select *'R(p-T)'* for `OrderOfTransform`, the created pose corresponds to the following chain of transformations, i.e., the sequence of rotation and translation is reversed and the translation is negated:

$$\mathbf{H}_{R(p-T)} = \begin{bmatrix} \mathbf{R}(\texttt{RotX},\texttt{RotY},\texttt{RotZ}) & \begin{matrix}0\\0\\0\end{matrix} \\ \begin{matrix}0 & 0 & 0\end{matrix} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -\texttt{TransX} \\ 0 & 1 & 0 & -\texttt{TransY} \\ 0 & 0 & 1 & -\texttt{TransZ} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{R}) \cdot \mathbf{H}(-\mathbf{T})$$

If you select *'coordinate_system'* for `ViewOfTransform`, the sequence of transformations remains constant, but the rotation angles are negated. Please note that, contrary to its name, this is not equivalent to transforming a coordinate system!

$$\mathbf{H}_{coordinate\_system} = \begin{bmatrix} 1 & 0 & 0 & \texttt{TransX} \\ 0 & 1 & 0 & \texttt{TransY} \\ 0 & 0 & 1 & \texttt{TransZ} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}(-\texttt{RotX},-\texttt{RotY},-\texttt{RotZ}) & \begin{matrix}0\\0\\0\end{matrix} \\ \begin{matrix}0 & 0 & 0\end{matrix} & 1 \end{bmatrix}$$

Returned data structure

The created 3D pose is returned in `Pose` which is a tuple of length seven. The first three elements hold the translation parameters `TransX`, `TransY`, and `TransZ`, followed by the rotation parameters `RotX`, `RotY`, and `RotZ`. The last element codes the representation type of the pose that you selected with the parameters `OrderOfTransform`, `OrderOfRotation`, and `ViewOfTransform`. The following table lists the possible combinations. As already noted, we recommend to use only the representation types with `OrderOfTransform` = *'Rp+T'* and `ViewOfTransform` = *'point'* (codes 0, 2, and 4).

OrderOfTransform	OrderOfRotation	ViewOfTransform	Code
'Rp+T'	*'gba'*	*'point'*	0
'Rp+T'	*'abg'*	*'point'*	2
'Rp+T'	*'rodriguez'*	*'point'*	4
'Rp+T'	*'gba'*	*'coordinate_system'*	1
'Rp+T'	*'abg'*	*'coordinate_system'*	3
'Rp+T'	*'rodriguez'*	*'coordinate_system'*	5
'R(p-T)'	*'gba'*	*'point'*	8
'R(p-T)'	*'abg'*	*'point'*	10
'R(p-T)'	*'rodriguez'*	*'point'*	12
'R(p-T)'	*'gba'*	*'coordinate_system'*	9
'R(p-T)'	*'abg'*	*'coordinate_system'*	11
'R(p-T)'	*'rodriguez'*	*'coordinate_system'*	13

You can convert poses into other representation types using `convert_pose_type` and query the type using `get_pose_type`.

Parameters

▷ **`TransX`** (input_control) . . . . . real ⇝ *real*
Translation along the x-axis (in [m]).
Default: 0.1
Suggested values: `TransX` ∈ {-1.0, -0.75, -0.5, -0.25, -0.2, -0.1, -0.5, -0.25, -0.125, -0.01, 0.0, 0.01, 0.125, 0.25, 0.5, 0.1, 0.2, 0.25, 0.5, 0.75, 1.0}

▷ **`TransY`** (input_control) . . . . . real ⇝ *real*
Translation along the y-axis (in [m]).
Default: 0.1
Suggested values: `TransY` ∈ {-1.0, -0.75, -0.5, -0.25, -0.2, -0.1, -0.5, -0.25, -0.125, -0.01, 0.0, 0.01, 0.125, 0.25, 0.5, 0.1, 0.2, 0.25, 0.5, 0.75, 1.0}

▷ **`TransZ`** (input_control) . . . . . real ⇝ *real*
Translation along the z-axis (in [m]).
Default: 0.1
Suggested values: `TransZ` ∈ {-1.0, -0.75, -0.5, -0.25, -0.2, -0.1, -0.5, -0.25, -0.125, -0.01, 0.0, 0.01, 0.125, 0.25, 0.5, 0.1, 0.2, 0.25, 0.5, 0.75, 1.0}

▷ **`RotX`** (input_control) . . . . . real ⇝ *real*
Rotation around x-axis or x component of the Rodriguez vector (in [°] or without unit).
Default: 90.0
Suggested values: `RotX` ∈ {0.0, 90.0, 180.0, 270.0}
Value range: 0 ≤ `RotX` ≤ 360

▷ **`RotY`** (input_control) . . . . . real ⇝ *real*
Rotation around y-axis or y component of the Rodriguez vector (in [°] or without unit).
Default: 90.0
Suggested values: `RotY` ∈ {0.0, 90.0, 180.0, 270.0}
Value range: 0 ≤ `RotY` ≤ 360

▷ **`RotZ`** (input_control) . . . . . real ⇝ *real*
Rotation around z-axis or z component of the Rodriguez vector (in [°] or without unit).
Default: 90.0
Suggested values: `RotZ` ∈ {0.0, 90.0, 180.0, 270.0}
Value range: 0 ≤ `RotZ` ≤ 360

▷ **`OrderOfTransform`** (input_control) . . . . . string ⇝ *string*
Order of rotation and translation.
Default: 'Rp+T'
Suggested values: `OrderOfTransform` ∈ {'Rp+T', 'R(p-T)'}

▷ **`OrderOfRotation`** (input_control) . . . . . string ⇝ *string*
Meaning of the rotation values.
Default: 'gba'
Suggested values: `OrderOfRotation` ∈ {'gba', 'abg', 'rodriguez'}

▷ **ViewOfTransform** (input_control) ..... string ⇝ *string*
View of transformation.
Default: 'point'
Suggested values: ViewOfTransform ∈ {'point', 'coordinate_system'}

▷ **Pose** (output_control) ..... pose ⇝ *real* / integer
3D pose.
Number of elements: 7

Example

```
* Create a pose.
create_pose (0.1, 0.2, 0.3, 40, 50, 60, 'Rp+T', 'gba', 'point', Pose)
```

Result

create_pose returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

pose_to_hom_mat3d, write_pose, camera_calibration, hand_eye_calibration

Alternatives

read_pose, hom_mat3d_to_pose

See also

hom_mat3d_rotate, hom_mat3d_translate, convert_pose_type, get_pose_type, hom_mat3d_to_pose, pose_to_hom_mat3d, write_pose, read_pose

Module

Foundation

deserialize_pose (: : SerializedItemHandle : Pose)

Deserialize a serialized pose.

deserialize_pose deserializes a pose, that was serialized by serialize_pose (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized pose is defined by the handle SerializedItemHandle. The deserialized values are stored in an automatically created pose with the handle Pose.

Parameters

▷ **SerializedItemHandle** (input_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **Pose** (output_control) ..... pose ⇝ *real* / integer
3D pose.
Number of elements: 7

Result

If the parameters are valid, the operator deserialize_pose returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_pose

Module

Foundation

dual_quat_to_pose (: : DualQuaternion : Pose)

Convert a dual quaternion to a 3D pose.

The operator dual_quat_to_pose converts the input DualQuaternion into its corresponding 3D Pose.

For a brief introduction to dual quaternions, the used notation, and the relationship between dual quaternions and screws, see "Solution Guide III-C - 3D Vision".

Multiple dual quaternions can be passed in DualQuaternion, in which case a pose is returned for each of them.

Attention

dual_quat_to_pose assumes that the input DualQuaternion is a unit dual quaternion, and hence represents a 3D rigid transformation. Otherwise the returned pose is not meaningful.

Parameters

▷ **DualQuaternion** (input_control) . . . . . dual_quaternion(-array) ⇝ *real*
Unit dual quaternion.

▷ **Pose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
3D pose.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

dual_quat_compose, dual_quat_interpolate, screw_to_dual_quat, dual_quat_conjugate

Possible Successors

camera_calibration, hand_eye_calibration, write_pose, disp_caltab, sim_caltab

Alternatives

dual_quat_to_screw, dual_quat_to_hom_mat3d

See also

pose_to_dual_quat, dual_quat_normalize, serialize_dual_quat, deserialize_dual_quat, dual_quat_trans_line_3d, dual_quat_trans_point_3d, quat_to_pose

Module

Foundation

get_circle_pose (Contour : : CameraParam, Radius, OutputType : Pose1, Pose2)

Determine the 3D pose of a circle from its perspective 2D projection.

Each ellipse in an image can be interpreted as the perspective projection of a circle into the image. In fact, for a given radius of the circle, there exist two differently oriented circles in 3D that result in the same projection. get_circle_pose determines the 3D positions and orientations of these two circles. First, each Contour is approximated by an ellipse. Then, based on the internal camera parameters (CameraParam) and the radius of the circle in 3D (Radius), the 3D positions and orientations (Pose1,Pose2) are determined in camera coordinates.

Depending on the value of the parameter OutputType, the position and orientation is returned as a 3D pose (OutputType = *'pose'*) or in the form of the center of the 3D circle and the normal vector of the plane in which the circle lies (OutputType = *'center_normal'*). In the former case, the angle for the rotation around the z axis is set to zero, because it cannot be determined. In the latter case, the first three elements of the output parameters Pose1 and Pose2 contain the position of the center of the circle. The following three elements contain the normal vector. The normal vectors are normalized and oriented such that they point away from the optical center which is the origin of the camera coordinate system. If OutputType is set to *'center_normal'*, the output parameters Pose1 and Pose2 contain only six elements which describe the position and orientation of the circle instead of the seven elements of the 3D pose that are returned if OutputType is set to *'pose'*.

If more than one contour is passed in Contour, Radius must either contain a tuple that contains a value for each contour or only one value which is then used for all contours. The resulting positions and orientations are stored one after another in Pose1 and Pose2, i.e., Pose1 and Pose2 contain first the pose or the position and the normal vector of the first contour, followed by the respective values for the second contour and so on.

Attention

The accuracy of the determined poses depends heavily on the accuracy of the extracted contours. The extraction of curved edges using relatively large filter masks leads to a slightly shifted edge position. Edge extraction approaches that are based on the first derivative of the image function (e.g., edges_sub_pix) yield edges that are shifted towards the center of curvature, i.e., extracted ellipses will be slightly to small. Approaches that are based on the second derivative of the image function (laplace_of_gauss followed by zero_crossing_sub_pix) result in edges that are shifted away from the center of curvature, i.e., extracted ellipses will be slightly too large.

These effects increase with the curvature of the edge and with the size of the filter mask that is used for the edge extraction. Therefore, to achieve high accuracy, the ellipses should appear large in the image and the filter parameter should be chosen such that small filter masks are used (see info_edges).

Parameters

- ▷ **Contour** (input_object) . . . . . xld(-array) ⇝ *object*
  Contours to be examined.
- ▷ **CameraParam** (input_control) . . . . . campar ⇝ *real* / integer / string
  Internal camera parameters.
- ▷ **Radius** (input_control) . . . . . number(-array) ⇝ *real*
  Radius of the circle in object space.
  Number of elements: Radius == Contour || Radius == 1
  Restriction: Radius > 0.0
- ▷ **OutputType** (input_control) . . . . . string ⇝ *string*
  Type of output parameters.
  Default: 'pose'
  List of values: OutputType ∈ {'pose', 'center_normal'}
- ▷ **Pose1** (output_control) . . . . . real-array ⇝ *real* / integer
  3D pose of the first circle.
  Number of elements: Pose1 == 7 * Contour || Pose1 == 6 * Contour
- ▷ **Pose2** (output_control) . . . . . real-array ⇝ *real* / integer
  3D pose of the second circle.
  Number of elements: Pose2 == 7 * Contour || Pose2 == 6 * Contour

Result

get_circle_pose returns 2 (H_MSG_TRUE) if all parameter values are correct and the position of the circle has been determined successfully. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

edges_sub_pix

Alternatives

find_marks_and_pose, camera_calibration

See also

get_rectangle_pose, fit_ellipse_contour_xld

Module

3D Metrology

get_pose_type (: : Pose : OrderOfTransform, OrderOfRotation, ViewOfTransform)

Get the representation type of a 3D pose.

With get_pose_type, the representation type of the 3D pose Pose can be queried. See create_pose for details about 3D poses, their representation types, and the meaning of the parameters OrderOfTransform, OrderOfRotation, and ViewOfTransform.

Parameters

▷ **Pose** (input_control) . . . . . pose ⇝ *real / integer*
3D pose.
Number of elements: 7

▷ **OrderOfTransform** (output_control) . . . . . string ⇝ *string*
Order of rotation and translation.

▷ **OrderOfRotation** (output_control) . . . . . string ⇝ *string*
Meaning of the rotation values.

▷ **ViewOfTransform** (output_control) . . . . . string ⇝ *string*
View of transformation.

Result

create_pose returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_pose, hom_mat3d_to_pose, camera_calibration, hand_eye_calibration

Possible Successors

convert_pose_type

See also

create_pose, convert_pose_type, write_pose, read_pose

Module

Foundation

get_rectangle_pose (Contour : : CameraParam, Width, Height, WeightingMode, ClippingFactor : Pose, CovPose, Error)

Determine the 3D pose of a rectangle from its perspective 2D projection

A rectangle in space is projected as a general quadrangle into the image. get_rectangle_pose determines the Pose of the rectangle from this projection (Contour).

The algorithm works as follows: First, Contour is segmented into four line segments and their intersections are considered as corners of the contour. The corners together with the internal camera parameters (CameraParam) and the rectangle size in meters (Width, Height) are used for an initial estimation of the rectangle pose. Then, the final Pose is refined with a non-linear optimization by minimizing the geometrical distance of the contour points from the back projection of the rectangle in the image.

The operator supports only area scan pinhole (projective) cameras. An error is returned if CameraParam specifies a line scan or a telecentric camera (see also Calibration).

Width and Height specify the size of the rectangle in x and y dimensions, respectively, in its coordinate system. The origin of this coordinate system is in the center of the rectangle. The z axis points away from the camera.

The arguments WeightingMode and ClippingFactor can be used to damp the impact of outliers on the algorithm. If WeightingMode is set to *'tukey'* or *'huber'*, the contour points are weighted based on the approach of Tukey or Huber respectively. In such a case a robust error statistics is used to estimate the standard deviation of the distances of the contour points from the backprojected rectangle excluding outliers. The parameter ClippingFactor (a scaling factor for the standard deviation) controls the amount of damping outliers: The smaller the value chosen for ClippingFactor the more outliers are detected. See a discussion about the properties of the different weighting modes in fit_line_contour_xld. Note that, unlike by fit_line_contour_xld, for the rectangle pose estimation the approach of Huber is recommended.

Output

The resulting Pose is of code 0 (see create_pose) and represents the pose of the center of the rectangle. You can compute the pose of the corners of the rectangle as follows:

```
set_origin_pose(Pose, Width/2, -Height/2, 0, PoseCorner1)
set_origin_pose(Pose, Width/2, Height/2, 0, PoseCorner2)
set_origin_pose(Pose, -Width/2, Height/2, 0, PoseCorner3)
set_origin_pose(Pose, -Width/2, -Height/2, 0, PoseCorner4)
```

A rectangle is symmetric with respect to its x, y, and z axis and one and the same contour can represent a rectangle in 4 different poses. The angles in Pose are normalized to be in the range $[-90; 90]$ degrees and the rest of the 4 possible poses can be computed by combining flips around the corresponding axis:

```
* NOTE: the following code works ONLY for pose of code 0
* as it is returned by get_rectangle_pose
*
* flip around z-axis
PoseFlippedZ := Pose
PoseFlippedZ[5] := PoseFlippedZ[5]+180
* flip around y-axis
PoseFlippedY := Pose
PoseFlippedY[4] := PoseFlippedY[4]+180
PoseFlippedY[5] := -PoseFlippedY[5]
* flip around x-axis
PoseFlippedX := Pose
PoseFlippedX[3] := PoseFlippedX[3]+180
PoseFlippedX[4] := -PoseFlippedX[4]
PoseFlippedX[5] := -PoseFlippedX[5]
```

Note that if the rectangle is a square (Width == Height) the number of alternative poses is 8.

If more than one contour are given in Contour, a corresponding tuple of values for both Width and Height has to be provided as well. Yet, if only one value is provided for each of these arguments, then this value is applied for each processed contour. A pose is estimated for each processed contour and all poses are concatenated in Pose (see the example below).

Accuracy of the pose

The accuracy of the estimated pose depends on the following three factors:

- ratio Width/Height
- length of the projected contour

- degree of perspective distortion of the contour

In order to achieve an accurate pose estimation, there are three corresponding criteria that should be considered:

The **ratio** Width/Height should fulfill

$$\frac{1}{3} < \texttt{Width}/\texttt{Height} < 3$$

For a rectangular object deviating from this criterion, its longer side dominates the determination of its pose. This causes instability in the estimation of the angle around the longer rectangle's axis. In the extreme case when one of the dimensions is 0, the rectangle is in fact a line segment, whose pose cannot be estimated.

Secondly, the **lengths of each side of the contour** should be at least 20 pixels. An error is returned if a side of the contour is less than 5 pixels long.

Thirdly, the more the contour appears **projectively distorted**, the more stable the algorithm works. Therefore, the pose of a rectangle tilted with respect to the image plane can be estimated accurately, whereas the pose of an rectangle parallel to the image plane of the camera could be unstable. This is further discussed in the next paragraph. Additionally, there is a rule of thumb that ensures projective distortion: the rectangle should be placed in space such that its size in x and y dimension in the camera coordinate system should not be less than $1/10th$ of its distance from the camera in z direction.

get_rectangle_pose provides two measures for the accuracy of the estimated Pose. Error is the average pixel error between the contour points and the modeled rectangle backprojected on the image. If Error is exceeding 0.5, this is an indication that the algorithm did not converge properly, and the resulting Pose should not be used. CovPose contains 36 entries representing the 6×6 covariance matrix of the first 6 entries of Pose. The above mentioned case of instability of the angle about the longer rectangle's axis be detected by checking that the absolute values of the variances and covariances of the rotations around the x and y axis (CovPose[21],CovPose[28], and CovPose[22] == CovPose[27]) do not exceed 0.05. Further, unusually increased values of any of the covariances and especially of the variances (the 6 values on the diagonal of CovPose with indices 0, 7, 14, 21, 28 and 35, respectively) indicate a poor quality of Pose.

Parameters

- ▷ **Contour** (input_object) ..... xld(-array) ⇝ *object*
  Contour(s) to be examined.
- ▷ **CameraParam** (input_control) ..... campar ⇝ *real* / integer / string
  Internal camera parameters.
- ▷ **Width** (input_control) ..... number(-array) ⇝ *real*
  Width of the rectangle in meters.
  Restriction: Width > 0
- ▷ **Height** (input_control) ..... number(-array) ⇝ *real*
  Height of the rectangle in meters.
  Restriction: Height > 0
- ▷ **WeightingMode** (input_control) ..... string ⇝ *string*
  Weighting mode for the optimization phase.
  Default: 'nonweighted'
  List of values: WeightingMode ∈ {'nonweighted', 'huber', 'tukey'}
- ▷ **ClippingFactor** (input_control) ..... number ⇝ *real*
  Clipping factor for the elimination of outliers (typical: *1.0* for *'huber'* and *3.0* for *'tukey'*).
  Default: 2.0
  Suggested values: ClippingFactor ∈ {1.0, 1.5, 2.0, 2.5, 3.0}
  Restriction: ClippingFactor > 0
- ▷ **Pose** (output_control) ..... pose(-array) ⇝ *real* / integer
  3D pose of the rectangle.
  Number of elements: Pose == 7 * Contour
- ▷ **CovPose** (output_control) ..... number-array ⇝ *real*
  Covariances of the pose values.
  Number of elements: CovPose == 36 * Contour
- ▷ **Error** (output_control) ..... number-array ⇝ *real*
  Root-mean-square value of the final residual error.
  Number of elements: Error == Contour

Example

```
* Process an image with several rectangles of the same size appearing
* as light objects.
RectWidth := 0.04
RectHeight := 0.025
gen_cam_par_area_scan_division (0.014, 830, 4.58e-006, 4.65e-006, \
                                509, 488, 800, 800, CameraParam)
read_image (Image, 'barcode/ean13/tea_box_01')
* Find light objects in the image.
mean_image (Image, ImageMean, 201, 201)
dyn_threshold (Image, ImageMean, Region, 5, 'light')
* Fill gaps in the objects.
fill_up (Region, RegionFillUp)
* Extract rectangular contours.
connection (RegionFillUp, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, ['area','rectangularity'], \
              'and', [50000,0.9], [640000,1])
boundary (SelectedRegions, RegionBorder, 'inner')
dilation_circle (RegionBorder, RegionDilation, 3.5)
* NOTE: for a real application, this step might require some additional
*       pre- or postprocessing.
reduce_domain (Image, RegionDilation, ImageReduced)
edges_sub_pix (ImageReduced, Edges, 'canny', 1, 20, 40)
* Get the pose of all contours found.
get_rectangle_pose (Edges, CameraParam, RectWidth, RectHeight, 'huber', 2, \
                    Poses, CovPose, Error)
NumPoses := |Poses|/7
for I := 0 to NumPoses-1 by 1
  Pose := Poses[I*7:I*7+6]
  * Use the Pose here.
  * ...
endfor
```

Result

`get_rectangle_pose` returns `2` (`H_MSG_TRUE`) if all parameter values are correct and the position of the rectangle has been determined successfully. If the provided contour(s) cannot be segmented as a quadrangle `get_rectangle_pose` returns H_ERR_FIT_QUADRANGLE. If further necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`edges_sub_pix`

See also

`get_circle_pose`, `set_origin_pose`, `camera_calibration`

References

G.Schweighofer and A.Pinz: “Robust Pose Estimation from a Planar Target”; Transactions on Pattern Analysis and Machine Intelligence (PAMI), 28(12):2024-2030, 2006

Module

3D Metrology

pose_average (: : Poses, Weights, Mode, SigmaT, SigmaR : AveragePose, Quality)

Compute the average of a set of poses.

pose_average computes the average rotation and translation of the poses passed in Poses, and returns this average pose in AveragePose. The poses can be weighted using the parameter Weights. If an empty tuple is passed as weight, all poses have the same influence on the result. Otherwise, a tuple that contains one positive weight per pose needs to be passed. A higher weight leads to a higher influence of that pose on the average.

pose_average supports two modes for averaging the poses, which can be selected with the parameter Mode. *'direct'* selects a direct computation of the average, where the translation and rotation are averaged independently from each other. *'iterative'* uses the average pose computed by the *'direct'* method as initial value for an iterative method, which computes the local mode of the poses. The iterative method is slower than the direct method. However, it returns more accurate poses especially in the presence of outlier poses, as such outliers are downweighted automatically.

For the iterative method, the relative weight of the translation and the rotation can be set with the parameters SigmaT and SigmaR, respectively. Both values can be set to *'auto'*, in which case they are automatically estimated. In this case, SigmaT is set to the spread of the translations of the poses, and SigmaR is set to a constant value. Both values describe the expected spread of the translation and the rotation, and influence the weighting of the poses. For the direct method, both parameters are ignored.

A measure of the quality of the computed pose is returned in Quality. Quality contains a tuple with four elements, which describe the average and maximum deviation of the passed poses from the returned average pose AveragePose. The order of the values is: Root-Mean-Square error of the translation, Root-Mean-Square error of the rotation, maximum translation and maximum rotation deviation. The weights passed in Weights are used in the Root-Mean-Square errors of translation and rotation, but not for the last two quality measures.

Parameters

▷ **Poses** (input_control) . . . . . pose-array ⇝ *integer*
Set of poses of which the average if computed.

▷ **Weights** (input_control) . . . . . number-array ⇝ *real* / integer
Empty tuple, or one weight per pose.
Default: []
Restriction: Weights > 0 && length(Weights) == 0 || length(Weights) == length(Poses) / 7

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Averaging mode.
Default: 'iterative'
List of values: Mode ∈ {'direct', 'iterative'}

▷ **SigmaT** (input_control) . . . . . number ⇝ *real* / integer / string
Weight of the translation.
Default: 'auto'
Suggested values: SigmaT ∈ {'auto', 0.1, 1, 100}

▷ **SigmaR** (input_control) . . . . . number ⇝ *real* / integer / string
Weight of the rotation.
Default: 'auto'
Suggested values: SigmaR ∈ {'auto', 0.1, 1, 10}

▷ **AveragePose** (output_control) . . . . . pose ⇝ *real* / integer
Weighted mean of the poses.

▷ **Quality** (output_control) . . . . . number-array ⇝ *real*
Deviation of the mean from the input poses.
Assertion: length(Quality) == 4

Result

If the parameters are valid, the operator pose_average returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

read_pose, hom_mat3d_to_pose

See also

read_pose, hom_mat3d_to_pose, pose_to_hom_mat3d

Module

Foundation

pose_compose (: : PoseLeft, PoseRight : PoseCompose)

Combine 3D poses given in two tuples.

pose_compose combines the poses in the tuples, PoseLeft and PoseRight. If both tuples contain the same number of poses, the corresponding elements of both tuples are composed. Otherwise, either tuple PoseLeft or PoseRight must contain only one pose. In this case, the composition is performed for each pose of the longer tuple with the single pose of the other tuple. For each composition, the poses are interpreted as transformations of coordinate systems. The poses in tuple PoseCompose are therefore the result of applying the corresponding poses in PoseLeft and PoseRight in sequence.

First, the respective poses in PoseLeft and PoseRight are transformed into the corresponding homogeneous transformation matrices $\mathbf{H}_1$ and $\mathbf{H}_2$. $\mathbf{H}_1$ is then multiplied with $\mathbf{H}_2$. The resulting matrix $\mathbf{H}_1 \cdot \mathbf{H}_2$ is converted into a pose and returned at the corresponding index in tuple PoseCompose.

If the respective poses in PoseLeft and PoseRight have different types, the default pose type 0 ('Rp+T', 'gba', 'point') is returned. Otherwise, the returned poses have the same types as the poses used for their composition.

Parameters

▷ **PoseLeft** (input_control) . . . . . pose(-array) ⇝ *real* / integer
Tuple containing the left poses.

▷ **PoseRight** (input_control) . . . . . pose(-array) ⇝ *real* / integer
Tuple containing the right poses.

▷ **PoseCompose** (output_control) . . . . . pose(-array) ⇝ *real* / integer
Tuple containing the returned poses.

Result

pose_compose returns 2 (H_MSG_TRUE) if all parameters are valid. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_pose, hom_mat3d_to_pose, create_pose, convert_pose_type, pose_invert

Possible Successors

convert_pose_type

Alternatives

hom_mat3d_compose, dual_quat_compose

See also

pose_to_hom_mat3d, hom_mat3d_to_pose, hom_mat3d_compose

Module

Foundation

pose_invert (: : Pose : PoseInvert)

Invert each pose in a tuple of 3D poses.

pose_invert inverts each pose of the given tuple Pose by transforming it into the corresponding homogeneous transformation matrix $\mathbf{H}$ and inverting this matrix. The resulting matrix is converted into a pose. This pose is returned at the respective index in the tuple PoseInvert. The returned poses have the same types as the original poses.

Parameters

▷ **Pose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
Tuple of 3D poses.

▷ **PoseInvert** (output_control) . . . . . pose(-array) ⇝ *real* / integer
Tuple of inverted 3D poses.

Result

pose_invert returns 2 (H_MSG_TRUE) if all parameters are valid. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_pose, hom_mat3d_to_pose, create_pose, convert_pose_type, pose_compose

Possible Successors

convert_pose_type

Alternatives

hom_mat3d_invert, dual_quat_conjugate

See also

pose_to_hom_mat3d, hom_mat3d_to_pose

Module

Foundation

pose_to_dual_quat (: : Pose : DualQuaternion)

Convert a 3D pose to a unit dual quaternion.

The operator pose_to_dual_quat converts the input Pose into its corresponding unit dual quaternion DualQuaternion.

For a brief introduction to dual quaternions, the used notation, and the relationship between dual quaternions and screws, see "Solution Guide III-C - 3D Vision".

Because $\hat{q}$ and $-\hat{q}$ represent the same rigid transformation, the sign of the resulting dual quaternion DualQuaternion is chosen such that the scalar part of the real part is ≥ 0.

Multiple poses can be passed in Pose, in which case a unit dual quaternion is returned for each of them.

Parameters

▷ **Pose** (input_control) . . . . . pose(-array) ⇝ *real* / integer
3D pose.

▷ **DualQuaternion** (output_control) . . . . . dual_quaternion(-array) ⇝ *real*
Unit dual quaternion.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_pose, read_pose

Possible Successors

dual_quat_compose, dual_quat_interpolate, dual_quat_to_screw, dual_quat_conjugate

Alternatives

pose_to_hom_mat3d, screw_to_dual_quat

See also

dual_quat_to_hom_mat3d, dual_quat_to_pose, dual_quat_normalize, serialize_dual_quat, deserialize_dual_quat, dual_quat_trans_line_3d, dual_quat_trans_point_3d, pose_to_quat

Module

Foundation

pose_to_quat (: : Pose : Quaternion)

Convert the rotational part of a 3D pose to a quaternion.

The operator pose_to_quat converts the rotation of the input 3D pose Pose into its corresponding quaternion Quaternion. Note that the translation of Pose is ignored. Multiple poses can be passed in Pose, in which case a quaternion is returned for each of them.

Parameters

- ▷ **Pose** (input_control) ..... pose(-array) ⇝ *real* / integer
  3D Pose.
- ▷ **Quaternion** (output_control) ..... quaternion(-array) ⇝ *real*
  Rotation quaternion.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

create_pose, read_pose

Possible Successors

quat_compose, quat_rotate_point_3d, quat_interpolate

Alternatives

axis_angle_to_quat

See also

quat_to_pose, quat_to_hom_mat3d, quat_compose, quat_rotate_point_3d, pose_to_dual_quat

Module

Foundation

proj_hom_mat2d_to_pose (: : Homography, CameraMatrix, Method : Pose)

Compute a pose out of a homography describing the relation between world and image coordinates.

proj_hom_mat2d_to_pose computes a pose out of a homography Homography that describes the relation between 2D world- (unit meters) and 2D image coordinates. The homography can be obtained by the use of vector_to_proj_hom_mat2d. To compute a pose, a camera matrix (obtained by cam_par_to_cam_mat,

for example) has to be passed in CameraMatrix. By setting the Parameter Method, it is possible to choose which computation algorithm is used. Currently a direct decomposition (Method = 'decomposition') and a singular value decomposition based method (Method = 'decomposition_svd') are supported. In general due to numeric reasons, the direct decomposition method returns more stable results.

The subsequent calls of vector_to_proj_hom_mat2d and proj_hom_mat2d_to_pose are similar to vector_to_pose for 2D objects, whereas no refinement due to redundancy is possible.

Parameters

- ▷ **Homography** (input_control) . . . . . hom_mat2d ⇝ *real*
  The homography from world- to image coordinates.
- ▷ **CameraMatrix** (input_control) . . . . . hom_mat2d ⇝ *real*
  The camera calibration matrix K.
- ▷ **Method** (input_control) . . . . . string ⇝ *string*
  Type of pose computation.
  Default: 'decomposition'
  List of values: Method ∈ {'decomposition', 'decomposition_svd'}
- ▷ **Pose** (output_control) . . . . . pose ⇝ *real* / integer
  Pose of the 2D object.

Result

proj_hom_mat2d_to_pose returns 2 (H_MSG_TRUE) if all parameter values are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

image_points_to_world_plane, vector_to_proj_hom_mat2d

See also

vector_to_proj_hom_mat2d, image_points_to_world_plane, vector_to_pose, camera_calibration

Module

Calibration

quat_to_pose (: : Quaternion : Pose)

Convert a quaternion into the corresponding 3D pose.

The operator quat_to_pose converts the input quaternion Quaternion into its corresponding 3D pose Pose. Note that the translation of Pose is zero in any case.

Parameters

- ▷ **Quaternion** (input_control) . . . . . quaternion ⇝ *real*
  Rotation quaternion.
- ▷ **Pose** (output_control) . . . . . pose ⇝ *real* / integer
  3D Pose.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

axis_angle_to_quat, quat_compose, quat_conjugate

Possible Successors

camera_calibration, write_pose, disp_caltab, sim_caltab

See also

pose_to_quat, axis_angle_to_quat, quat_compose, quat_rotate_point_3d, quat_to_hom_mat3d, quat_normalize, dual_quat_to_pose

Module

Foundation

read_pose (: : PoseFile : Pose)

Read a 3D pose from a text file.

read_pose is used to read the 3D pose Pose from a text file with the name PoseFile. The default HALCON file extension for the 3D pose is 'dat'.

A pose describes a rigid 3D transformation, i.e., a transformation consisting of an arbitrary translation and rotation, with 6 parameters, three for the translation, three for the rotation. With a seventh parameter different pose types can be indicated (see create_pose).

A suitable file can be generated by the operator write_pose and looks like the following:

```
\# 3D POSE PARAMETERS: rotation and translation

\# Used representation type:
f 0

\# Rotation angles [deg] or Rodriguezvector:
r -17.8134 1.83816 0.288092

\# Translation vector (x y z [m]):
t 0.280164 0.150644 1.7554
```

Parameters

▷ **PoseFile** (input_control) . . . . . filename.read ⇝ *string*
File name of the external camera parameters.
Default: 'campose.dat'
Suggested values: PoseFile ∈ {'campose.dat', 'campose_initial.dat', 'campose_final.dat'}
File extension: .dat

▷ **Pose** (output_control) . . . . . pose ⇝ *real* / integer
3D pose.
Number of elements: 7

Result

read_pose returns 2 (H_MSG_TRUE) if all parameter values are correct and the file has been read successfully. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_cam_par

Possible Successors

pose_to_hom_mat3d, camera_calibration, disp_caltab, sim_caltab

See also

create_pose, find_marks_and_pose, camera_calibration, disp_caltab, sim_caltab, write_pose, pose_to_hom_mat3d, hom_mat3d_to_pose

Module

Foundation

serialize_pose (: : Pose : SerializedItemHandle)

Serialize a pose.

serialize_pose serializes the data of a pose (see fwrite_serialized_item for an introduction of the basic principle of serialization). The same data that is written in a file by write_pose is converted to a serialized item. The pose is defined by the handle Pose. The serialized pose is returned by the handle SerializedItemHandle and can be deserialized by deserialize_pose.

Parameters

▷ **Pose** (input_control) ..... pose ⇝ *real / integer*
3D pose.
Number of elements: 7

▷ **SerializedItemHandle** (output_control) ..... serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_pose returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_pose

Module

Foundation

set_origin_pose (: : PoseIn, DX, DY, DZ : PoseNewOrigin)

Translate the origin of a 3D pose.

set_origin_pose translates the origin of the 3D pose PoseIn by the vector given by DX, DY, and DZ and returns the result in PoseNewOrigin. Note that the translation is performed relative to the local coordinate system of the pose itself. For example, if PoseIn describes the pose of an object in camera coordinates, PoseNewOrigin is obtained by translating the object's coordinate system by DX along its own x-axis (and so on for the other axes) and not along the x-axis of the camera coordinate system. This corresponds to the following chain of transformations:

$$\mathtt{PoseNewOrigin} = \mathtt{PoseIn} \cdot \begin{bmatrix} 1 & 0 & 0 & \begin{pmatrix} \mathtt{DX} \\ \mathtt{DY} \\ \mathtt{DZ} \end{pmatrix} \\ 0 & 1 & 0 & \\ 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus, set_origin_pose is a shortcut for the following sequence of operator calls:

```
pose_to_hom_mat3d(PoseIn, HomMat3DIn)
hom_mat3d_translate_local(HomMat3DIn, DX, DY, DZ,
HomMat3DNewOrigin)
hom_mat3d_to_pose(HomMat3DNewOrigin, PoseNewOrigin)
```

A typical application of this operator when defining a world coordinate system by placing the standard calibration plate on the plane of measurements. In this case, the external camera parameters returned by `camera_calibration` correspond to a coordinate system that lies above the measurement plane, because the coordinate system of the calibration plate is located on its surface and the plate has a certain thickness. To correct the pose, call `set_origin_pose` with the translation vector (0,0,D), where D is the thickness of the calibration plate.

Parameters

- ▷ **`PoseIn`** (input_control) . . . . . pose ⇝ *real* / integer
  original 3D pose.
  Number of elements: 7
- ▷ **`DX`** (input_control) . . . . . real ⇝ *real*
  translation of the origin in x-direction.
  Default: 0
- ▷ **`DY`** (input_control) . . . . . real ⇝ *real*
  translation of the origin in y-direction.
  Default: 0
- ▷ **`DZ`** (input_control) . . . . . real ⇝ *real*
  translation of the origin in z-direction.
  Default: 0
- ▷ **`PoseNewOrigin`** (output_control) . . . . . pose ⇝ *real* / integer
  new 3D pose after applying the translation.
  Number of elements: 7

Result

`set_origin_pose` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_pose`, `hom_mat3d_to_pose`, `camera_calibration`, `hand_eye_calibration`

Possible Successors

`write_pose`, `pose_to_hom_mat3d`, `image_points_to_world_plane`, `contour_to_world_plane_xld`

See also

`hom_mat3d_translate_local`

Module

Foundation

vector_to_pose (: : WorldX, WorldY, WorldZ, ImageRow, ImageColumn, CameraParam, Method, QualityType : Pose, Quality)

Compute an absolute pose out of point correspondences between world and image coordinates.

The operator `vector_to_pose` computes a pose out of at least three or four (depending on `Method`) point correspondences of 3D world coordinates (`WorldX`, `WorldY`, `WorldZ`), given in meters, and 2D image coordinates (`ImageRow`, `ImageColumn`), given in pixels, as well as the internal camera parameters (`CameraParam`).

Thereby the pose (the external camera parameters) is in the form ${}^{ccs}\mathbf{P}_{wcs}$, where *ccs* denotes the camera coordinate system and *wcs* the world coordinate system (see Transformations / Poses and "Solution Guide III-C - 3D Vision").

Parameter Method

By setting the parameter Method, it is possible to choose what kind of algorithm is used for the pose computation.

Methods supported for **perspective area scan cameras**:

Method	*When to use*	*Minimum number of point correspondences*
'analytic' [1]	Default method for general cases	4
'iterative' [2]	If only three or four point correspondences are used or if the world points are close to being planar	3
'planar_analytic' [4]	If the world points lie in a horizontal plane (WorldZ = 0)	4

The numbers in square brackets in the table above refer to the publications the implementations of the corresponding methods are based on.

Methods supported for **telecentric area or line scan cameras**:

Method	*When to use*	*Minimum number of point correspondences*
'telecentric' [3]	Default method for general cases	4
'telecentric_robust' [3]	For very ill-posed point configurations where Quality has an unlikely large value	4
'telecentric_planar' [3]	If the world points lie in a horizontal plane (WorldZ = 0)	3
'telecentric_planar_robust' [3]	For very ill-posed point configurations where the world points lie in a horizontal plane (WorldZ = 0) and Quality has an unlikely large value	3

The numbers in square brackets in the table above refer to the publications the implementations of the corresponding methods are based on.

Methods supported for **perspective line scan cameras**:

Method	*When to use*	*Minimum number of point correspondences*
'line_scan'	Default method for general cases	3

Parameters CameraParam and Quality

All methods need the inner camera parameters obtained from camera_calibration to solve the pose estimation problem. They must be passed in CameraParam.

The user can specify in QualityType one or more quality measures of the pose to be evaluated. The resulting quality evaluations are returned concatenated in Quality. Currently, only *'error'* is supported. It corresponds to the root-mean-square error in pixels of the projected 3D world coordinates.

General Remarks

If a method for planar world points is chosen, all world points are assumed to lie in the plane WorldZ = 0. Therefore, the z-component of the world coordinates can be left empty (WorldZ = *[]*) since only 2D correspondences are used in this case.

For telecentric cameras, the translation in *z* obviously cannot be determined. It is set to *0* in Pose.

For planar world points and telecentric cameras, there are always two possible equivalent poses. This ambiguity can only be resolved by some additional knowledge. `vector_to_pose` returns an arbitrary solution of the two possible solutions in this case. The other solution can be calculated easily by replacing the values of α and β in `Pose` by $360 - \alpha$ and $360 - \beta$.

For perspective line scan cameras, the call to `vector_to_pose` corresponds to the following operator sequence:

```
get_line_of_sight(ImageRow, ImageColumn, CameraParam, PX, PY, PZ,
QX, QY, QZ)
points_to_pluecker_line(PX, PY, PZ, QX, QY, QZ, LX, LY, LZ, MX,
MY, MZ)
point_pluecker_line_to_hom_mat3d('rigid', WorldX, WorldY, WorldZ,
LX, LY, LZ, MX, MY, MZ, HomMat3D)
hom_mat3d_to_pose(HomMat3D, Pose)
```

In contrast to the methods for other camera types, an error in 3D space is minimized in this case (see `point_pluecker_line_to_hom_mat3d`). However, like for all other methods, the quality evaluation is performed in pixels. If it is essential that a pixel error is optimized, this can be done as shown in the example below.

Attention

The method *'analytic'* only allows a maximum number of 32767 point correspondences.

Parameters

- ▷ **`WorldX`** (input_control) ..... number-array ⇝ *real* / integer
  X-Component of world coordinates.
  Number of elements: WorldX >= 4
- ▷ **`WorldY`** (input_control) ..... number-array ⇝ *real* / integer
  Y-Component of world coordinates.
  Number of elements: WorldY == WorldX
- ▷ **`WorldZ`** (input_control) ..... number-array ⇝ *real* / integer
  Z-Component of world coordinates.
  Number of elements: WorldZ == WorldX || WorldZ == 0
- ▷ **`ImageRow`** (input_control) ..... number-array ⇝ *real* / integer
  Row-Component of image coordinates.
  Number of elements: ImageRow == WorldX
- ▷ **`ImageColumn`** (input_control) ..... number-array ⇝ *real* / integer
  Column-Component of image coordinates.
  Number of elements: ImageColumn == WorldX
- ▷ **`CameraParam`** (input_control) ..... campar ⇝ *real* / integer / string
  The inner camera parameters from camera calibration.
- ▷ **`Method`** (input_control) ..... string ⇝ *string*
  Kind of algorithm
  Default: 'iterative'
  List of values: `Method` ∈ {'iterative', 'analytic', 'planar_analytic', 'telecentric', 'telecentric_robust', 'telecentric_planar', 'telecentric_planar_robust', 'line_scan'}
- ▷ **`QualityType`** (input_control) ..... string(-array) ⇝ *string*
  Type of pose quality to be returned in Quality.
  Default: 'error'
  List of values: `QualityType` ∈ {'error'}
- ▷ **`Pose`** (output_control) ..... pose ⇝ *real* / integer
  Pose.
- ▷ **`Quality`** (output_control) ..... number(-array) ⇝ *real* / integer
  Pose quality.

Example

```
* Optimize the pixel error line scan cameras. We assume that the
```

```
* point correspondences (Row, Col) - (PX, PY, PZ) have already been
* computed and the internal camera parameters CamParam have already
* been calibrated.
vector_to_pose (PX, PY, PZ, Row, Col, CamParam, 'line_scan', \
                'error', PoseObjectSpace, RMS)
* Use PoseObjectSpace as the starting value for a camera calibration
* that only optimizes the pose.
create_calib_data ('calibration_object', 1, 1, CalibDataID)
set_calib_data_cam_param (CalibDataID, 0, [], CamParam)
set_calib_data_calib_object (CalibDataID, 0, [PX, PY, PZ])
set_calib_data (CalibDataID, 'camera', 0, \
                'excluded_settings', 'params')
set_calib_data_observ_points (CalibDataID, 0, 0, 0, Row, Col, \
                              'all', PoseObjectSpace)
calibrate_cameras (CalibDataID, Error)
get_calib_data (CalibDataID, 'calib_obj_pose', [0, 0], 'pose', Pose)
clear_calib_data (CalibDataID)
* As an alternative, the following one-line code can be used.
camera_calibration (PX, PY, PZ, Row, Col, CamParam, PoseObjectSpace, \
                    'pose', _, Pose, Error)
```

Result

`vector_to_pose` returns `2` (`H_MSG_TRUE`) if all parameter values are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`proj_hom_mat2d_to_pose`, `vector_to_rel_pose`, `camera_calibration`

References

[1] Francesc Moreno-Noguer, Vincent Lepetit, and Pascal Fua: “Accurate Non-Iterative O(n) Solution to the PnP Problem”; Eleventh IEEE International Conference on Computer Vision, 2007.

[2] Gerald Schweighofer, and Axel Pinz: “Robust Pose Estimation from a Planar Target”; Transactions on Pattern Analysis and Machine Intelligence (PAMI), 28(12):2024-2030, 2006.

[3] Carsten Steger: “Algorithms for the Orthographic-*n*-Point Problem”; Journal of Mathematical Imaging and Vision, vol. 60, no. 2, pp. 246-266, 2018.

[4] Zhengyou Zhang: “A flexible new technique for camera calibration.”; Transactions on Pattern Analysis and Machine Intelligence (PAMI), 22(11):1330-1334, 2000.

Module

Calibration

write_pose (: : Pose, PoseFile :)

Write a 3D pose to a text file.

`write_pose` is used to write a 3D pose `Pose` into a text file with the name `PoseFile`. The default HALCON file extension for the 3D pose is ’dat’.

A pose describes a rigid 3D transformation, i.e., a transformation consisting of an arbitrary translation and rotation, with 6 parameters, three for the translation, three for the rotation. With a seventh parameter different pose types can be indicated (see `create_pose`).

A file generated by `write_pose` looks like the following:

```
\# 3D POSE PARAMETERS: rotation and translation

\# Used representation type:
f 0

\# Rotation angles [deg] or Rodriguez vector:
r -17.8134 1.83816 0.288092

\# Translation vector (x y z [m]):
t 0.280164 0.150644 1.7554
```

Parameters

▷ **Pose** (input_control) . . . . . pose ⇝ *real* / integer
3D pose.
Number of elements: 7

▷ **PoseFile** (input_control) . . . . . filename.write ⇝ *string*
File name of the external camera parameters.
Default: 'campose.dat'
Suggested values: PoseFile ∈ {'campose.dat', 'campose_initial.dat', 'campose_final.dat'}
File extension: .dat

Example

```
* Perform hand-eye calibration.
calibrate_hand_eye (CalibDataID, Errors)
* Query the camera parameters and the poses.
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
* Get poses computed by the hand eye calibration.
get_calib_data (CalibDataID, 'camera', 0, 'base_in_cam_pose', BaseInCamPose)
get_calib_data (CalibDataID, 'calib_obj', 0,\
                'obj_in_tool_pose', ObjInToolPose)
* Store the camera parameters to file.
write_cam_par (CamParam, DataNameStart + 'final_campar.dat')
* Save the hand eye calibration results to file.
write_pose (BaseInCamPose, DataNameStart + 'final_pose_cam_base.dat')
write_pose (ObjInToolPose, DataNameStart + 'final_pose_tool_calplate.dat')
```

Result

write_pose returns 2 (H_MSG_TRUE) if all parameter values are correct and the file has been written successfully. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

camera_calibration, get_calib_data, hom_mat3d_to_pose

See also

create_pose, find_marks_and_pose, camera_calibration, disp_caltab, sim_caltab, read_pose, pose_to_hom_mat3d, hom_mat3d_to_pose

Module

Foundation

27.6 Quaternions

axis_angle_to_quat (: : AxisX, AxisY, AxisZ, Angle : Quaternion)

Create a rotation quaternion.

Quaternions are an extension of the complex numbers. The set of quaternions is given by $\{x_0 + x_1 * i + x_2 * j + x_3 * k | x_0, x_1, x_2, x_3 \in R\}$. In HALCON, quaternions are represented by a tuple containing four elements: $[x_0, x_1, x_2, x_3]$.

Unit quaternions can be used to describe rotations. A quaternion is a unit quaternion if its norm is 1. A counter-clockwise rotation around an unit vector v (AxisX, AxisY, AxisZ) by an angle a (Angle) can be described by the unit quaternion $q = (cos(a/2), sin(a/2)v)$.

The operator axis_angle_to_quat can be used to create such a quaternion. To rotate a point using Quaternion use the operator quat_rotate_point_3d.

Note that two rotations can be concatenated using the operator quat_compose. Further, you can use a rotation quaternion to set the orientation of the 3D plot in set_paint.

Attention

The operator axis_angle_to_quat does not check whether the vector (AxisX, AxisY, AxisZ) is of unit length (i.e. of length 1). If this is not the case, Quaternion will be no valid rotation quaternion.

Parameters

- ▷ **AxisX** (input_control) ..... real ⇝ *real* / integer
  X component of the rotation axis.
- ▷ **AxisY** (input_control) ..... real ⇝ *real* / integer
  Y component of the rotation axis.
- ▷ **AxisZ** (input_control) ..... real ⇝ *real* / integer
  Z component of the rotation axis.
- ▷ **Angle** (input_control) ..... real ⇝ *real* / integer
  Rotation angle in radians.
- ▷ **Quaternion** (output_control) ..... quaternion ⇝ *real* / integer
  Rotation quaternion.

Example

```
** Normalize a vector and create a rotation quaternion
Length := sqrt(AxisX*AxisX+AxisY*AxisY+AxisZ*AxisZ)
AxisX := AxisX/Length
AxisY := AxisY/Length
AxisZ := AxisZ/Length
axis_angle_to_quat (AxisX, AxisY, AxisZ, rad(90), Quaternion)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

quat_to_hom_mat3d, quat_to_pose, quat_rotate_point_3d, quat_compose

See also

quat_normalize, quat_conjugate, quat_interpolate, screw_to_dual_quat

Module

Foundation

deserialize_quat (: : SerializedItemHandle : Quaternion)

Deserialize a serialized quaternion.

deserialize_quat deserializes a quaternion, that was serialized by serialize_quat (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized quaternion is defined by the handle SerializedItemHandle. The deserialized values are stored in Quaternion.

Parameters

▷ **SerializedItemHandle** (input_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

▷ **Quaternion** (output_control) . . . . . quaternion ⇝ *real* / integer
Quaternion.

Result

If the parameters are valid, the operator deserialize_quat returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fread_serialized_item, receive_serialized_item, serialize_quat

Possible Successors

quat_to_hom_mat3d, quat_to_pose, quat_rotate_point_3d, quat_compose

Module

Foundation

quat_compose (: : QuaternionLeft,
QuaternionRight : QuaternionComposed)

Multiply two quaternions.

The operator quat_compose multiplies the two quaternions QuaternionLeft and QuaternionRight and returns the result in QuaternionComposed.

A quaternion x is given by $x = x_0 + x_1 i + x_2 j + x_3 k$. In HALCON, a quaternion is represented by a four value tuple: $[x_0, x_1, x_2, x_3]$.

The product of two quaternions x and y is defined as:

$$\begin{aligned} xy \quad = \quad & (x_0y_0 - x_1y_1 - x_2y_2 - x_3y_3) + \\ & (x_0y_1 + x_1y_0 + x_2y_3 - x_3y_2)i + \\ & (x_0y_2 - x_1y_3 + x_2y_0 + x_3y_1)j + \\ & (x_0y_3 + x_1y_2 - x_2y_1 + x_3y_0)k \end{aligned}$$

As a consequence, the multiplication of two quaternions is not commutative, i.e. $xy \neq yx$.

Parameters

▷ **QuaternionLeft** (input_control) . . . . . quaternion ⇝ *real*
Left quaternion.

▷ **QuaternionRight** (input_control) . . . . . quaternion ⇝ *real*
Right quaternion.

▷ **QuaternionComposed** (output_control) . . . . . quaternion ⇝ *real*
Product of the input quaternions.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

axis_angle_to_quat

Possible Successors

quat_to_hom_mat3d, quat_to_pose, quat_rotate_point_3d

See also

axis_angle_to_quat, quat_to_hom_mat3d, quat_rotate_point_3d, quat_normalize, quat_conjugate, dual_quat_compose

Module

Foundation

quat_conjugate (: : Quaternion : ConjugatedQuaternion)

Generate the conjugation of a quaternion.

The operator quat_conjugate generates the conjugation ConjugatedQuaternion of the input quaternion Quaternion.

The conjugation of a quaternion $q = x_0 + x_1 i + x_2 j + x_3 k$ is given by $\bar{q} = x_0 - x_1 i - x_2 j - x_3 k$.

Parameters

▷ **Quaternion** (input_control) ..... quaternion ⇝ *real*
Input quaternion.

▷ **ConjugatedQuaternion** (output_control) ..... quaternion ⇝ *real*
Conjugated quaternion.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

axis_angle_to_quat

Possible Successors

quat_to_hom_mat3d, quat_rotate_point_3d

See also

axis_angle_to_quat, quat_to_hom_mat3d, quat_rotate_point_3d, quat_normalize, quat_compose, quat_interpolate, dual_quat_conjugate

Module

Foundation

quat_interpolate (: : QuaternionStart, QuaternionEnd, InterpPos : QuaternionInterpolated)

Interpolation of two quaternions.

The operator quat_interpolate interpolates the two quaternions QuaternionStart and QuaternionEnd at the interpolation position InterpPos. This position must lie within the interval $[0, 1]$. In the case of InterpPos $= 0$, the interpolated quaternion QuaternionInterpolated

corresponds to QuaternionStart. In the case of InterpPos = 1, the interpolated quaternion QuaternionInterpolated corresponds to QuaternionEnd.

The interpolation is done using spherical linear interpolation. As a consequence if both QuaternionStart and QuaternionEnd are rotation quaternions, QuaternionInterpolated will be a rotation quaternion as well. Further, if InterpPos is increased at constant speed, a point on the unit sphere that is rotated using QuaternionInterpolated travels with constant speed on an arc on the unit sphere.

Parameters

- ▷ **QuaternionStart** (input_control) . . . quaternion ⇝ *real*
  Start quaternion.
- ▷ **QuaternionEnd** (input_control) . . . quaternion ⇝ *real*
  End quaternion.
- ▷ **InterpPos** (input_control) . . . real-array ⇝ *real*
  Interpolation parameter.
  Default: 0.5
  Suggested values: InterpPos ∈ {0.0, 0.25, 0.5, 0.75, 1.0}
- ▷ **QuaternionInterpolated** (output_control) . . . quaternion ⇝ *real*
  Interpolated quaternion.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

axis_angle_to_quat, quat_normalize

Possible Successors

quat_to_hom_mat3d, quat_rotate_point_3d, quat_to_pose

See also

quat_rotate_point_3d, quat_conjugate, quat_compose, dual_quat_interpolate

Module

Foundation

quat_normalize (: : Quaternion : NormalizedQuaternion)

Normalize a quaternion.

The operator quat_normalize scales the input quaternion Quaternion such that its norm is 1. The result is returned in NormalizedQuaternion.

Parameters

- ▷ **Quaternion** (input_control) . . . quaternion ⇝ *real*
  Input quaternion.
- ▷ **NormalizedQuaternion** (output_control) . . . quaternion ⇝ *real*
  Normalized quaternion.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

axis_angle_to_quat

Possible Successors

quat_to_hom_mat3d, quat_rotate_point_3d

See also

axis_angle_to_quat, quat_interpolate, quat_compose, quat_conjugate, dual_quat_to_hom_mat3d

Module

Foundation

quat_rotate_point_3d (: : Quaternion, Px, Py, Pz : Qx, Qy, Qz)

Perform a rotation by a unit quaternion.

Given a rotation quaternion (e.g., generated by axis_angle_to_quat), the operator quat_rotate_point_3d can be used to rotate a 3D point (Px, Py, Pz). The rotated point is returned in (Qx, Qy, Qz).

The rotation of a point x by an unit quaternion q is given by

$x' = qx\bar{q}$.

$\bar{q}$ corresponds to the conjugation of a quaternion q (see quat_conjugate). Note that x will be transformed into quaternion form to perform the rotation. This is done by setting the real part to zero and the three imaginary components to the three components of x.

Attention

The operator quat_rotate_point_3d does not check whether Quaternion is a unit quaternion. If Quaternion is not a unit quaternion, the result of this operator is not defined.

Parameters

- ▷ **Quaternion** (input_control) . . . quaternion ⇝ *real*
  Rotation quaternion.
- ▷ **Px** (input_control) . . . real ⇝ *real*
  X coordinate of the point to be rotated.
- ▷ **Py** (input_control) . . . real ⇝ *real*
  Y coordinate of the point to be rotated.
- ▷ **Pz** (input_control) . . . real ⇝ *real*
  Z coordinate of the point to be rotated.
- ▷ **Qx** (output_control) . . . real ⇝ *real*
  X coordinate of the rotated point.
- ▷ **Qy** (output_control) . . . real ⇝ *real*
  Y coordinate of the rotated point.
- ▷ **Qz** (output_control) . . . real ⇝ *real*
  Z coordinate of the rotated point.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

axis_angle_to_quat

See also

axis_angle_to_quat, quat_to_hom_mat3d, quat_compose, quat_normalize, quat_conjugate, quat_interpolate, dual_quat_trans_point_3d, dual_quat_trans_line_3d

Module

Foundation

quat_to_hom_mat3d (: : Quaternion : RotationMatrix)

Convert a quaternion into the corresponding rotation matrix.

The operator quat_to_hom_mat3d converts a unit quaternion Quaternion into its corresponding rotation matrix RotationMatrix.

The RotationMatrix for a quaternion $x = x_0 + x_1 i + x_2 j + x_3 k$ is given by:

$$\begin{pmatrix} x_0^2 + x_1^2 - x_2^2 - x_3^2 & 2x_1x_2 - 2x_0x_3 & 2x_0x_2 + 2x_1x_3 \\ 2x_0x_3 + 2x_1x_2 & x_0^2 - x_1^2 + x_2^2 - x_3^2 & 2x_2x_3 - 2x_0x_1 \\ 2x_1x_3 - 2x_0x_2 & 2x_0x_1 + 2x_2x_3 & x_0^2 - x_1^2 - x_2^2 + x_3^2 \end{pmatrix}$$

Attention

RotationMatrix will only be a valid rotation matrix if Quaternion is a unit quaternion.

Parameters

▷ **Quaternion** (input_control) . . . . . quaternion ⇝ *real*
Rotation quaternion.

▷ **RotationMatrix** (output_control) . . . . . hom_mat3d ⇝ *real*
Rotation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

axis_angle_to_quat, quat_compose

Possible Successors

affine_trans_point_3d

See also

axis_angle_to_quat, quat_rotate_point_3d, quat_to_pose, quat_normalize, quat_conjugate, quat_interpolate

Module

Foundation

serialize_quat (: : Quaternion : SerializedItemHandle)

Serialize a quaternion.

serialize_quat serializes the data of the Quaternion (see fwrite_serialized_item for an introduction of the basic principle of serialization). The serialized quaternion is returned by the handle SerializedItemHandle and can be deserialized by deserialize_quat.

Parameters

▷ **Quaternion** (input_control) . . . . . quaternion ⇝ *real* / integer
Quaternion.

▷ **SerializedItemHandle** (output_control) . . . . . serialized_item ⇝ *handle*
Handle of the serialized item.

Result

If the parameters are valid, the operator serialize_quat returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

axis_angle_to_quat

Possible Successors

fwrite_serialized_item, send_serialized_item, deserialize_quat

Module

Foundation

Chapter 28

Tuple

To learn more about the basic concept of tuples in HALCON you can also see the example program `halcon_basic_concepts.hdev`.

28.1 Arithmetic

tuple_abs (: : T : Abs)

Compute the absolute value of a tuple.

`tuple_abs` computes the absolute value of the input tuple `T`. The absolute value of an integer number is again an integer number. The absolute value of a floating point number is a floating point number. The absolute value of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_abs`, which can be used in an expression in the following syntax:

```
Abs := abs(T)
```

Parameters

- ▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input tuple.
- ▷ **Abs** (output_control) . . . . . number(-array) ⇝ *real* / integer
  Absolute value of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_fabs`

Module

Foundation

tuple_acos (: : T : ACos)

Compute the arccosine of a tuple.

tuple_acos computes the arccosine of the input tuple T. The arccosine is always returned as a floating point number in ACos. The angles in ACos are represented in radians. The arccosine of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_acos, which can be used in an expression in the following syntax:

```
ACos := acos(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.
Restriction: -1 <= T <= 1

▷ **ACos** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Arccosine of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_asin, tuple_atan, tuple_atan2

See also

tuple_cos, tuple_cosh, tuple_acosh

Module

Foundation

tuple_acosh (: : T : Acosh)

Compute the inverse hyperbolic cosine of a tuple.

tuple_acosh computes the inverse hyperbolic cosine of the input tuple T. The inverse hyperbolic cosine is always returned as a floating point number in Acosh. The inverse hyperbolic cosine of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_acosh, which can be used in an expression in the following syntax:

```
Acosh := acosh(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Acosh** (output_control) . . . . . number(-array) ⇝ *real*
Inverse hyperbolic cosine of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_asinh, tuple_atanh

See also

tuple_cosh, tuple_cos, tuple_acos

Module

Foundation

tuple_add (: : S1, S2 : Sum)

Add two tuples.

tuple_add computes the sum of the input tuples S1 and S2. If both tuples have the same length the corresponding elements of both tuples are added. Otherwise, either S1 or S2 must have length 1. In this case, the addition is performed for each element of the longer tuple with the single element of the other tuple. If two integer numbers are added, the result is again an integer number. If a floating point number is added to another number, the result is a floating point number. If two strings are added, the addition corresponds to a string concatenation. If a number and a string are added, the number is converted to a string first. Thus, the addition also corresponds to a string concatenation in this case.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_add, which can be used in an expression in the following syntax:

```
Sum := S1 + S2
```

The + operation corresponds to a string concatenation if S1 and S2 contain strings.

Parameters

- ▷ **S1** (input_control) ..... tuple(-array) ⇝ *real* / integer / string
  Input tuple 1.
- ▷ **S2** (input_control) ..... tuple(-array) ⇝ *real* / integer / string
  Input tuple 2.
- ▷ **Sum** (output_control) ..... tuple(-array) ⇝ *real* / integer / string
  Sum of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sub

See also

tuple_cumul

Module

Foundation

tuple_asin (: : T : ASin)

Compute the arcsine of a tuple.

tuple_asin computes the arcsine of the input tuple T. The arcsine is always returned as a floating point number in ASin. The angles in ASin are represented in radians. The arcsine of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_asin`, which can be used in an expression in the following syntax:

```
ASin := asin(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.
Restriction: -1 <= T <= 1

▷ **ASin** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Arcsine of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_acos`, `tuple_atan`, `tuple_atan2`

See also

`tuple_sin`, `tuple_sinh`, `tuple_asinh`

Module

Foundation

tuple_asinh (: : T : Asinh)

Compute the inverse hyperbolic sine of a tuple.

`tuple_asinh` computes the inverse hyperbolic sine of the input tuple `T`. The inverse hyperbolic sine is always returned as a floating point number in `Asinh`. The inverse hyperbolic sine of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_asinh`, which can be used in an expression in the following syntax:

```
Asinh := asinh(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Asinh** (output_control) . . . . . number(-array) ⇝ *real*
Inverse hyperbolic sine of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_acosh, tuple_atanh

See also

tuple_asin, tuple_sin, tuple_asin

Module

Foundation

tuple_atan (: : T : ATan)

Compute the arctangent of a tuple.

tuple_atan computes the arctangent of the input tuple T. The arctangent is always returned as a floating point number in ATan. The angles in ATan are represented in radians. The arctangent of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_atan, which can be used in an expression in the following syntax:

```
ATan := atan(T)
```

Parameters

- ▷ **T** (input_control) ..... number(-array) ⇝ *real* / integer
  Input tuple.
- ▷ **ATan** (output_control) ..... angle.rad(-array) ⇝ *real*
  Arctangent of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_atan2, tuple_asin, tuple_acos

See also

tuple_tan, tuple_tanh, tuple_atanh

Module

Foundation

tuple_atan2 (: : Y, X : ATan)

Compute the arctangent of a tuple for all four quadrants.

tuple_atan2 computes the arctangent of the input tuples Y/X while treating all four quadrants correctly. The arctangent is always returned as a floating point number in ATan. The angles in ATan are represented in radians. The arctangent of a string is not allowed.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_atan2, which can be used in an expression in the following syntax:

```
ATan := atan2(Y, X)
```

Parameters

▷ **Y** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple of the y-values.

▷ **X** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple of the x-values.

▷ **ATan** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Arctangent of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_atan, tuple_asin, tuple_acos

See also

tuple_tan, tuple_tanh

Module

Foundation

tuple_atanh (: : T : Atanh)

Compute the inverse hyperbolic tangent of a tuple.

tuple_atanh computes the inverse hyperbolic tangent of the input tuple T. The inverse hyperbolic tangent is always returned as a floating point number in Atanh. The inverse hyperbolic tangent of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_atanh, which can be used in an expression in the following syntax:

```
Atanh := atanh(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Atanh** (output_control) . . . . . number(-array) ⇝ *real*
Inverse hyperbolic tangent of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_asinh, tuple_acosh

See also

tuple_tanh, tuple_tan, tuple_atan, tuple_atan2

Module

Foundation

tuple_cbrt (: : T : Cbrt)

Compute the cube root of a tuple.

tuple_cbrt computes the cube root of the input tuple T. The cube root is always returned as a floating point number. The cube root of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_cbrt, which can be used in an expression in the following syntax:

```
Cbrt := cbrt(T)
```

Parameters

- ▷ **T** (input_control) . . . number(-array) ⇝ *real* / integer
  Input tuple.
- ▷ **Cbrt** (output_control) . . . number(-array) ⇝ *real*
  Cube root of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_pow

See also

tuple_sqrt

Module

Foundation

tuple_ceil (: : T : Ceil)

Compute the ceiling function of a tuple.

tuple_ceil computes the ceiling function of the input tuple T, i.e., the smallest integer greater than or equal to T. The ceiling function is always returned as a floating point number. The ceiling function of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_ceil, which can be used in an expression in the following syntax:

```
Ceil := ceil(T)
```

Parameters

- ▷ **T** (input_control) . . . number(-array) ⇝ *real* / integer
  Input tuple.
- ▷ **Ceil** (output_control) . . . number(-array) ⇝ *real*
  Ceiling function of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

tuple_cos (: : T : Cos)

Compute the cosine of a tuple.

tuple_cos computes the cosine of the input tuple T. The cosine is always returned as a floating point number in Cos. The angles in T are represented in radians. The cosine of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_cos, which can be used in an expression in the following syntax:

```
Cos := cos(T)
```

Parameters

▷ **T** (input_control) ..... angle.rad(-array) ⇝ *real* / integer
Input tuple.

▷ **Cos** (output_control) ..... number(-array) ⇝ *real*
Cosine of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sin, tuple_tan

See also

tuple_acos, tuple_cosh, tuple_acosh

Module

Foundation

tuple_cosh (: : T : Cosh)

Compute the hyperbolic cosine of a tuple.

tuple_cosh computes the hyperbolic cosine of the input tuple T. The hyperbolic cosine is always returned as a floating point number in Cosh. The hyperbolic cosine of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_cosh, which can be used in an expression in the following syntax:

```
Cosh := cosh(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Cosh** (output_control) . . . . . number(-array) ⇝ *real*
Hyperbolic cosine of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sinh, tuple_tanh

See also

tuple_acosh, tuple_cos, tuple_acos

Module

Foundation

tuple_cumul (: : Tuple : Cumul)

Calculate the cumulative sums of a tuple.

tuple_cumul returns the different cumulative sums of the corresponding elements of the input tuple Tuple, i.e. the i-th element of the result tuple Cumul is the sum of the first i elements of the input tuple Tuple. For example, if Tuple contains the values [1,2,3,4], the output tuple Cumul will contain the values [1,3,6,10]. All elements of Tuple must be numbers (integer or floating point numbers). Each element of the result tuple Cumul will contain a floating point number if at least one element of the calculation is a floating point number. If all elements of the calculation are integer numbers the resulting sum in Cumul will also be an integer number.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_cumul, which can be used in an expression in the following syntax:

```
Cumul := cumul(Tuple)
```

Parameters

▷ **Tuple** (input_control) . . . . . number(-array) ⇝ *integer* / real
Input tuple.

▷ **Cumul** (output_control) . . . . . number(-array) ⇝ *real* / integer
Cumulative sum of the corresponding tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sum

See also

tuple_min2, tuple_max2, tuple_add

Module

Foundation

tuple_deg (: : Rad : Deg)

Convert a tuple from radians to degrees.

tuple_deg converts the input tuple Rad from radians to degrees. The result is always returned as a floating point number. The conversion of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_deg, which can be used in an expression in the following syntax:

```
Deg := deg(Rad)
```

Parameters

▷ **Rad** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Deg** (output_control) . . . . . number(-array) ⇝ *real*
Input tuple in degrees.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

tuple_rad

Module

Foundation

tuple_div (: : Q1, Q2 : Quot)

Divide two tuples.

tuple_div computes the quotient of the input tuples Q1 and Q2. If both tuples have the same length the corresponding elements of both tuples are divided. Otherwise, either Q1 or Q2 must have length 1. In this case, the division is performed for each element of the longer tuple with the single element of the other tuple. If two integer numbers are divided, the result is again an integer number. If one of the operands is a floating point number, the result is a floating point number. The division of strings is not allowed.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_div, which can be used in an expression in the following syntax:

```
Quot := Q1 / Q2
```

Parameters

▷ **Q1** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple 1.

▷ **Q2** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple 2.
Restriction: Q2 != 0

▷ **Quot** (output_control) . . . . . number(-array) ⇝ *real* / integer
Quotient of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_mult

Module

Foundation

tuple_erf (: : T : Erf)

Compute the error function of a tuple.

tuple_erf computes the error function of the input tuple T. The error function is defined as:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

The value of the error function is always returned as a floating point number in Erf. The error function of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_erf, which can be used in an expression in the following syntax:

```
Erf := erf(T)
```

Parameters

▷ **T** (input_control) ..... number(-array) ⇝ *real* / integer
Input tuple.

▷ **Erf** (output_control) ..... number(-array) ⇝ *real*
Value of the error function of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_erfc

See also

tuple_exp

Module

Foundation

tuple_erfc (: : T : Erfc)

Compute the complementary error function of a tuple.

tuple_erfc computes the complementary error function of the input tuple T. The complementary error function is defined as $1 - \mathrm{erf}(x)$ (for the definition of $\mathrm{erf}(x)$ see tuple_erf). The value of the complementary error function is always returned as a floating point number in Erfc. The complementary error function of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_erfc, which can be used in an expression in the following syntax:

```
Erfc := erfc(T)
```

Parameters

▷ **T** (input_control) . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Erfc** (output_control) . . . number(-array) ⇝ *real*
Value of the complementary error function of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_erf

See also

tuple_exp

Module

Foundation

tuple_exp (: : T : Exp)

Compute the exponential of a tuple.

tuple_exp computes the exponential of the input tuple T. The exponential is always returned as a floating point number. The exponential of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_exp, which can be used in an expression in the following syntax:

```
Exp := exp(T)
```

Parameters

▷ **T** (input_control) . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Exp** (output_control) . . . number(-array) ⇝ *real*
Exponential of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Alternatives

tuple_exp2, tuple_exp10, tuple_pow

See also

tuple_log, tuple_log2, tuple_log10

Module

Foundation

tuple_exp10 (: : T : Exp)

Compute the base 10 exponential of a tuple.

tuple_exp10 computes the base 10 exponential of the input tuple T. The exponential is always returned as a floating point number. The exponential of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_exp10, which can be used in an expression in the following syntax:

```
Exp := exp10(T)
```

Parameters

▷ **T** (input_control) ..... number(-array) ⇝ *real* / integer
Input tuple.

▷ **Exp** (output_control) ..... number(-array) ⇝ *real*
Base 10 exponential of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_exp, tuple_exp2, tuple_pow

See also

tuple_log, tuple_log2, tuple_log10

Module

Foundation

tuple_exp2 (: : T : Exp)

Compute the base 2 exponential of a tuple.

tuple_exp2 computes the base 2 exponential of the input tuple T. The exponential is always returned as a floating point number. The exponential of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_exp2, which can be used in an expression in the following syntax:

```
Exp := exp2(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Exp** (output_control) . . . . . number(-array) ⇝ *real*
Base 2 exponential of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_exp, tuple_exp10, tuple_pow

See also

tuple_log, tuple_log2, tuple_log10

Module

Foundation

tuple_fabs (: : T : Abs)

Compute the absolute value of a tuple (as floating point numbers).

tuple_fabs computes the absolute value of the input tuple T. In contrast to tuple_abs, the absolute value is always returned as a floating point number by tuple_fabs. The absolute value of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_fabs, which can be used in an expression in the following syntax:

```
Abs := fabs(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Abs** (output_control) . . . . . number(-array) ⇝ *real*
Absolute value of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_abs

Module

Foundation

tuple_floor (: : T : Floor)

Compute the floor function of a tuple.

tuple_floor computes the floor function of the input tuple T, i.e., the largest integer less than or equal to T. The floor function is always returned as a floating point number. The floor function of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_floor, which can be used in an expression in the following syntax:

```
Floor := floor(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Floor** (output_control) . . . . . number(-array) ⇝ *real*
Floor function of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_ceil

Module

Foundation

tuple_fmod (: : T1, T2 : Fmod)

Calculate the remainder of the floating point division of two tuples.

tuple_fmod computes the remainder of the floating point division of the input tuples T1/T2. If both tuples have the same length the division is performed for the corresponding elements of both tuples. Otherwise, either T1 or T2 must have length 1. In this case, the division is performed for each element of the longer tuple with the single element of the other tuple. The result is always a floating point number. The division of strings is not allowed.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_fmod, which can be used in an expression in the following syntax:

```
Fmod := fmod(T1, T2)
```

Parameters

▷ **T1** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple 1.

▷ **T2** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple 2.
Restriction: T2 != 0.0

▷ **Fmod** (output_control) . . . . . number(-array) ⇝ *real*
Remainder of the division of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_mod

See also

tuple_floor, tuple_ceil

Module

Foundation

tuple_hypot (: : T1, T2 : Hypot)

Calculate the hypotenuse of two tuples.

tuple_hypot computes the hypotenuse $\sqrt{T1^2 + T2^2}$ of the input tuples. The calculation is performed without undue arithmetic overflow or underflow during the intermediate steps of the calculation. If both tuples have the same length, the hypotenuse is applied to the corresponding elements of both tuples. Otherwise, either T1 or T2 must have length 1. In this case, the hypotenuse is calculated for each element of the longer tuple with the single element of the other tuple. The result is always a floating point number. The hypotenuse of strings is not allowed.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_hypot, which can be used in an expression in the following syntax:

```
Hypot := hypot(T1, T2)
```

Parameters

▷ **T1** (input_control) . . . number(-array) ⇝ *real* / integer
Input tuple 1.

▷ **T2** (input_control) . . . number(-array) ⇝ *real* / integer
Input tuple 2.

▷ **Hypot** (output_control) . . . number(-array) ⇝ *real*
Hypotenuse of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sqrt

See also

tuple_mult, tuple_add

Module

Foundation

tuple_ldexp (: : T1, T2 : Ldexp)

Calculate the ldexp function of two tuples.

tuple_ldexp computes the ldexp function of the input tuples, i.e., $T1 \cdot 2^{floor(T2)}$. If both tuples have the same length the operation is performed for the corresponding elements of both tuples. Otherwise, either T1 or T2 must

have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The result is always a floating point number. The ldexp function of strings is not allowed.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_ldexp`, which can be used in an expression in the following syntax:

```
Ldexp := ldexp(T1, T2)
```

Parameters

- ▷ **T1** (input_control) . . . number(-array) ⇝ *real* / integer
  Input tuple 1.
- ▷ **T2** (input_control) . . . number(-array) ⇝ *real* / integer
  Input tuple 2.
- ▷ **Ldexp** (output_control) . . . number(-array) ⇝ *real*
  Ldexp function of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`tuple_exp`

Module

Foundation

tuple_lgamma (: : T : LogGamma)

Compute the logarithm of the absolute value of the gamma function of a tuple.

`tuple_lgamma` computes the logarithm of the absolute value of the gamma function of the input tuple `T` (for the definition of the gamma function, see `tuple_tgamma`). It is defined for every real number except for nonpositive integers. The logarithm of the absolute value of the gamma function is always returned as a floating point number in `LogGamma`. The logarithm of the absolute value of the gamma function of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_lgamma`, which can be used in an expression in the following syntax:

```
LogGamma := lgamma(T)
```

Parameters

- ▷ **T** (input_control) . . . number(-array) ⇝ *real* / integer
  Input tuple.
- ▷ **LogGamma** (output_control) . . . number(-array) ⇝ *real*
  Logarithm of the absolute value of the gamma function of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Alternatives

tuple_tgamma

Module

Foundation

tuple_log (: : T : Log)

Compute the natural logarithm of a tuple.

tuple_log computes the natural logarithm of the input tuple T. The natural logarithm is always returned as a floating point number. The natural logarithm of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_log, which can be used in an expression in the following syntax:

```
Log := log(T)
```

Parameters

▷ **T** (input_control) ..... number(-array) ⇝ *real* / integer
Input tuple.
Restriction: T > 0

▷ **Log** (output_control) ..... number(-array) ⇝ *real*
Natural logarithm of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_log2, tuple_log10

See also

tuple_exp, tuple_exp2, tuple_exp10, tuple_pow

Module

Foundation

tuple_log10 (: : T : Log)

Compute the base 10 logarithm of a tuple.

tuple_log10 computes the base 10 logarithm of the input tuple T. The logarithm is always returned as a floating point number. The logarithm of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_log10, which can be used in an expression in the following syntax:

```
Log := log10(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.
Restriction: T > 0

▷ **Log** (output_control) . . . . . number(-array) ⇝ *real*
Base 10 logarithm of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_log, tuple_log2

See also

tuple_exp, tuple_exp2, tuple_exp10, tuple_pow

Module

Foundation

tuple_log2 (: : T : Log)

Compute the base 2 logarithm of a tuple.

tuple_log2 computes the base 2 logarithm of the input tuple T. The logarithm is always returned as a floating point number. The logarithm of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_log2, which can be used in an expression in the following syntax:

```
Log := log2(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.
Restriction: T > 0

▷ **Log** (output_control) . . . . . number(-array) ⇝ *real*
Base 2 logarithm of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_log, tuple_log10

See also

tuple_exp, tuple_exp2, tuple_exp10, tuple_pow

Module

Foundation

tuple_max2 (: : T1, T2 : Max2)

Calculate the elementwise maximum of two tuples.

tuple_max2 returns the elementwise maximum of the input tuples T1 and T2 in the output tuple Max2. If both tuples have the same length the corresponding elements of both tuples are compared. Otherwise, either T1 or T2 must have length 1. In this case, the comparison is performed for each element of the longer tuple with the single element of the other tuple. If the corresponding elements of the two tuples have the same type, it is allowed to mix strings with numerical values.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_max2, which can be used in an expression in the following syntax:

```
Max2 := max2(T1, T2)
```

Parameters

- ▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string
  Input tuple 1.
- ▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string
  Input tuple 2.
- ▷ **Max2** (output_control) . . . . . tuple(-array) ⇝ *real* / integer / string
  Elementwise maximum of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

tuple_min2, tuple_min, tuple_max, tuple_cumul

Module

Foundation

tuple_min2 (: : T1, T2 : Min2)

Calculate the elementwise minimum of two tuples.

tuple_min2 returns the elementwise minimum of the input tuples T1 and T2 in the output tuple Min2. If both tuples have the same length the corresponding elements of both tuples are compared. Otherwise, either T1 or T2 must have length 1. In this case, the comparison is performed for each element of the longer tuple with the single element of the other tuple. If the corresponding elements of the two tuples have the same type, it is allowed to mix strings with numerical values.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_min2, which can be used in an expression in the following syntax:

```
Min2 := min2(T1, T2)
```

Parameters

▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string
Input tuple 1.

▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string
Input tuple 2.

▷ **Min2** (output_control) . . . . . tuple(-array) ⇝ *real* / integer / string
Elementwise minimum of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`tuple_max2`, `tuple_min`, `tuple_max`

Module

Foundation

tuple_mod (: : T1, T2 : Mod)

Calculate the remainder of the integer division of two tuples.

`tuple_mod` returns the remainder of the integer division of the input tuples `T1`/`T2` in the output tuple `Mod`. If both tuples have the same length the division is performed for the corresponding elements of both tuples. Otherwise, either `T1` or `T2` must have length 1. In this case, the division is performed for each element of the longer tuple with the single element of the other tuple. The result is always an integer number. The division of strings is not allowed.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_mod`, which can be used in an expression in the following syntax:

```
Mod := T1 % T2
```

Parameters

▷ **T1** (input_control) . . . . . number(-array) ⇝ *integer*
Input tuple 1.

▷ **T2** (input_control) . . . . . number(-array) ⇝ *integer*
Input tuple 2.
Restriction: T2 != 0

▷ **Mod** (output_control) . . . . . number(-array) ⇝ *integer*
Remainder of the division of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

`tuple_fmod`, `tuple_div`

Module

Foundation

tuple_mult (: : P1, P2 : Prod)

Multiply two tuples.

tuple_mult computes the product of the input tuples P1 and P2. If both tuples have the same length the corresponding elements of both tuples are multiplied. Otherwise, either P1 or P2 must have length 1. In this case, the multiplication is performed for each element of the longer tuple with the single element of the other tuple. If two integer numbers are multiplied, the result is again an integer number. If one of the operands is a floating point number, the result is a floating point number. The multiplication of strings is not allowed.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_mult, which can be used in an expression in the following syntax:

```
Prod := P1 * P2
```

Parameters

- ▷ **P1** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input tuple 1.
- ▷ **P2** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input tuple 2.
- ▷ **Prod** (output_control) . . . . . number(-array) ⇝ *real* / integer
  Product of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_div

Module

Foundation

tuple_neg (: : T : Neg)

Negate a tuple.

tuple_neg computes the negation of the input tuple T, i.e., Neg = −T. The negation of an integer number is again an integer number. The negation of a floating point number is a floating point number. The negation of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_neg, which can be used in an expression in the following syntax:

```
Neg := -T
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Neg** (output_control) . . . . . number(-array) ⇝ *real* / integer
Negation of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

tuple_pow (: : T1, T2 : Pow)

Calculate the power function of two tuples.

`tuple_pow` computes the power function of the input tuples $T1^{T2}$. If both tuples have the same length the power function is applied to the corresponding elements of both tuples. Otherwise, either `T1` or `T2` must have length 1. In this case, the power function is performed for each element of the longer tuple with the single element of the other tuple. The result is always a floating point number. The power function of strings is not allowed.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_pow`, which can be used in an expression in the following syntax:

```
Pow := pow(T1, T2)
```

Parameters

▷ **T1** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple 1.

▷ **T2** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple 2.

▷ **Pow** (output_control) . . . . . number(-array) ⇝ *real*
Power function of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_exp`

See also

`tuple_log`, `tuple_log10`

Module

Foundation

tuple_rad (: : Deg : Rad)

Convert a tuple from degrees to radians.

tuple_rad converts the input tuple Deg from degrees to radians. The result is always returned as a floating point number. The conversion of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_rad, which can be used in an expression in the following syntax:

```
Rad := rad(Deg)
```

Parameters

▷ **Deg** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Rad** (output_control) . . . . . number(-array) ⇝ *real*
Input tuple in radians.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

tuple_deg

Module

Foundation

tuple_sgn (: : T : Sgn)

Calculate the sign of a tuple.

tuple_sgn calculates the sign of the elements of the input tuple T and returns them as integer numbers (-1,0,1) in the output tuple Sgn. The calculation of the sign of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_sgn, which can be used in an expression in the following syntax:

```
Sgn := sgn(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Sgn** (output_control) . . . . . number(-array) ⇝ *integer*
Signs of the input tuple as integer numbers.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

tuple_sin (: : T : Sin)

Compute the sine of a tuple.

tuple_sin computes the sine of the input tuple T. The sine is always returned as a floating point number in Sin. The angles in T are represented in radians. The sine of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_sin, which can be used in an expression in the following syntax:

```
Sin := sin(T)
```

Parameters

- ▷ **T** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
  Input tuple.
- ▷ **Sin** (output_control) . . . . . number(-array) ⇝ *real*
  Sine of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_cos, tuple_tan

See also

tuple_asin, tuple_sinh, tuple_asinh

Module

Foundation

tuple_sinh (: : T : Sinh)

Compute the hyperbolic sine of a tuple.

tuple_sinh computes the hyperbolic sine of the input tuple T. The hyperbolic sine is always returned as a floating point number in Sinh. The hyperbolic sine of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_sinh, which can be used in an expression in the following syntax:

```
Sinh := sinh(T)
```

Parameters

- ▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
  Input tuple.
- ▷ **Sinh** (output_control) . . . . . number(-array) ⇝ *real*
  Hyperbolic sine of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_cosh, tuple_tanh

See also

tuple_asinh, tuple_sin, tuple_asin

Module

Foundation

tuple_sqrt (: : T : Sqrt)

Compute the square root of a tuple.

tuple_sqrt computes the square root of the input tuple T. The square root is always returned as a floating point number. The square root of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_sqrt, which can be used in an expression in the following syntax:

```
Sqrt := sqrt(T)
```

Parameters

▷ **T** (input_control) ..... number(-array) ⇝ *real* / integer
Input tuple.
Restriction: T >= 0

▷ **Sqrt** (output_control) ..... number(-array) ⇝ *real*
Square root of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_pow

See also

tuple_cbrt, tuple_hypot

Module

Foundation

tuple_sub (: : D1, D2 : Diff)

Subtract two tuples.

tuple_sub computes the difference of the input tuples D1 and D2. If both tuples have the same length the corresponding elements of both tuples are subtracted. Otherwise, either D1 or D2 must have length 1. In this case, the subtraction is performed for each element of the longer tuple with the single element of the other tuple. If two integer numbers are subtracted, the result is again an integer number. If one of the operands is a floating point number, the result is a floating point number. The subtraction of strings is not allowed.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_sub`, which can be used in an expression in the following syntax:

```
Diff := D1 - D2
```

Parameters

▷ **D1** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple 1.

▷ **D2** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple 2.

▷ **Diff** (output_control) . . . . . number(-array) ⇝ *real* / integer
Difference of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_add`

Module

Foundation

tuple_tan (: : T : Tan)

Compute the tangent of a tuple.

`tuple_tan` computes the tangent of the input tuple `T`. The tangent is always returned as a floating point number in `Tan`. The angles in `T` are represented in radians. The tangent of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_tan`, which can be used in an expression in the following syntax:

```
Tan := tan(T)
```

Parameters

▷ **T** (input_control) . . . . . angle.rad(-array) ⇝ *real* / integer
Input tuple.

▷ **Tan** (output_control) . . . . . number(-array) ⇝ *real*
Tangent of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_sin`, `tuple_cos`

See also

`tuple_atan`, `tuple_atan2`, `tuple_tanh`, `tuple_atanh`

Module

Foundation

tuple_tanh (: : T : Tanh)

Compute the hyperbolic tangent of a tuple.

tuple_tanh computes the hyperbolic tangent of the input tuple T. The hyperbolic tangent is always returned as a floating point number in Tanh. The hyperbolic tangent of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_tanh, which can be used in an expression in the following syntax:

```
Tanh := tanh(T)
```

Parameters

- ▷ **T** (input_control) . . . number(-array) ⇝ *real* / integer
  Input tuple.
- ▷ **Tanh** (output_control) . . . number(-array) ⇝ *real*
  Hyperbolic tangent of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sinh, tuple_cosh

See also

tuple_atanh, tuple_tan, tuple_atan, tuple_atan2

Module

Foundation

tuple_tgamma (: : T : Gamma)

Compute the gamma function of a tuple.

tuple_tgamma computes the gamma function of the input tuple T. The gamma function is defined as:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt.$$

It is defined for every real number except for nonpositive integers. For nonnegative integer m, it holds that

$$\Gamma(m+1) = m!$$

and, more generally, for all x:

$$\Gamma(x+1) = x\Gamma(x).$$

The value of the gamma function is always returned as a floating point number in Gamma. The gamma function of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_tgamma`, which can be used in an expression in the following syntax:

```
Gamma := tgamma(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Gamma** (output_control) . . . . . number(-array) ⇝ *real*
Value of the gamma function of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_lgamma`

Module

Foundation

28.2 Bit Operations

tuple_band (: : T1, T2 : BAnd)

Compute the bitwise and of two tuples.

`tuple_band` computes the bitwise and of the input tuples `T1` and `T2`. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either `T1` or `T2` must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_band`, which can be used in an expression in the following syntax:

```
BAnd := T1 band T2
```

Parameters

▷ **T1** (input_control) . . . . . integer(-array) ⇝ *integer*
Input tuple 1.

▷ **T2** (input_control) . . . . . integer(-array) ⇝ *integer*
Input tuple 2.

▷ **BAnd** (output_control) . . . . . integer(-array) ⇝ *integer*
Binary and of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_bor, tuple_bxor, tuple_bnot

See also

tuple_and, tuple_or, tuple_xor, tuple_not

Module

Foundation

tuple_bnot (: : T : BNot)

Compute the bitwise not of a tuple.

tuple_bnot computes the bitwise not of the input tuple T. The input tuple must contain only integer numbers.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_bnot, which can be used in an expression in the following syntax:

```
BNot := bnot T
```

Parameters

- ▷ **T** (input_control) ..... integer(-array) ⇝ *integer*
  Input tuple.
- ▷ **BNot** (output_control) ..... integer(-array) ⇝ *integer*
  Binary not of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_band, tuple_bor, tuple_bxor

See also

tuple_and, tuple_or, tuple_xor, tuple_not

Module

Foundation

tuple_bor (: : T1, T2 : BOr)

Compute the bitwise or of two tuples.

tuple_bor computes the bitwise or of the input tuples T1 and T2. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either T1 or T2 must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_bor, which can be used in an expression in the following syntax:

```
BOr := T1 bor T2
```

Parameters

▷ **T1** (input_control) . . . integer(-array) ⇝ *integer*
Input tuple 1.
▷ **T2** (input_control) . . . integer(-array) ⇝ *integer*
Input tuple 2.
▷ **BOr** (output_control) . . . integer(-array) ⇝ *integer*
Binary or of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_band, tuple_bxor, tuple_bnot

See also

tuple_and, tuple_or, tuple_xor, tuple_not

Module

Foundation

tuple_bxor (: : T1, T2 : BXor)

Compute the bitwise exclusive or of two tuples.

tuple_bxor computes the bitwise exclusive or of the input tuples T1 and T2. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either T1 or T2 must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_bxor, which can be used in an expression in the following syntax:

```
BXor := T1 bxor T2
```

Parameters

▷ **T1** (input_control) . . . integer(-array) ⇝ *integer*
Input tuple 1.
▷ **T2** (input_control) . . . integer(-array) ⇝ *integer*
Input tuple 2.
▷ **BXor** (output_control) . . . integer(-array) ⇝ *integer*
Binary exclusive or of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_band, tuple_bor, tuple_bnot

See also

tuple_and, tuple_or, tuple_xor, tuple_not

Module

Foundation

tuple_lsh (: : T, Shift : Lsh)

Shift a tuple bitwise to the left.

tuple_lsh shifts the tuple T bitwise to the left by Shift places. If no overflow occurs, this operation is equivalent to a multiplication by 2^{Shift}. If T is negative, the result depends on the hardware. If Shift is negative or larger than 32, the result is undefined. If both tuples have the same length the corresponding elements of both tuples are shifted. Otherwise, either T or Shift must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_lsh, which can be used in an expression in the following syntax:

```
Lsh := lsh(T, Shift)
```

Parameters

- ▷ **T** (input_control) ..... integer(-array) ⇝ *integer*
  Input tuple.
- ▷ **Shift** (input_control) ..... integer(-array) ⇝ *integer*
  Number of places to shift the input tuple.
- ▷ **Lsh** (output_control) ..... integer(-array) ⇝ *integer*
  Shifted input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_mult

See also

tuple_rsh

Module

Foundation

tuple_rsh (: : T, Shift : Rsh)

Shift a tuple bitwise to the right.

tuple_rsh shifts the tuple T bitwise to the right by Shift places. This operation is equivalent to a division by 2^{Shift}. If T is negative, the result depends on the hardware. If Shift is negative or larger than 32, the result is undefined. If both tuples have the same length the corresponding elements of both tuples are shifted. Otherwise, either T or Shift must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Exception: Empty input tuple

If any of the input tuples is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_rsh, which can be used in an expression in the following syntax:

```
Rsh := rsh(T, Shift)
```

Parameters

▷ **T** (input_control) . . . . . integer(-array) ⇝ *integer*
Input tuple.
▷ **Shift** (input_control) . . . . . integer(-array) ⇝ *integer*
Number of places to shift the input tuple.
▷ **Rsh** (output_control) . . . . . integer(-array) ⇝ *integer*
Shifted input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_div

See also

tuple_lsh

Module

Foundation

28.3 Comparison

tuple_equal (: : T1, T2 : Equal)

Test whether two tuples are equal.

tuple_equal tests whether the two input tuples T1 and T2 are equal by comparing the tuples elementwise. Two tuples are equal, if they have got the same number of elements and if their elements are equal. Two tuple elements are equal, if they are both (integer or floating point) numbers or both are strings and contain the same value.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_equal, which can be used in an expression in the following syntax:

Equal := T1 == T2 or Equal := T1 = T2

Parameters

▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 1.
▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 2.
▷ **Equal** (output_control) . . . . . integer ⇝ *integer*
Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_not_equal, tuple_less, tuple_greater, tuple_less_equal, tuple_greater_equal, tuple_equal_elem, tuple_not_equal_elem, tuple_less_elem, tuple_greater_elem, tuple_less_equal_elem, tuple_greater_equal_elem

Module

Foundation

tuple_equal_elem (: : T1, T2 : Equal)

Test, whether two tuples are elementwise equal.

`tuple_equal_elem` tests whether the two input tuples `T1` and `T2` are elementwise equal. If both tuples have the same length, the corresponding elements of both tuples are compared. Otherwise, either `T1` or `T2` must have length 1. In this case, the comparison is performed for each element of the longer tuple with the single element of the other tuple. Two tuple elements are equal if they are both (integer or floating point) numbers or both are strings and contain the same value.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_equal_elem`, which can be used in an expression in the following syntax:

`Equal := T1 [==] T2` or `Equal := T1 [=] T2`

Parameters

- ▷ **T1** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple 1.
- ▷ **T2** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple 2.
- ▷ **Equal** (output_control) ..... integer(-array) ⇝ *integer*
  Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_not_equal_elem`, `tuple_less_elem`, `tuple_greater_elem`, `tuple_less_equal_elem`, `tuple_greater_equal_elem`, `tuple_equal`, `tuple_not_equal`, `tuple_less`, `tuple_greater`, `tuple_less_equal`, `tuple_greater_equal`

Module

Foundation

tuple_greater (: : T1, T2 : Greater)

Test whether a tuple is greater than another tuple.

`tuple_greater` tests whether the input tuple `T1` is greater than `T2`. A tuple `T1` is said to be greater than a tuple `T2`, if `T1` has been found to be greater when comparing it elementwise to `T2` or if the first $min(|T1|, |T2|)$ elements are equal and `T1` has got more elements than `T2`.

With the elementwise comparison, the elements of `T1` and `T2` are compared one by one. If the two elements are equal, the next pair of elements will be examined. If the element of `T1` is greater than that from `T2`, the result is 1 and the comparison will be aborted. If the element of `T1` is less than that from `T2`, the result is 0 and the comparison will be aborted. If all elements have been tested and if all were equal, the result is 0.

As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise `tuple_greater` returns an error.

Exception: Empty input tuples

If one input tuple is empty and the other is not, the tuple that is not empty is considered greater than the empty one.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_greater`, which can be used in an expression in the following syntax:

```
Greater := T1 > T2
```

Parameters

▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 1.

▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 2.

▷ **Greater** (output_control) . . . . . integer ⇝ *integer*
Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_greater_equal`, `tuple_less`, `tuple_less_equal`, `tuple_equal`, `tuple_not_equal`, `tuple_equal_elem`, `tuple_not_equal_elem`, `tuple_less_elem`, `tuple_greater_elem`, `tuple_less_equal_elem`, `tuple_greater_equal_elem`

Module

Foundation

tuple_greater_elem (: : T1, T2 : Greater)

Test, whether a tuple is elementwise greater than another tuple.

`tuple_greater_elem` tests elementwise whether the input tuple `T1` is greater than `T2`. If both tuples have the same length, the corresponding elements of both tuples are compared. Otherwise, either `T1` or `T2` must have length 1. In this case, the comparison is performed for each element of the longer tuple with the single element of the other tuple. As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise `tuple_greater_elem` returns an error.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_greater_elem`, which can be used in an expression in the following syntax:

```
Greater := T1 [>] T2
```

Parameters

▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 1.

▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 2.

▷ **Greater** (output_control) . . . . . integer(-array) ⇝ *integer*
Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Alternatives

tuple_equal_elem, tuple_not_equal_elem, tuple_less_elem, tuple_less_equal_elem, tuple_greater_equal_elem, tuple_equal, tuple_not_equal, tuple_less, tuple_greater, tuple_less_equal, tuple_greater_equal

Module

Foundation

tuple_greater_equal (: : T1, T2 : Greatereq)

Test whether a tuple is greater or equal to another tuple.

tuple_greater_equal tests whether the input tuple T1 is greater or equal to T2. A tuple T1 is said to be greater or equal to a tuple T2, if T1 is not less than T2. A tuple T1 is said to be less than a tuple T2, if T1 has been found to be less when comparing it elementwise to T2 or if the first $min(|T1|, |T2|)$ elements are equal and T1 has got less elements than T2.

With the elementwise comparison, the elements of T1 and T2 are compared one by one. If the two elements are equal, the next pair of elements will be examined. If the element of T1 is greater than that from T2, the result is 1 and the comparison will be aborted. If the element of T1 is less than that from T2, the result is 0 and the comparison will be aborted. If all elements have been tested and if all were equal, the result is 1.

As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise tuple_greater_equal returns an error.

Exception: Empty input tuples

If one input tuple is empty and the other is not, the tuple that is not empty is considered greater than the empty one.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_greater_equal, which can be used in an expression in the following syntax:

```
Greatereq := T1 >= T2
```

Parameters

- ▷ **T1** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple 1.
- ▷ **T2** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple 2.
- ▷ **Greatereq** (output_control) ..... integer ⇝ *integer*
  Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_greater, tuple_less, tuple_less_equal, tuple_equal, tuple_not_equal, tuple_equal_elem, tuple_not_equal_elem, tuple_less_elem, tuple_greater_elem, tuple_less_equal_elem, tuple_greater_equal_elem

Module

Foundation

tuple_greater_equal_elem (: : T1, T2 : Greatereq)

Test, whether a tuple is elementwise greater or equal to another tuple.

tuple_greater_equal_elem tests elementwise whether the input tuple T1 is greater than or equal to T2. If both tuples have the same length, the corresponding elements of both tuples are compared. Otherwise, either T1 or T2 must have length 1. In this case, the comparison is performed for each element of the longer tuple with the single element of the other tuple. As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise tuple_greater_equal_elem returns an error.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_greater_equal_elem, which can be used in an expression in the following syntax:

```
Greatereq := T1 [>=] T2
```

Parameters

- ▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Input tuple 1.
- ▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Input tuple 2.
- ▷ **Greatereq** (output_control) . . . . . integer(-array) ⇝ *integer*
  Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_equal_elem, tuple_not_equal_elem, tuple_less_elem, tuple_greater_elem, tuple_less_equal_elem, tuple_equal, tuple_not_equal, tuple_less, tuple_greater, tuple_less_equal, tuple_greater_equal

Module

Foundation

tuple_less (: : T1, T2 : Less)

Test whether a tuple is less than another tuple.

tuple_less tests whether the input tuple T1 is less than T2. A tuple T1 is said to be less than a tuple T2, if T1 has been found to be less when comparing it elementwise to T2 or if the first $min(|T1|, |T2|)$ elements are equal and T1 has got less elements than T2.

With the elementwise comparison, the elements of T1 and T2 are compared one by one. If the two elements are equal, the next pair of elements will be examined. If the element of T1 is less than that from T2, the result is 1 and the comparison will be aborted. If the element of T1 is greater than that from T2, the result is 0 and the comparison will be aborted. If all elements have been tested and if all were equal, the result is 0.

As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise tuple_less returns an error.

Exception: Empty input tuples

If one input tuple is empty and the other is not, the tuple that is empty is considered less than the one that is not empty.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_less, which can be used in an expression in the following syntax:

```
Less := T1 < T2
```

Parameters

▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 1.

▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 2.

▷ **Less** (output_control) . . . . . integer ⇝ *integer*
Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_less_equal, tuple_greater, tuple_greater_equal, tuple_equal, tuple_not_equal, tuple_equal_elem, tuple_not_equal_elem, tuple_less_elem, tuple_greater_elem, tuple_less_equal_elem, tuple_greater_equal_elem

Module

Foundation

tuple_less_elem (: : T1, T2 : Less)

Test, whether a tuple is elementwise less than another tuple.

tuple_less_elem tests elementwise whether the input tuple T1 is less than T2. If both tuples have the same length, the corresponding elements of both tuples are compared. Otherwise, either T1 or T2 must have length 1. In this case, the comparison is performed for each element of the longer tuple with the single element of the other tuple. As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise tuple_less_elem returns an error.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_less_elem, which can be used in an expression in the following syntax:

```
Less := T1 [<] T2
```

Parameters

▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 1.

▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple 2.

▷ **Less** (output_control) . . . . . integer(-array) ⇝ *integer*
Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_equal_elem, tuple_not_equal_elem, tuple_greater_elem, tuple_less_equal_elem, tuple_greater_equal_elem, tuple_equal, tuple_not_equal, tuple_less, tuple_greater, tuple_less_equal, tuple_greater_equal

Module

Foundation

tuple_less_equal (: : T1, T2 : Lesseq)

Test whether a tuple is less or equal to another tuple.

tuple_less_equal tests whether the input tuple T1 is less or equal to T2. A tuple T1 is said to be less or equal to a tuple T2, if T1 is not greater than T2. A tuple T1 is said to be greater than a tuple T2, if T1 has been found to be greater when comparing it elementwise to T2 or if the first $min(|T1|, |T2|)$ elements are equal and T1 has got more elements than T2.

With the elementwise comparison, the elements of T1 and T2 are compared one by one. If the two elements are equal, the next pair of elements will be examined. If the element of T1 is less than that from T2, the result is 1 and the comparison will be aborted. If the element of T1 is greater than that from T2, the result is 0 and the comparison will be aborted. If all elements have been tested and if all were equal, the result is 1.

As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise tuple_less_equal returns an error.

Exception: Empty input tuples

If one input tuple is empty and the other is not, the tuple that is empty is considered less than the one that is not empty.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_less_equal, which can be used in an expression in the following syntax:

```
Lesseq := T1 <= T2
```

Parameters

- ▷ **T1** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple 1.
- ▷ **T2** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple 2.
- ▷ **Lesseq** (output_control) ..... integer ⇝ *integer*
  Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_less, tuple_greater, tuple_greater_equal, tuple_equal, tuple_not_equal, tuple_equal_elem, tuple_not_equal_elem, tuple_less_elem, tuple_greater_elem, tuple_less_equal_elem, tuple_greater_equal_elem

Module

Foundation

tuple_less_equal_elem (: : T1, T2 : Lesseq)

Test, whether a tuple is elementwise less or equal to another tuple.

tuple_less_equal_elem tests elementwise whether the input tuple T1 is less than or equal to T2. If both tuples have the same length, the corresponding elements of both tuples are compared. Otherwise, either T1 or T2 must have length 1. In this case, the comparison is performed for each element of the longer tuple with the single element of the other tuple. As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise tuple_less_equal_elem returns an error.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_less_equal_elem, which can be used in an expression in the following syntax:

```
Lesseq := T1 [<=] T2
```

Parameters

- ▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Input tuple 1.
- ▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Input tuple 2.
- ▷ **Lesseq** (output_control) . . . . . integer(-array) ⇝ *integer*
  Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_equal_elem, tuple_not_equal_elem, tuple_greater_elem, tuple_less_elem, tuple_greater_equal_elem, tuple_equal, tuple_not_equal, tuple_less, tuple_greater, tuple_less_equal, tuple_greater_equal

Module

Foundation

tuple_not_equal (: : T1, T2 : Nequal)

Test whether two tuples are not equal.

tuple_not_equal tests whether the two input tuples T1 and T2 are not equal. Two tuples are not equal, if they consist of a different number of elements or if they differ when compared elementwise. Two tuple elements differ, if they are of types that may not be compared (e.g., one string and one integer) or if they differ in their values.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_not_equal, which can be used in an expression in the following syntax:

```
Nequal := T1 != T2
```
or
```
Nequal := T1 # T2
```

Parameters

- ▷ **T1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Input tuple 1.
- ▷ **T2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Input tuple 2.
- ▷ **Nequal** (output_control) . . . . . integer ⇝ *integer*
  Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_equal, tuple_less, tuple_greater, tuple_less_equal, tuple_greater_equal, tuple_equal_elem, tuple_not_equal_elem, tuple_less_elem, tuple_greater_elem, tuple_less_equal_elem, tuple_greater_equal_elem

Module

Foundation

tuple_not_equal_elem (: : T1, T2 : Nequal)

Test, whether two tuples are elementwise not equal.

tuple_not_equal_elem tests whether the two input tuples T1 and T2 are elementwise not equal. If both tuples have the same length, the corresponding elements of both tuples are compared. Otherwise, either T1 or T2 must have length 1. In this case, the comparison is performed for each element of the longer tuple with the single element of the other tuple. Two tuple elements differ if they are of types that may not be compared (e.g., one string and one integer) or if they differ in their values.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_not_equal_elem, which can be used in an expression in the following syntax:

Nequal := T1 [!=] T2 or Nequal := T1 [#] T2

Parameters

▷ **T1** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
Input tuple 1.

▷ **T2** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
Input tuple 2.

▷ **Nequal** (output_control) ..... integer(-array) ⇝ *integer*
Result of the comparison of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_equal_elem, tuple_less_elem, tuple_greater_elem, tuple_less_equal_elem, tuple_greater_equal_elem, tuple_equal, tuple_not_equal, tuple_less, tuple_greater, tuple_less_equal, tuple_greater_equal

Module

Foundation

28.4 Conversion

handle_to_integer (: : Handle : CastedHandle)

Convert a handle into an integer.

It is not recommended to use this operator in HDevelop. This operator is only provided for reasons of backward compatibility.

handle_to_integer converts the handle Handle into an integer representation and returns it in CastedHandle.

If the legacy handle mode is disabled (the default), the ownership of the handle is not transferred. Once all instances of the handle, such as in the tuple Handle, were overwritten, the handle and its content will be destroyed and CastedHandle will become invalid. In that case, if CastedHandle should be passed to code parts that run in legacy handle mode, a reference to the original Handle must be kept in order to avoid clearing the handle.

If the legacy handle mode is enabled, the ownership of the handle is transferred. The handle must then be cleared using clear_handle or the clear operator of the corresponding semantic type of the handle.

Attention

It is not recommended to use this operator in HDevelop. It is solely provided to enable backward compatibility with legacy code.

Parameters

▷ **Handle** (input_control) ..... handle(-array) ⇝ *handle*
The handle to be cast.

▷ **CastedHandle** (output_control) ..... pointer(-array) ⇝ *integer*
The handle cast to an integer value.

Result

If the parameters are valid, the operator handle_to_integer returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

integer_to_handle

Module

Foundation

integer_to_handle (: : IntegerHandle : Handle)

Convert an integer into a handle.

It is not recommended to use this operator in HDevelop. This operator is only provided for reasons of backward compatibility.

integer_to_handle converts the integer IntegerHandle, that represents a handle, to a handle of type HANDLE_PAR and returns it in Handle. This allows to convert handles created in code parts where the legacy handle mode is enabled into proper handles.

If the legacy handle mode is disabled (the default), the ownership of the handle is transferred onto the handle. In this case, the handle will automatically be cleared once all references to that handle, such as in the tuple Handle, are overwritten. In that case, if the original IntegerHandle should continue to be used, a reference to the returned Handle must be kept in order to avoid clearing the handle.

If the legacy handle mode is enabled, the returned value will again be an integer. Additionally, the handle must then be cleared using clear_handle or the clear operator of the corresponding semantic type of the handle.

Attention

It is not recommended to use this operator in HDevelop. This operator is solely provided to enable backward compatibility with legacy code.

Parameters

▷ **IntegerHandle** (input_control) . . . . . pointer(-array) ⇝ *integer*
The handle as integer.

▷ **Handle** (output_control) . . . . . handle(-array) ⇝ *handle*
The handle as handle.

Result

If the parameters are valid, the operator `integer_to_handle` returns the value `2` (`H_MSG_TRUE`). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

See also

`handle_to_integer`

Module

Foundation

tuple_chr (: : T : Chr)

Convert a tuple of integer numbers into strings.

`tuple_chr` converts the input tuple `T`, consisting of integer numbers, into a tuple of strings, each of length 1. When the encoding used in the HALCON library is UTF-8 (see `set_system ('filename_encoding', 'utf8')`) and the string operators are set to work by code points (see `set_system('tuple_string_operator_mode', 'codepoint')`), which is the default for both, the operator accepts Unicode character codes and returns strings with the appropriate UTF-8 representations of the characters. When the HALCON library encoding is set to *'locale'* or the string operator mode is *'byte'*, the operator accepts only numbers between 0 and 256. In that case the operator `tuple_chr` returns strings with one byte length each, where the input number is set as ANSI code. See also Tuple / String Operations for a more detailed description of the different modes and further encoding issues.

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_chr`, which can be used in an expression in the following syntax:

```
Chr := chr(T)
```

Parameters

▷ **T** (input_control) . . . . . integer(-array) ⇝ *integer*
Input tuple with Unicode character codes or ANSI codes.
Restriction: 0 <= T

▷ **Chr** (output_control) . . . . . string(-array) ⇝ *string*
Output tuple with strings built from the character codes in the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_chrt

See also

tuple_ord, tuple_ords

Module

Foundation

tuple_chrt (: : T : Chrt)

Convert a tuple of integer numbers into strings.

tuple_chrt converts the input tuple T, consisting of integer numbers, into a tuple of strings and integer numbers (where only the number 0 is passed as number to the output). All other numbers are converted to characters that are concatenated to one string up to the next 0 in the input tuple. When the encoding used in the HALCON library is UTF-8 (see set_system('filename_encoding', 'utf8')) and the string operators are set to work by code points (see set_system('tuple_string_operator_mode', 'codepoint')), which is the default for both, the operator interprets the input numbers as Unicode character codes and transforms them to the appropriate UTF-8 representations of the characters. When the HALCON library encoding is set to *'locale'* or the string operator mode is *'byte'*, the operator accepts only numbers between 0 and 256. In that case they are interpreted as ANSI codes from which the output string has to be built byte by byte. In that mode it is not checked whether the byte sequences will build valid characters according to the current string encoding. See also Tuple / String Operations for a more detailed description of the different modes and further encoding issues.

The operator tries to pack as many input numbers into one string as possible. If the number 0 occurs in the input tuple T, the current string is terminated and the number 0 is added to the output tuple. If in the input tuple more numbers follow, a new string is started. This operator can be used to convert data read with read_serial into strings. This approach allows also to read bytes with the value 0.

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_chrt, which can be used in an expression in the following syntax:

```
Chrt := chrt(T)
```

Parameters

▷ **T** (input_control) ..... integer(-array) ⇝ *integer*
Input tuple with integer numbers.
Restriction: 0 <= T

▷ **Chrt** (output_control) ..... string(-array) ⇝ *string* / integer
Output tuple with strings that are separated by the number 0.

Example

```
read_serial (SerialHandle, 100, Data)
tuple_chrt (Data, Strings)
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_chr

See also

tuple_ord, tuple_ords, read_serial

Module

Foundation

tuple_int (: : T : Int)

Convert a tuple into a tuple of integer numbers.

tuple_int converts the input tuple T into a tuple of integer numbers by truncating T. The conversion of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_int, which can be used in an expression in the following syntax:

```
Int := int(T)
```

Parameters

- ▷ **T** (input_control) ..... number(-array) ⇝ *real* / integer
  Input tuple.
- ▷ **Int** (output_control) ..... number(-array) ⇝ *integer*
  Result of the conversion into integer numbers.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_round

Module

Foundation

tuple_number (: : T : Number)

Convert a tuple (of strings) into a tuple of numbers.

tuple_number converts the input tuple T into a tuple of numbers. If the input tuple contains numbers, they are simply copied into the output tuple. Strings are converted into the appropriate type of number (integer or floating point numbers) or are copied as strings if they do not represent a number. Note that strings starting with 0x are interpreted as hexadecimal numbers, and strings starting with 0 as octal numbers. For example, the string '20' is converted to the integer 20, '020' to 16, and '0x20' to 32.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

Exception: Out of range integers

If the input tuple contains strings with integers that cannot be represented as Hlong (32-bit signed integer for 32-bit HALCON, 64-bit signed integer for 64-bit HALCON), an exception is raised.

String padding

If tuple_number converts a string to a number, the same string with added leading and/or trailing spaces will be converted to the same number. Example: number('55.6') == 55.6 and number(' 55.6 ') == 55.6

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_number, which can be used in an expression in the following syntax:

```
Number := number(T)
```

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *string* / real / integer
Input tuple.

▷ **Number** (output_control) . . . . . tuple(-array) ⇝ *real* / integer / string
Input tuple as numbers.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

tuple_is_number, tuple_string

Module

Foundation

tuple_ord (: : T : Ord)

Convert a tuple of strings of length 1 into a tuple of integer numbers.

tuple_ord converts the input tuple T, which may only contain strings of length 1, into a tuple of integer numbers. When the encoding used in the HALCON library is UTF-8 (see set_system ('filename_encoding', 'utf8')) and the string operators are set to work by code points (see set_system('tuple_string_operator_mode', 'codepoint')), which is the default for both, the operator accepts the UTF-8 representation of a Unicode character (code point) and returns the appropriate Unicode character code. When the HALCON library encoding is set to *'locale'* or the string operator mode is *'byte'*, the operator accepts only a single byte for each input string. In that case the operator tuple_ord returns the ANSI code of the input byte as integer number between 0 and 256. See also Tuple / String Operations for a more detailed description of the different modes and further encoding issues.

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_ord, which can be used in an expression in the following syntax:

```
Ord := ord(T)
```

Parameters

▷ **T** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with strings of length 1.

▷ **Ord** (output_control) . . . . . integer(-array) ⇝ *integer*
Output tuple with Unicode character codes or ANSI codes of the characters passed in the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_ords

See also

tuple_chr, tuple_chrt

Module

Foundation

tuple_ords (: : T : Ords)

Convert a tuple of strings into a tuple of integer numbers.

tuple_ords converts the input tuple T, which may only contain strings and integer numbers, into a tuple of integer numbers. When the encoding used in the HALCON library is UTF-8 (see set_system('filename_encoding', 'utf8')) and the string operators are set to work by code points (see set_system('tuple_string_operator_mode', 'codepoint')), which is the default for both, the operator returns for the input strings the appropriate Unicode character codes. When the HALCON library encoding is set to *'locale'* or the string operator mode is *'byte'*, the operator returns the ANSI code for every byte of the input string. In that mode, the result may depend on the currently used code page for strings that contain non-ASCII characters. See also Tuple / String Operations for a more detailed description of the different modes and further encoding issues.

The character codes of the individual strings are written to the output according to their order within the string and within the tuple. Integer numbers are simply copied to an appropriate position in the output tuple. This operator can be used to prepare outputs with write_serial. In particular, a byte with value 0 can be written by inserting the integer number 0 into T.

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_ords, which can be used in an expression in the following syntax:

```
Ords := ords(T)
```

Parameters

- ▷ **T** (input_control) ..... string(-array) ⇝ *string* / integer
  Input tuple with strings.
- ▷ **Ords** (output_control) ..... integer(-array) ⇝ *integer*
  Output tuple with the Unicode character codes or ANSI codes of the input string.

Example

```
tuple_ords (['String 1', 0, 'String 2', 0], Data)
write_serial (SerialHandle, Data)
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_ord

See also

tuple_chr, tuple_chrt, write_serial

Module

Foundation

tuple_real (: : T : Real)

Convert a tuple into a tuple of floating point numbers.

tuple_real converts the input tuple T into a tuple of floating point numbers. The conversion of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_real, which can be used in an expression in the following syntax:

```
Real := real(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Real** (output_control) . . . . . number(-array) ⇝ *real*
Input tuple as floating point numbers.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

tuple_round (: : T : Round)

Convert a tuple into a tuple of integer numbers.

tuple_round converts the input tuple T into a tuple of integer numbers by rounding T to the nearest integer. The conversion of a string is not allowed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_round, which can be used in an expression in the following syntax:

```
Round := round(T)
```

Parameters

▷ **T** (input_control) . . . . . number(-array) ⇝ *real* / integer
Input tuple.

▷ **Round** (output_control) . . . . . number(-array) ⇝ *integer*
Result of the rounding.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_int

Module

Foundation

tuple_string (: : T, Format : String)

Convert a tuple into a tuple of strings.

tuple_string converts numbers into strings or modifies strings. The operator has two parameters: T is the number or string that has to be converted. Format specifies the conversion.

In the following, first some examples for the use of tuple_string are given and then, the structure of the Format string is explained in detail.

Examples

Examples for the conversion of numbers into strings:

T (Input)	Format (Input)	String (Output)
23	'10.2f'	' 23.00'
23	'-10.2f'	'23.00 '
4	'.7f'	'4.0000000'
1234.56789	'+10.3f'	' +1234.568'
255	'x'	'ff'
255	'X'	'FF'
0xff	'.5d'	'00255'

Examples for the modification of strings:

T (Input)	Format (Input)	String (Output)
'total'	'10s'	' total'
'total'	'-10s'	'total '
'total'	'-10.3s'	'tot '

Format string

The Format string consists of the following four parts:

```
<flags><field width>.<precision><conversion characters>
```

flags Zero or more flags, in any order, which modify the meaning of the conversion specification. Flags may consist of the following characters:

- **-** The result of the conversion is left justified within the field.
- **+** The result of a signed conversion always begins with a sign, + or −.
- <**space**> If the first character of a signed conversion is not a sign, a space character is prefixed to the result. This means that if the space flag (<space>) and + flag both appear, the space flag is ignored.
- **#** The value is to be converted to an "alternate form". For d and s conversions, this flag has no effect. For o conversion (see below), it increases the precision to force the first digit of the result to be a zero. For x or X conversion (see below), a non- zero result has 0x or 0X prefixed to it. For e, E, f, g, and G conversions, the result always contains a radix character, even if no digits follow the radix character. For g and G conversions, trailing zeros are not removed from the result, contrary to usual behavior.
- **0** The value should be zero padded. For d, o, u, x, X, e, E, f, F, g, and G conversions, the converted value is padded on the left with zeros rather than blanks. If the 0 and − flags both appear, the 0 flag is ignored. If a precision is given with a numeric conversion (d, o, u, x, and X), the 0 flag is ignored. For other conversions, the behavior is undefined.

field width An optional string of decimal digits to specify a minimum field width. For an output field, if the converted value has fewer characters than the field width, it is padded on the left (or right, if the left-adjustment flag, - has been given) to the field width.

precision The precision specifies the minimum number of digits to appear for the `d`, `o`, `x`, or `X` conversions (the field is padded with leading zeros), the number of digits to appear after the radix character for the `e` and `f` conversions, the maximum number of significant digits for the `g` conversion, or the maximum number of characters to be printed from a string in `s` conversion. The precision takes the form of a period **.** followed by a decimal digit string. A null digit string is treated as a zero.

conversion characters A conversion character indicates the type of conversion to be applied:

d,u,o,x,X The integer argument is printed in signed decimal (`d`), unsigned decimal (`u`), unsigned octal (`o`), or unsigned hexadecimal notation (`x` and `X`). The `x` conversion uses the numbers and letters `0123456789abcdef`, and the `X` conversion uses the numbers and letters `0123456789ABCDEF`. The precision component of the argument specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of 0 is no characters.

f The floating-point number argument is printed in decimal notation in the style `[-]dddrddd`, where the number of digits after the radix character, `r`, is equal to the precision specification. If the precision is omitted from the argument, six digits are output; if the precision is explicitly 0, no radix appears.

e,E The floating-point-number argument is printed in the style `[-]d.ddde±dd`, where there is one digit before the radix character, and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the precision is 0, no radix character appears. The `E` conversion character produces a number with `E` introducing the exponent instead of `e`. The exponent always contains at least two digits. However, if the value to be printed requires an exponent greater than two digits, additional exponent digits are printed as necessary.

g,G The floating-point-number argument is printed in style `f` or `e` (or in style `E` in the case of a `G` conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted; style `e` is used only if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit.

s The argument is taken to be a string, and characters from the string are printed until the end of the string or the number of characters indicated by the precision specification of the argument is reached. If the precision is omitted from the argument, it is interpreted as infinite and all characters up to the end of the string are printed.

b Similar to the `s` conversion specifier, except that the string can contain backslash-escape sequences which are then converted to the characters they represent.

In no case does a nonexistent or insufficient field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.

Exception: Empty input tuple

If any of the input tuples is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_string`, which can be used in an expression in the following syntax:

```
String := T $ Format
```

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string
Input tuple.

▷ **Format** (input_control) . . . . . string ⇝ *string*
Format string.

▷ **String** (output_control) . . . . . string(-array) ⇝ *string*
Input tuple converted to strings.

Example

```
*
* '     23.00'
tuple_string (23, '10.2f', String)
String := 23$'10.2f'
*
* '23.00     '
tuple_string (23, '-10.2f', String)
String := 23$'-10.2f'
*
* '4.0000000'
tuple_string (4, '.7f', String)
String := 4$'.7f'
*
* ' +1234.568'
tuple_string (1234.56789, '+10.3f', String)
String := 1234.56789$'+10.3f'
*
* 'ff'
tuple_string (255, 'x', String)
String := 255$'x'
*
* 'FF'
tuple_string (255, 'X', String)
String := 255$'X'
*
* '00255'
tuple_string (0xff, '.5d', String)
String := 0xff$'.5d'
*
* '     total'
tuple_string ('total', '10s', String)
String := 'total'$'10s'
*
* 'total     '
tuple_string ('total', '-10s', String)
String := 'total'$'-10s'
*
* 'tot       '
tuple_string ('total', '-10.3s', String)
String := 'total'$'-10.3s'
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sub

Module

Foundation

28.5 Creation

clear_handle (: : Handle :)

Clear the content of a handle.

clear_handle clears the content of the handle contained in Handle. The handle is left in a cleared state afterwards. Attempts to use it in any other operator will raise an error.

Usually, handles are automatically cleared once all references in all tuples are overwritten. clear_handle can be used to clear a handle at a specific time, even if there are still references to it.

This operator can be used to clear handles of any type instead of the more specialized operators, such as clear_matrix or clear_shape_model.

Parameters

▷ **Handle** (input_control) ..... tuple(-array) ⇝ *handle*
Handle to clear.

Result

If the parameters are valid, the operator clear_handle returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- Handle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

See also

tuple_is_valid_handle

Module

Foundation

tuple_concat (: : T1, T2 : Concat)

Concatenate two tuples to a new one.

tuple_concat concatenates the input tuples T1 and T2 to a new tuple Concat. The first elements of Concat conform to the elements of T1 and the remaining elements of Concat conform to those of T2.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_concat, which can be used in an expression in the following syntax:

```
Concat := [T1, T2]
```

Parameters

▷ **T1** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
Input tuple 1.

▷ **T2** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
Input tuple 2.

▷ **Concat** (output_control) ..... tuple(-array) ⇝ *integer* / real / string
Concatenation of input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_gen_const, tuple_str_bit_select, tuple_select, tuple_str_first_n, tuple_str_last_n, tuple_substr

See also

tuple_remove

Module

Foundation

tuple_constant (: : Name : Value)

Generate a tuple with the value of a HDevelop language constant.

tuple_constant generates a new tuple with the value of a HDevelop language constant.

Attention

H_INT64_MIN and H_INT64_MAX will both produce an error if 32-bit HALCON is being used, as they cannot be represented in the then 32-bit integer used in tuples.

H_INT_MIN and H_INT_MAX will produce different values depending on whether HALCON 32- or 64-bit is being used.

Parameters

▷ **Name** (input_control) ..... tuple ⇝ *string*
The name of the HDevelop language constant as string.
Default: 'H_INT32_MIN'
List of values: Name ∈ {'H_INT_MIN', 'H_INT_MAX', 'H_INT32_MIN', 'H_INT32_MAX', 'H_INT64_MIN', 'H_INT64_MAX', 'H_FLOAT32_MIN', 'H_FLOAT32_MAX', 'H_FLOAT32_MIN_POSITIVE', 'H_FLOAT32_EPSILON', 'H_FLOAT64_MIN', 'H_FLOAT64_MAX', 'H_FLOAT64_MIN_POSITIVE', 'H_FLOAT64_EPSILON', 'H_FLOAT_INFINITY', 'H_FLOAT_NEG_INFINITY', 'H_FLOAT_NAN', 'HNULL', 'H_MSG_TRUE', 'H_MSG_FALSE', 'H_MSG_VOID', 'H_MSG_FAIL', 'H_TYPE_INT', 'H_TYPE_REAL', 'H_TYPE_STRING', 'H_TYPE_HANDLE', 'H_TYPE_MIXED', 'H_TYPE_ANY', 'true', 'false'}

▷ **Value** (output_control) ..... tuple ⇝ *integer* / real / handle
The value of the constant.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

tuple_gen_const (: : Length, Const : Newtuple)

Generate a tuple of a specific length and initialize its elements.

tuple_gen_const generates a new tuple in Newtuple. The input parameter Length determines the number of elements for the new tuple. Thus, Length may only consist of a single number. If Length contains a floating point number, this may only represent an integer value (without fraction). The data type and value of each element of the new generated tuple is determined by the input parameter Const that may only consist of a single element. All elements in Newtuple have got the same data type and value as the single element in Const.

Exception: Empty input tuples

If any of the input tuples is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_gen_const`, which can be used in an expression in the following syntax:

```
Newtuple := gen_tuple_const(Length, Const)
```

Parameters

- ▷ **Length** (input_control) . . . . . number ⇝ *integer* / real
  Length of tuple to generate.
- ▷ **Const** (input_control) . . . . . tuple ⇝ *integer* / real / string / handle
  Constant for initializing the tuple elements.
- ▷ **Newtuple** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  New Tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_str_bit_select`, `tuple_select`, `tuple_str_first_n`, `tuple_str_last_n`, `tuple_concat`, `tuple_insert`, `tuple_replace`, `tuple_gen_sequence`

See also

`tuple_rand`

Module

Foundation

tuple_gen_sequence (: : Start, End, Step : Sequence)

Generate a tuple with a sequence of equidistant values.

`tuple_gen_sequence` generates a new tuple `Sequence` with a sequence of equidistant values:

[Start, Start + Step, Start + 2*Step, ... End]

`Step` must not be zero and the sign of (`End` - `Start`) must be equal to the sign of `Step`. The last entry of `Sequence` may actually be less than `End`, if (`End` - `Start`) is not divisible by `Step` without remainder.

Exception: Empty input tuples

If any of the input tuples is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_gen_sequence`, which can be used in an expression in the following syntax:

```
Sequence := [Start:Step:End]
```

An alternative syntax is `Sequence := [Start:End]`, where `Step` defaults to *1*.

Parameters

- ▷ **Start** (input_control) . . . . . number ⇝ *integer* / real
  Start value of the tuple.
- ▷ **End** (input_control) . . . . . number ⇝ *integer* / real
  Maximum value for the last entry.
- ▷ **Step** (input_control) . . . . . number ⇝ *integer* / real
  Increment value.
- ▷ **Sequence** (output_control) . . . . . number(-array) ⇝ *integer* / real
  The resulting sequence.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_str_bit_select, tuple_select, tuple_str_first_n, tuple_str_last_n, tuple_concat, tuple_insert, tuple_replace, tuple_gen_const

See also

tuple_rand

Module

Foundation

tuple_rand (: : Length : Rand)

Return a tuple of random numbers between 0 and 1.

tuple_rand returns a tuple of random numbers distributed uniformly in the interval [0,1). The parameter Length specifies the length of the output tuple, i.e., how many random numbers are generated.

The random numbers are generated using the C function "drand48()". See the parameter *'seed_rand'* of set_system for information on the used random seed.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_rand, which can be used in an expression in the following syntax:

```
Rand := rand(Length)
```

Parameters

▷ **Length** (input_control) ..... number ⇝ *integer*
Length of tuple to generate.

▷ **Rand** (output_control) ..... number(-array) ⇝ *real*
Tuple of random numbers.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

tuple_gen_const

Module

Foundation

tuple_repeat (: : Tuple, Num : Result)

Repeat a tuple.

tuple_repeat repeats Tuple Num times and returns them with Result. Thus, Result contains Num times more elements than Tuple. For example, the first Num elements of Result are element-wise equal to the elements of Tuple. Num must be positive and integer (also for type real). If Num is equal to 0, an empty tuple is returned.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_repeat`, which can be used in an expression in the following syntax:

```
Result := rep(Tuple, Num)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **Num** (input_control) . . . . . integer ⇝ *integer* / real
Number of repetitions.

▷ **Result** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Tuple with multiple copies.

Example

```
tuple_repeat (['a', 'b', 'c'], 2, Result)
* Returns ['a', 'b', 'c', 'a', 'b', 'c']
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_gen_const`, `tuple_concat`

See also

`tuple_repeat_elem`

Module

Foundation

tuple_repeat_elem (: : Tuple, Num : Result)

Repeat the elements of a tuple.

`tuple_repeat_elem` repeats each element of `Tuple` `Num` times and returns them with `Result`. Thus, `Result` contains `Num` times more elements than `Tuple`. For example, the first `Num` elements of `Result` are equal to the first element of `Tuple`. `Num` must be positive and integer (also for type `real`). If `Num` is equal to 0, an empty tuple is returned.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_repeat_elem`, which can be used in an expression in the following syntax:

```
Result := rep_elem(Tuple, Num)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **Num** (input_control) . . . . . integer ⇝ *integer* / real
Number of repetitions.

▷ **Result** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Tuple with repeated elements.

Example

```
tuple_repeat_elem (['a', 'b', 'c'], 2, Result)
* Returns ['a', 'a', 'b', 'b', 'c', 'c']
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_concat, tuple_gen_const

See also

tuple_repeat

Module

Foundation

28.6 Data Containers

copy_dict (: : DictHandle, GenParamName, GenParamValue : CopiedDictHandle)

Copy a dictionary.

copy_dict creates a deep copy (i.e., all contained handles and iconic objects are deep copied) of the dictionary DictHandle in the dictionary CopiedDictHandle.

If DictHandle contains a handle that can not be copied or that has been freed already, an exception is raised per default. This behavior is controlled by GenParamName *'raise_error_if_content_not_serializable'*, and the corresponding GenParamValue can take the following values:

'true': The default: Errors are raised and the copy process aborted.

'low_level': Only low level errors are raised. Instead of the handle concerned an empty handle is copied in CopiedDictHandle and the copy process will be continued. The behavior regarding HALCON low level errors is determined by *'do_low_error'* in set_system.

'false': The errors are suppressed. Instead of the handle concerned an empty handle is copied in CopiedDictHandle and the copy process will be continued.

Parameters

▷ **DictHandle** (input_control) . . . . . dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: GenParamName ∈ {'raise_error_if_content_not_serializable'}

▷ **GenParamValue** (input_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
Value of the generic parameter.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false', 'low_level'}

▷ **CopiedDictHandle** (output_control) . . . . . dict ⇝ *handle*
Copied dictionary handle.
Number of elements: CopiedDictHandle == 1

Result

If the parameters are valid, the operator `copy_dict` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`read_dict`, `deserialize_handle`, `create_dict`

Possible Successors

`write_dict`

See also

`read_dict`, `write_dict`, `serialize_handle`, `deserialize_handle`

Module

Foundation

`create_dict` (: : : DictHandle)

Create a new empty dictionary.

`create_dict` creates a new empty dictionary and returns it in `DictHandle`.

The dictionary serves as an associative array-like container allowing to store an arbitrary number of values associated with unique keys (integers or strings). Each key can refer either to a tuple or to an iconic object. These are stored in the dictionary using `set_dict_tuple` or `set_dict_object`, respectively, from where they can be retrieved again using `get_dict_tuple` or `get_dict_object`.

The data stored in the dictionary is always a copy of the original data, being it control parameters or objects. The original data can thus be reused immediately after the `set_dict_tuple` or `set_dict_object` calls. The following particularities apply:

- Objects: The copy is a reference, as in `copy_obj`. In particular, changes made using the operators `set_grayval` or `overpaint_region` affect the object stored in the dictionary as well.
- Handles: Storing any handle in the dictionary will copy the handle value, but not the resource behind the handle.

Multiple threads can add, retrieve and remove keys concurrently as long as every thread accesses a different key. As an exception to this rule, multiple threads can retrieve the same key from a dictionary simultaneously.

HDevelop In-line Operation

HDevelop provides an in-line operation for `create_dict`, which can be used in an expression in the following syntax:

```
DictHandle := dict{}
```

Parameters

▷ **DictHandle** (output_control) ..... dict ⇝ *handle*
Handle of the newly created dictionary.
Number of elements: DictHandle == 1

Example

```
Dicts := []
for idx := 0 to 4 by 1
  create_dict (DictHandle)
  Dicts[idx] := DictHandle
endfor
* ...
```

Result

Returns 2 (H_MSG_TRUE) unless a resource allocation error occurs.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

set_dict_tuple, set_dict_object

See also

set_dict_tuple, get_dict_tuple, set_dict_object, get_dict_object, get_dict_param

Module

Foundation

dict_to_json (: : DictHandle, GenParamName, GenParamValue : JsonString)

Transform a dictionary into a JSON string.

dict_to_json transforms the dictionary passed in DictHandle into a JSON string and returns that string in JsonString.

Several optional parameters can be set that control the creation of the JSON strings. Those parameters can be set via GenParamName and GenParamValue and are described in the documentation of write_dict:

- *'raise_error_if_content_not_serializable'*,
- *'compact_json'*,
- *'use_json_arrays'*.

dict_to_json can transform several dictionaries at once. For this, DictHandle must contain a tuple of dictionaries. The output JsonString will then contain one string per input dictionary.

Parameters

▷ **DictHandle** (input_control) . . . . . dict(-array) ⇝ *handle*
Dictionary handle.

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: GenParamName ∈ {'compact_json', 'raise_error_if_content_not_serializable', 'use_json_arrays'}

▷ **GenParamValue** (input_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
Value of the generic parameter.
Default: []
Suggested values: GenParamValue ∈ {'true', 'false'}

▷ **JsonString** (output_control) . . . . . string(-array) ⇝ *string*
String in JSON format.

Example

```
dict_to_json (dict{foo: 'bar', bar: [0:2]}, [], [], JSONString)
```

Result

If the parameters are valid, the operator `dict_to_json` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_dict`, `read_dict`, `json_to_dict`

Possible Successors

`json_to_dict`

Alternatives

`read_dict`, `create_dict`

See also

`json_to_dict`, `read_dict`, `write_dict`, `serialize_handle`, `deserialize_handle`, `copy_dict`

Module

Foundation

get_dict_object (: Object : DictHandle, Key :)

Retrieve an object associated with the key from the dictionary.

`get_dict_object` retrieves an object associated with the `Key` from the dictionary denoted by the `DictHandle`. The object has to be previously stored to the dictionary using `set_dict_object`.

The operator returns the data in the parameter `Object`. The iconic object is copied by the operation. Therefore, clearing or reusing the dictionary object afterwards will not have any side-effect on the returned iconic object.

If the given `Key` is not present in the dictionary or if the data associated with the key is not an object, `get_dict_object` fails. Presence of keys and information about the data associated with the key can be verified using `get_dict_param`.

HDevelop In-line Operation

HDevelop provides an in-line operation for `get_dict_object`, which can be used in an expression in the following syntax:

- Dynamic syntax:
  `Object := DictHandle.['Key']`
- Static syntax:
  `Object := DictHandle.Key`

Parameters

▷ **Object** (output_object) ..... object(-array) ⇝ *object*
Object value retrieved from the dictionary.

▷ **DictHandle** (input_control) ..... dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

▷ **Key** (input_control) ..... string ⇝ *string* / integer
Key string.
Number of elements: Key == 1
Restriction: length(Key) > 0

Example

```
* ...
get_dict_param (Dict, 'key_exists', ['simple_string','foo','my_image'], \
                KeysPresence)
get_dict_param (Dict, 'key_data_type', ['simple_string','my_image'], \
                KeysType)
get_dict_object (Image, Dict, 'my_image')
```

Result

If the operation succeeds, `get_dict_object` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised. Possible error conditions include invalid parameters (handle or key), the required key not found in the dictionary or other than object data associated with given key.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`set_dict_object`

Possible Successors

`get_dict_object`

Alternatives

`get_dict_tuple`

See also

`create_dict`, `set_dict_tuple`, `get_dict_tuple`, `set_dict_object`, `remove_dict_key`, `get_dict_param`

Module

Foundation

get_dict_param (: : DictHandle, GenParamName, Key : GenParamValue)

Query dictionary parameters or information about a dictionary.

`get_dict_param` queries current values of dictionary parameters or other information about the dictionary status.

With each call of `get_dict_param`, only a single parameter value can be queried. However, there are two types of parameters/queries:

- Parameters/queries applicable to the entire dictionary. In this case no keys must be specified, the parameter `Key` must be an empty tuple.
- Parameters/queries applicable to the individual keys. In this case a list of keys must be specified in the parameter `Key`. The keys are processed in the same order as specified in the `Key` parameter.

Key-independent (global) parameter names:

'keys': Queries all the keys stored in the dictionary, no matter whether they are associated with tuple or object data. The list of keys is reported as a string tuple via `GenParamValue`. For this query the parameter `Key` must be an empty tuple.

Currently supported key-specific parameter names are:

'key_exists': Reports *1* if the given key is stored in the dictionary, *0* otherwise. The results are reported via `GenParamValue`, one value for each key.

'key_data_type': Reports *'tuple'* for keys associated with tuple data within the dictionary (the data can be retrieved using `get_dict_tuple`). Reports *'object'* for keys associated with object data (the data can be retrieved using `get_dict_object`). The results are reported via `GenParamValue`, one value for each key. This parameter is useful to decide dynamically whether to use `get_dict_tuple` or `get_dict_object` to get the data of a specific key.

Parameters

▷ **`DictHandle`** (input_control) . . . . . dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

▷ **`GenParamName`** (input_control) . . . . . attribute.name ⇝ *string*
Names of the dictionary parameters or info queries.
Number of elements: GenParamName == GenParamValue
Default: 'keys'
List of values: `GenParamName` ∈ {'keys', 'key_exists', 'key_data_type'}

▷ **`Key`** (input_control) . . . . . string(-array) ⇝ *string* / integer
Dictionary keys the parameter/query should be applied to (empty for `GenParamName` = *'keys'*).
Default: []

▷ **`GenParamValue`** (output_control) . . . . . attribute.value(-array) ⇝ *string* / integer / real
Values of the dictionary parameters or info queries.

Example

```
get_dict_param (Dict, 'keys', [], AllKeys)
get_dict_param (Dict, 'key_data_type', AllKeys, KeysType)
```

Result

If all the operator parameters, and the specified keys are valid, `get_dict_param` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`create_dict`

See also

`create_dict`, `set_dict_tuple`, `get_dict_tuple`, `set_dict_object`, `get_dict_object`, `remove_dict_key`

Module

Foundation

`get_dict_tuple ( : : DictHandle, Key : Tuple )`

Retrieve a tuple associated with the key from the dictionary.

`get_dict_tuple` retrieves a tuple associated with the `Key` from the dictionary denoted by the `DictHandle`. The tuple has to be previously stored to the dictionary using `set_dict_tuple`.

The operator returns the data in the parameter `Tuple`. The data including strings is copied by the operation, the dictionary can thus be immediately reused.

If the given `Key` is not present in the dictionary or if the data associated with the key is not a tuple, `get_dict_tuple` fails. Presence of keys and information about the data associated with the key can be verified using `get_dict_param`.

Obtaining the values for multiple keys at once is only possible if for each of those keys, the dictionary contains a tuple with a single element.

HDevelop In-line Operation

HDevelop provides an in-line operation for `get_dict_tuple`, which can be used in an expression in the following syntax:

- Dynamic syntax:
  `Tuple := DictHandle.['Key']`
- Static syntax:
  `Tuple := DictHandle.Key`

Parameters

▷ **`DictHandle`** (input_control) . . . . . dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

▷ **`Key`** (input_control) . . . . . string(-array) ⇝ *string* / integer
Key string.

▷ **`Tuple`** (output_control) . . . . . tuple(-array) ⇝ *string* / integer / real
Tuple value retrieved from the dictionary.

Example

```
* ...
get_dict_param (Dict, 'key_exists', ['simple_string','foo','my_image'], \
                KeysPresence)
get_dict_param (Dict, 'key_data_type', ['simple_string','my_image'], \
                KeysType)
get_dict_tuple (Dict, 'simple_string', TupleString)
*
* Access multiple keys, if their tuple all have length 1
get_dict_tuple (Dict, ['key1','key2','key3'], MultipleValues)
```

Result

If the operation succeeds, `get_dict_tuple` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised. Possible error conditions include invalid parameters (handle or key), the required key not found in the dictionary, or other than tuple data associated with given key.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`set_dict_tuple`

Possible Successors

`get_dict_tuple`, `set_dict_tuple`, `set_dict_tuple_at`

Alternatives

`get_dict_object`

See also

`create_dict`, `set_dict_tuple`, `set_dict_object`, `get_dict_object`, `remove_dict_key`, `get_dict_param`

Module

Foundation

json_to_dict (: : JsonString, GenParamName, GenParamValue : DictHandle)

Transform a JSON string into a dictionary.

json_to_dict transforms the string passed in JsonString, which must be valid JSON, into a dictionary and returns the handle of the dictionary in DictHandle.

Several optional parameters can be set that control the values of JSON constants. Those parameters can be set via GenParamName and GenParamValue and are described in the documentation of read_dict.

json_to_dict can transform several strings at once. For this, JsonString must contain a tuple of strings. The output DictHandle will then contain one dictionary per input string.

Parameters

- ▷ **JsonString** (input_control) . . . . . string(-array) ⇝ *string*
  String in JSON format.
  Default: ['"key":"value"']
- ▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
  Name of the generic parameter.
  Default: []
  List of values: GenParamName ∈ {'json_value_true', 'json_value_false', 'json_value_null', 'convert_json_arrays_to'}
- ▷ **GenParamValue** (input_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
  Value of the generic parameter.
  Default: []
  Suggested values: GenParamValue ∈ {0, 1, 'HNULL', 'true', 'false', 'dict', 'tuple', 'tuple_if_possible'}
- ▷ **DictHandle** (output_control) . . . . . dict(-array) ⇝ *handle*
  Dictionary handle.

Example

```
json_to_dict ('{"capacity": "medium", "image_height": 1024, "image_width":
1024}', [], [], DictHandle)
```

Result

If the parameters are valid, the operator json_to_dict returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. This is especially the case if JsonString does not contain valid JSON.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Successors

write_dict, copy_dict

Alternatives

read_dict, create_dict

See also

read_dict, write_dict, serialize_handle, deserialize_handle

Module

Foundation

read_dict (: : FileName, GenParamName, GenParamValue : DictHandle)

Read a dictionary from a file.

read_dict reads a dictionary from the file FileName and returns the handle of the dictionary in DictHandle.

The operator supports the following file formats:

'hdict': Binary HALCON format for dictionaries. Files with this format can be written by write_dict. The default file extension for this format is 'hdict'.

'json': JSON (JavaScript Object Notation) file format. The default file extension for this format is 'json'.

If the given file does not exist, the operator attempts to find the file by appending the default file extensions to the filename. The file type is automatically recognized based on the file content and the file ending (which have to be consistent).

A set of additional optional parameters can be set. The names and values of the parameters are passed in GenParamName and GenParamValue, respectively. Some of the optional parameters can only be set for a certain file type. The following values for GenParamName are possible:

'json_value_true', *'json_value_false'*, *'json_value_null'*: Set the values that are used for JSON primitives. Valid JSON primitives are 'true', 'false', and 'null'. When encountering such a primitive in a JSON file, the corresponding value in the read dictionary is set to the value defined with these parameters. The value passed in GenParamValue must be a tuple of length 1.

The default values for primitives are *1* for 'true', *0* for 'false', and *'HNULL'* for 'null'. These parameters only have an effect when reading a JSON file.

'convert_json_arrays_to': This parameter controls which HALCON-internal data type is used to represent JSON arrays. Possible values for GenParamValue are:

'dict' **(default):** JSON arrays are converted into HALCON-Dictionaries with consecutive integer keys starting at *0*. This is always possible, even if the JSON array contains further JSON arrays. However, using dictionaries might require more memory than storing the same data in HALCON tuples.

'tuple': JSON arrays are converted into HALCON tuples. If this is not possible, for example if an array contains an array, an exception is raised.

'tuple_if_possible': JSON arrays are converted into HALCON tuples if possible, and into HALCON dictionaries otherwise.

Note that when using this option, the structure of the returned dictionary can change even if the JSON data follows the same schema. For example, an array of 2D point coordinates of the form *'"pt": [[1,2], [3,4]]'* would be converted into an outer dictionary which contains two tuples. However, if under the same schema no points are contained in the array, the corresponding entry in the JSON of the form *'"pt": []'* would be converted into a tuple instead of a dictionary. Code using the read dictionary must be prepared to deal with this kind of change in types.

Parameters

▷ **FileName** (input_control) . . . . . filename.read ⇝ *string*
File name.
File extension: .hdict, .json

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: GenParamName ∈ {'json_value_true', 'json_value_false', 'json_value_null', 'convert_json_arrays_to'}

▷ **GenParamValue** (input_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
Value of the generic parameter.
Default: []
Suggested values: GenParamValue ∈ {0, 1, 'HNULL', 'true', 'false', 'dict', 'tuple', 'tuple_if_possible'}

▷ **DictHandle** (output_control) . . . . . dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

Result

If the parameters are valid, the operator read_dict returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator returns a handle. Note that the state of an instance of this handle type may be changed by specific operators even though the handle is used as an input parameter by those operators.

Possible Predecessors

write_dict

Alternatives

json_to_dict, create_dict

See also

write_dict, serialize_handle, deserialize_handle

Module

Foundation

remove_dict_key (: : DictHandle, Key :)

Remove keys from a dictionary.

remove_dict_key removes the keys specified in Key from the dictionary passed in DictHandle and releases all the (tuple or object) data associated with those keys.

If an error occurs while processing one or more keys (in particular if a key is invalid), the operator attempts to continue removing as many keys as possible before reporting an appropriate failure.

Parameters

▷ **DictHandle** (input_control) . . . . . dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

▷ **Key** (input_control) . . . . . string(-array) ⇝ *string* / integer
Key to remove.
Restriction: length(Key) > 0

Example

```
* Remove all keys
get_dict_param (Dict, 'keys', [], Keys)
remove_dict_key (Dict, Keys)
```

Result

If all the operator parameters are valid, remove_dict_key returns 2 (H_MSG_TRUE). Otherwise an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DictHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_dict

Possible Successors

get_dict_param

See also

create_dict, set_dict_tuple, get_dict_tuple, set_dict_object, get_dict_object, get_dict_param

Module

Foundation

set_dict_object (Object : : DictHandle, Key :)

Add a key/object pair to the dictionary.

set_dict_object stores an object associated with a key in the dictionary. The dictionary is denoted by the DictHandle parameter.

Object is copied by the operation and can thus be immediately reused. Thereby the object data is copied in HALCON's object database, meaning the new object contains a reference to Object, see copy_obj.

Both an empty object or an object tuple are considered as a valid value that can be associated with the key. If any data (tuple or object) was already associated with given key (Key), the old data is destroyed by set_dict_object and replaced by Object.

The Key has to be a string or an integer. Strings are treated case sensitive.

The object data for the given key can be retrieved again from the dictionary using get_dict_object.

HDevelop In-line Operation

HDevelop provides an in-line operation for set_dict_object, which can be used in an expression in the following syntax:

- Dynamic syntax:
  DictHandle.['Key'] := Object
- Static syntax:
  DictHandle.Key := Object

Parameters

▷ **Object** (input_object) ..... object(-array) ⇝ *object*
Object to be associated with the key.

▷ **DictHandle** (input_control) ..... dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

▷ **Key** (input_control) ..... string ⇝ *string* / integer
Key string.
Number of elements: Key == 1
Restriction: length(Key) > 0

Example

```
create_dict (Dict)
read_image (Image, 'filename')
set_dict_object (Image, Dict, 'my_image')
```

Result

If the operation succeeds, set_dict_object returns 2 (H_MSG_TRUE). Otherwise an exception is raised. Possible error conditions include invalid parameters (handle or key) or resource allocation error.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DictHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_dict

Possible Successors

set_dict_object, set_dict_tuple

Alternatives

set_dict_tuple

See also

create_dict, set_dict_tuple, get_dict_tuple, get_dict_object, remove_dict_key, get_dict_param

Module

Foundation

set_dict_tuple (: : DictHandle, Key, Tuple :)

Add a key/tuple pair to the dictionary.

set_dict_tuple stores a tuple associated with a key in the dictionary. The dictionary is denoted by the DictHandle parameter.

Tuple including strings is copied by the operation, and can thus be immediately reused. An empty tuple is considered as a valid value that can be associated with the key. If any data (tuple or object) was already associated with given key (Key), the old data is destroyed by set_dict_tuple and replaced by Tuple.

The Key has to be a string or an integer. Strings are treated case sensitive.

The tuple data for the given key can be retrieved again from the dictionary using get_dict_tuple.

set_dict_tuple allows setting the values of multiple keys with a single call. In this case, the length of Tuple must either be equal to the number of keys or *1*. In the first case, Tuple is split into segments of length *1*, one for each key. In the second case, if Tuple has length *1*, that one value is associated with each key. If no keys are passed, the values in Tuple are ignored. The following table summarizes the possible combinations of number of keys and values, where N is an arbitrary non-negative integer:

Length of Key	Length of Tuple	Effect
N	1	Value in Tuple is associated with all keys
1	N	Associate Tuple with Key
N	N	Associate tuples of length *1* with each passed key

HDevelop In-line Operation

HDevelop provides an in-line operation for set_dict_tuple, which can be used in an expression in the following syntax:

- Dynamic syntax:
  ```
  DictHandle.['Key'] := ['The answer', 42]
  ```
- Static syntax:
  ```
  DictHandle.Key := ['The answer', 42]
  ```

Attention

If the tuple contains any handles only the handle values are copied by the operation, not the resources behind those handles (no deep copy is created).

Parameters

▷ **DictHandle** (input_control) . . . . . dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

▷ **Key** (input_control) . . . . . string(-array) ⇝ *string* / integer
Key string.

▷ **Tuple** (input_control) . . . . . tuple-array ⇝ *string* / integer / real / handle
Tuple value to be associated with the key.

Example

```
create_dict (Dict)
set_dict_tuple (Dict, 'simple_integer', 27)
set_dict_tuple (Dict, 'simple_string', 'Hello world')
set_dict_tuple (Dict, 'mixed_tuple', ['The answer', 42])
set_dict_tuple (Dict, 0, 'This is zero')
```

Result

If the operation succeeds, `set_dict_tuple` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised. Possible error conditions include invalid parameters (handle or key) or resource allocation error.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DictHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

`create_dict`

Possible Successors

`set_dict_tuple`, `set_dict_object`

Alternatives

`set_dict_object`, `set_dict_tuple_at`

See also

`create_dict`, `set_dict_tuple_at`, `get_dict_tuple`, `set_dict_object`, `get_dict_object`, `get_dict_param`, `remove_dict_key`

Module

Foundation

set_dict_tuple_at (: : DictHandle, Key, Index, Value :)

Assignment of one or several values to one or several tuple elements in a dictionary

`set_dict_tuple_at` assigns a single value to one or several elements of a tuple, or it assigns a number of values elementwise to the specified elements of a tuple that is stored in `DictHandle` under the key `Key`. The operator allows to efficiently replace parts of a tuple stored in a dictionary without first reading the previous values from the dictionary.

If the dictionary does not yet contain a tuple under the given key, or if the given key maps to an iconic value, a new tuple is created and stored under the given key. In the latter case, the iconic object previously stored under that key is removed from the dictionary.

If the passed indices are out of the current range of the tuple stored in the dictionary, the tuple length is increased and the new values are initialized to a default value.

The `Key` has to be a string or an integer. Strings are treated case sensitive. The `Index` parameter can be any expression that evaluates to any number of positive integer values. The `Value` parameter must evaluate to exactly one value or to the same number of indices that are provided via the `Index` parameter.:

The tuple data for the given key can be retrieved again from the dictionary using `get_dict_tuple`.

Attention

Note that if `Value` contains any handles, only the handle values are copied by the operation, not the resources behind those handles.

Parameters

▷ **`DictHandle`** (input_control) . . . . . dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

▷ **`Key`** (input_control) . . . . . string ⇝ *string* / integer
Key string.
Number of elements: Key == 1
Restriction: length(Key) > 0

▷ **`Index`** (input_control) . . . . . integer-array ⇝ *integer*
Indices of the elements that have to be replaced by the new value(s).
Default: 0
Suggested values: `Index` ∈ {0, 1, 2, 3, 4, 5, 6}
Minimum increment: 1

▷ **`Value`** (input_control) . . . . . tuple-array ⇝ *string* / integer / real / handle
Value(s) that is to be assigned.

Example

```
Dict := dict{}
Dict.some_key := 27

set_dict_tuple_at (Dict, 'some_key', 1, 5)
* Dict.some_key is now [27, 5]

* Alternative TRIAS syntax
Dict.some_key[0] := 66
* Dict.some_key is now [66, 5]
```

Result

If the operation succeeds, `set_dict_tuple_at` returns `2` (`H_MSG_TRUE`). Otherwise an exception is raised. Possible error conditions include invalid parameters or resource allocation error.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

This operator modifies the state of the following input parameter:

- DictHandle

During execution of this operator, access to the value of this parameter must be synchronized if it is used across multiple threads.

Possible Predecessors

create_dict

Possible Successors

set_dict_tuple, get_dict_tuple

Alternatives

set_dict_tuple

See also

create_dict, set_dict_tuple, get_dict_tuple, set_dict_object, get_dict_object, get_dict_param, remove_dict_key

Module

Foundation

write_dict (: : DictHandle, FileName, GenParamName, GenParamValue :)

Write a dictionary to a file.

write_dict writes the dictionary denoted by the DictHandle to the file FileName.

The operator supports the file formats *'hdict'* and *'json'* (see read_dict). The format is selected based on the filename ending or by using the generic parameter *'file_type'* (see below). If neither is conclusive or set, the file is written as *'hdict'*.

Note that due to restrictions of the *'json'*-format, some information contained in dictionaries can not be stored in that file format. This includes iconic objects and handles that are not dictionaries. By default, write_dict will return an error if the dictionary contains such an item. This behavior can be changed using the generic parameter *'raise_error_if_content_not_serializable'* (see below).

A set of additional optional parameters can be set. The names and values of the parameters are passed in GenParamName and GenParamValue, respectively. Some of the optional parameters can only be set for a certain file type. The following values for GenParamName are possible:

'raise_error_if_content_not_serializable': If DictHandle contains an item that can not be serialized, an exception is raised per default. This behavior is controlled by this parameter, for which GenParamValue can take the following values:

- *'true'*: The default behavior, errors are raised.
- *'false'*: The errors are suppressed. Depending on the file type an empty handle (*'hdict'*) or no entry at all (*'json'*) are written in place of the non serializable item.
- *'low_level'*: The file output is the same as for *'false'*. Additionally, low level errors are raised if non-serializable items are encountered. The behavior regarding HALCON low level errors is determined by *'do_low_error'* in set_system.

'file_type': Sets the file type to be used. The corresponding entry in GenParamValue must be set to *'hdict'* (default) or *'json'*. If this parameter is not set, the file type is determined based on the ending of the file name. If that ending is not conclusive, the default *'hdict'* is used.

'compact_json': Controls if JSON files are written in a compact format without additional whitespace, or human-readable with newlines and indentation. In both cases the file will contain the same information. The corresponding entry in GenParamValue must be set to *'true'* (default) or *'false'*.

'use_json_arrays': Controls if write_dict should convert dictionaries into native JSON arrays where possible. For this, the dictionaries must only contain ascending integer keys starting at zero. The corresponding entry in GenParamValue must be set to *'true'* (default) or *'false'*.

Parameters

▷ **DictHandle** (input_control) . . . . . dict ⇝ *handle*
Dictionary handle.
Number of elements: DictHandle == 1

▷ **FileName** (input_control) . . . . . filename.write ⇝ *string*
File name.
File extension: .hdict

▷ **GenParamName** (input_control) . . . . . attribute.name(-array) ⇝ *string*
Name of the generic parameter.
Default: []
List of values: GenParamName ∈ {'file_type', 'raise_error_if_content_not_serializable', 'compact_json', 'use_json_arrays'}

▷ **GenParamValue** (input_control) . . . . . attribute.name(-array) ⇝ *string* / integer / real
Value of the generic parameter.
Default: []
Suggested values: GenParamValue ∈ {'hdict', 'json', 'true', 'false', 'low_level'}

Result

If the parameters are valid, the operator write_dict returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

read_dict, deserialize_handle, create_dict

Possible Successors

read_dict, create_dict

See also

read_dict, serialize_handle, deserialize_handle, dict_to_json

Module

Foundation

28.7 Element Order

tuple_inverse (: : Tuple : Inverted)

Invert a tuple.

tuple_inverse inverts the input tuple Tuple. Thus, Inverted contains the same elements as Tuple but with the reverse order.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_inverse, which can be used in an expression in the following syntax:

```
Inverted := inverse(Tuple)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple.

▷ **Inverted** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Inverted input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sort, tuple_sort_index

Module

Foundation

tuple_sort (: : Tuple : Sorted)

Sort the elements of a tuple in ascending order.

tuple_sort sorts all elements of Tuple in ascending order and returns the result with Sorted. As a precondition the single elements of Tuple must be comparable. Thus, Tuple must either exclusively consist of strings or it must only contain (integer or floating point) numbers. In the latter case integers and floating point numbers may be mixed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_sort, which can be used in an expression in the following syntax:

```
Sorted := sort(Tuple)
```

Parameters

▷ **Tuple** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
Input tuple.

▷ **Sorted** (output_control) ..... tuple(-array) ⇝ *integer* / real / string
Sorted tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

tuple_sort_index, tuple_inverse

Module

Foundation

tuple_sort_index (: : Tuple : Indices)

Sort the elements of a tuple and return the indices of the sorted tuple.

tuple_sort_index sorts all elements of Tuple in ascending order and returns the indices of the elements of the sorted tuple (in relation to the input tuple Tuple) with Indices. As a precondition the single elements of Tuple must be comparable. Thus, Tuple must either exclusively consist of strings or it must only contain (integer or floating point) numbers. In the latter case integers and floating point numbers may be mixed.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_sort_index, which can be used in an expression in the following syntax:

```
Indices := sort_index(Tuple)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple.

▷ **Indices** (output_control) . . . . . integer(-array) ⇝ *integer*
Sorted tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sort, tuple_inverse

Module

Foundation

28.8 Features

get_handle_object (: Object : Handle, Key :)

Retrieve an object associated with a key from a handle.

get_handle_object retrieves an iconic object associated with the Key from Handle and returns it in Object. The list of keys supported by a handle can be queried with get_handle_param.

Note that this operator is provided only for implementing generic debug and inspection mechanisms. The keys returned for certain handle types can change without notice. Additionally, some handle types might not be supported by this operator. Also, this operator is not optimized for performance. To obtain more reliable information about a handle in a program, it is strongly recommended to use the operators specialized for the particular handle type, such as get_object_model_3d_params, get_generic_shape_model_param etc.

Parameters

▷ **Object** (output_object) . . . . . object(-array) ⇝ *object*
Iconic value of the key.

▷ **Handle** (input_control) . . . . . handle ⇝ *handle*
Handle of which to get the key.

▷ **Key** (input_control) . . . . . string ⇝ *string* / integer
Key to get.

Result

If the parameters are valid, the operator get_handle_object returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

get_handle_tuple

See also

get_handle_param

Module

Foundation

get_handle_param (: : Handle, GenParamName, Key : GenParamValue)

Return information about a handle.

get_handle_param returns details about Handle in GenParamValue. It provides a generic mechanism for inspecting and debugging handles of any type by returning keys of that handle that can later be queried using get_handle_tuple and get_handle_object.

Note that this operator is provided only for implementing generic debug and inspection mechanisms. The keys returned for certain handle types can change without notice. Additionally, some handle types might not be supported by this operator. Also, this operator is not optimized for performance. To obtain more reliable information about a handle in a program, it is strongly recommended to use the operators specialized for the particular handle type, such as get_object_model_3d_params, get_generic_shape_model_param etc.

Depending on the value of GenParamName, the following values can be queried:

'keys': Returns a list of keys that this handle supports in GenParamValue. Each key contains either a tuple or an iconic object. For this query, the parameter Key must be empty. Note that due to the generic nature of this operator, some of the returned keys might not be valid for the particular handle. In this case, get_handle_tuple or get_handle_object will return an error for those keys.

'key_data_type': Returns the data type of Key. The type is returned in GenParamValue and is either *'tuple'* or *'object'*. Depending on the type, the key's corresponding value can be queried with get_handle_tuple or get_handle_object, respectively.

Parameters

▷ **Handle** (input_control) . . . . . handle ⇝ *handle*
Handle of which to get the parameter.

▷ **GenParamName** (input_control) . . . . . string ⇝ *string*
Parameter to get.
Default: 'keys'
List of values: GenParamName ∈ {'keys', 'key_data_type'}

▷ **Key** (input_control) . . . . . string(-array) ⇝ *string*
Optional key.
Default: []

▷ **GenParamValue** (output_control) . . . . . tuple(-array) ⇝ *string* / integer / real
Returned value.

Result

If the parameters are valid, the operator get_handle_param returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

clear_handle, get_handle_tuple, get_handle_object

See also

get_handle_tuple, get_handle_object

Module

Foundation

get_handle_tuple (: : Handle, Key : Tuple)

Retrieve a tuple associated with a key from a handle.

get_handle_tuple retrieves a tuple associated with the Key from Handle and returns it in Tuple. The list of keys supported by a handle can be queried with get_handle_param.

Note that this operator is provided only for implementing generic debug and inspection mechanisms. The keys returned for certain handle types can change without notice. Additionally, some handle types might not be supported by this operator. Also, this operator is not optimized for performance. To obtain more reliable information about a handle in a program, it is strongly recommended to use the operators specialized for the particular handle type, such as get_object_model_3d_params, get_generic_shape_model_param etc.

Parameters

▷ **Handle** (input_control) . . . . . handle ⇝ *handle*
Handle of which to get the key.

▷ **Key** (input_control) . . . . . string ⇝ *string* / integer
Key to get.
Number of elements: Key == 1
Restriction: length(Key) > 0

▷ **Tuple** (output_control) . . . . . tuple(-array) ⇝ *string* / integer / real
Control value of the key.

Result

If the parameters are valid, the operator get_handle_tuple returns the value 2 (H_MSG_TRUE). If the handle is invalid or does not support the given key, or if the key references an iconic object instead of a tuple, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_handle_param

Alternatives

get_handle_object

See also

get_handle_param, get_handle_object

Module

Foundation

tuple_deviation (: : Tuple : Deviation)

Return the standard deviation of the elements of a tuple.

tuple_deviation calculates the standard deviation of all elements of the input tuple Tuple. It returns the deviation as a floating point number in the output parameter Deviation. The input tuple may only consist of numbers (integer or floating point numbers).

Exception: Empty input tuple

If the input tuple is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_deviation, which can be used in an expression in the following syntax:

```
Deviation := deviation(Tuple)
```

Parameters

▷ **Tuple** (input_control) . . . . . number(-array) ⇝ *integer* / real
Input tuple.

▷ **Deviation** (output_control) . . . . . number(-array) ⇝ *real*
Standard deviation of tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_mean, tuple_sum, tuple_min, tuple_max, tuple_length, tuple_median

Module

Foundation

tuple_histo_range (: : Tuple, Min, Max, NumBins : Histo, BinSize)

Calculate the value distribution of a tuple within a certain value range.

tuple_histo_range calculates the value distribution Histo of the Tuple within the value range [Min, Max]. The values for Min and Max are rounded down to the next integer if all entries of the Tuple are of type integer. The value range is divided into NumBins bins of the same size, which is returned in BinSize. If a value lies between two bins the value is assigned to the smaller bin. If the Tuple has entries of type real $\texttt{BinSize} = \frac{\texttt{Max}-\texttt{Min}}{\texttt{NumBins}}$. If all entries are of type integer the size of a bin is computed with $\texttt{BinSize} = \frac{\texttt{Max}-\texttt{Min}+1}{\texttt{NumBins}}$.

Exception: Empty input tuples

If any of the input tuples is empty, an exception is raised.

Attention

If all the data of the Tuple are of type integer the value of BinSize may cause the following effects: For $\texttt{BinSize} > 1$ multiple consecutive numbers are assigned to the same bin. If BinSize is no integer the numbers are distributed uneven among the bins, e.g., for $\texttt{BinSize} = 1.5$ the first and second number are assigned to the first bin, the third number is assigned to the second bin, and the fourth and fifth number are assigned to the third bin. This becomes noticeable in several peeks in the histogram Histo. If $\texttt{BinSize} < 1$ some classes are not assigned by any number, e.g., for $\texttt{BinSize} = 0.5$ the first number is assigned to the first bin and the second number is assigned to the third bin. The histogram Histo shows some gaps, which resembles the structure of a comb.

If the Tuple has entries of type real and $\texttt{Min} = \texttt{Max}$, all entries of the corresponding value are assigned only to the first bin.

Parameters

▷ **Tuple** (input_control) . . . . . number-array ⇝ *real* / integer
Input tuple.

▷ **Min** (input_control) . . . . . number ⇝ *real* / integer
Minimum value.

▷ **Max** (input_control) . . . . . number ⇝ *real* / integer
Maximum value.
Restriction: Max >= Min

▷ **NumBins** (input_control) . . . . . integer ⇝ *integer*
Number of bins.
Restriction: NumBins >= 1

▷ **Histo** (output_control) . . . . . histogram(-array) ⇝ *integer*
Histogram to be calculated.

▷ **BinSize** (output_control) . . . . . real ⇝ *real*
Bin size.

Result

If the parameters are valid, the operator `tuple_histo_range` returns the value `2` (`H_MSG_TRUE`). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`tuple_min`, `tuple_max`

Possible Successors

`create_funct_1d_array`

See also

`gray_histo`, `gray_histo_abs`, `gray_histo_range`

Module

Foundation

tuple_length (: : Tuple : Length)

Return the number of elements of a tuple.

`tuple_length` returns the number of elements of the input tuple `Tuple`.

Exception: Empty input tuple

If the input tuple is empty, the operator returns 0.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_length`, which can be used in an expression in the following syntax:

```
Length := |Tuple|
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple.

▷ **Length** (output_control) . . . . . integer ⇝ *integer*
Number of elements of input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_min`, `tuple_max`, `tuple_mean`, `tuple_deviation`, `tuple_sum`, `tuple_median`

See also

`tuple_type`

Module

Foundation

tuple_max (: : Tuple : Max)

Return the maximal element of a tuple.

tuple_max returns the maximal element of all elements of the input tuple Tuple. All elements of Tuple either have to be strings or numbers (integer or floating point numbers). It is not allowed to mix strings with numerical values. The result parameter Max will contain a floating point number, if at least one element of Tuple is a floating point number. If all elements of Tuple are integer numbers the resulting maximum in Max will also be an integer number.

Exception: Empty input tuple

If the input tuple is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_max, which can be used in an expression in the following syntax:

```
Max := max(Tuple)
```

Parameters

- ▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Input tuple.
- ▷ **Max** (output_control) . . . . . tuple(-array) ⇝ *real* / integer / string
  Maximal element of the input tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_min, tuple_mean, tuple_deviation, tuple_sum, tuple_length, tuple_median

See also

tuple_max2, tuple_min2

Module

Foundation

tuple_mean (: : Tuple : Mean)

Return the mean value of a tuple of numbers.

tuple_mean returns the mean value of all elements of the input tuple Tuple as a floating point number in the output parameter Mean. The input tuple may only consist of numbers (integer or floating point numbers).

Exception: Empty input tuple

If the input tuple is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_mean, which can be used in an expression in the following syntax:

```
Mean := mean(Tuple)
```

Parameters

- ▷ **Tuple** (input_control) . . . . . number(-array) ⇝ *integer* / real
  Input tuple.
- ▷ **Mean** (output_control) . . . . . number ⇝ *real*
  Mean value of tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_deviation, tuple_sum, tuple_min, tuple_max, tuple_length, tuple_median

Module

Foundation

tuple_median (: : Tuple : Median)

Return the median of the elements of a tuple.

tuple_median calculates the median of all elements of the input tuple Tuple and returns it in the output parameter Median. The input tuple may only consist of numbers (integer or floating point numbers). The median is defined as the element with rank n/2 (see tuple_select_rank).

Note that for an even number of input elements, this will return the upper median, not the arithmetic median. To calculate the arithmetic median, you can use tuple_select_rank to select the upper and lower median values and then calculate the arithmetic mean of both values.

Exception: Empty input tuple

If the input tuple is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_median, which can be used in an expression in the following syntax:

```
Median := median(Tuple)
```

Parameters

▷ **Tuple** (input_control) ..... number(-array) ⇝ *integer* / real
Input tuple.

▷ **Median** (output_control) ..... number ⇝ *integer* / real
Median of the tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_select_rank

See also

tuple_mean, tuple_min, tuple_max

Module

Foundation

tuple_min (: : Tuple : Min)

Return the minimal element of a tuple.

tuple_min returns the minimal element of all elements of the input tuple Tuple. All elements of Tuple either have to be strings or numbers (integer or floating point numbers). It is not allowed to mix strings with numerical

values. The result parameter Min will contain a floating point number, if at least one element of Tuple is a floating point number. If all elements of Tuple are integer numbers the resulting minimum in Min will also be an integer number.

Exception: Empty input tuple

If the input tuple is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_min, which can be used in an expression in the following syntax:

```
Min := min(Tuple)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple.

▷ **Min** (output_control) . . . . . tuple(-array) ⇝ *real* / integer / string
Minimal element of the input tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_max, tuple_mean, tuple_deviation, tuple_sum, tuple_length, tuple_median

See also

tuple_max2, tuple_min2

Module

Foundation

tuple_sum (: : Tuple : Sum)

Return the sum of all elements of a tuple.

tuple_sum returns the sum of all elements of the input tuple Tuple. All elements of Tuple either have to be strings or numbers (integer or floating point numbers). It is not allowed to mix strings with numerical values. The result parameter Sum will contain a floating point number, if at least one element of Tuple is a floating point number. If all elements of Tuple are integer numbers the resulting sum will also be an integer number. If Tuple contains strings, the concatenation will be used for building the sum.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_sum, which can be used in an expression in the following syntax:

```
Sum := sum(Tuple)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple.

▷ **Sum** (output_control) . . . . . tuple(-array) ⇝ *real* / integer / string
Sum of tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_mean, tuple_deviation, tuple_min, tuple_max, tuple_length, tuple_median

See also

tuple_cumul

Module

Foundation

28.9 Logical Operations

tuple_and (: : T1, T2 : And)

Compute the logical and of two tuples.

tuple_and computes the logical and of the input tuples T1 and T2. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either T1 or T2 must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Exception: Empty input tuples

If any of the input tuples is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_and, which can be used in an expression in the following syntax:

```
And := T1 and T2
```

Parameters

▷ **T1** (input_control) ..... integer(-array) ⇝ *integer*
Input tuple 1.

▷ **T2** (input_control) ..... integer(-array) ⇝ *integer*
Input tuple 2.

▷ **And** (output_control) ..... integer(-array) ⇝ *integer*
Logical and of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_or, tuple_xor, tuple_not

See also

tuple_band, tuple_bor, tuple_bxor, tuple_bnot

Module

Foundation

tuple_not (: : T : Not)

Compute the logical not of a tuple.

tuple_not computes the logical not of the input tuple T. The input tuple must contain only integer numbers.

Exception: Empty input tuple

If the input tuple is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_not`, which can be used in an expression in the following syntax:

```
Not := not T1
```

Parameters

▷ **T** (input_control) . . . integer(-array) ⇝ *integer*
Input tuple.

▷ **Not** (output_control) . . . integer(-array) ⇝ *integer*
Binary not of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_and`, `tuple_or`, `tuple_xor`

See also

`tuple_band`, `tuple_bor`, `tuple_bxor`, `tuple_bnot`

Module

Foundation

tuple_or (: : T1, T2 : Or)

Compute the logical or of two tuples.

`tuple_or` computes the logical or of the input tuples `T1` and `T2`. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either `T1` or `T2` must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Exception: Empty input tuples

If any of the input tuples is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_or`, which can be used in an expression in the following syntax:

```
Or := T1 or T2
```

Parameters

▷ **T1** (input_control) . . . integer(-array) ⇝ *integer*
Input tuple 1.

▷ **T2** (input_control) . . . integer(-array) ⇝ *integer*
Input tuple 2.

▷ **Or** (output_control) . . . integer(-array) ⇝ *integer*
Logical or of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Alternatives

tuple_and, tuple_xor, tuple_not

See also

tuple_band, tuple_bor, tuple_bxor, tuple_bnot

Module

Foundation

tuple_xor (: : T1, T2 : Xor)

Compute the logical exclusive or of two tuples.

tuple_xor computes the logical exclusive or of the input tuples T1 and T2. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either T1 or T2 must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Exception: Empty input tuples

If any of the input tuples is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_xor, which can be used in an expression in the following syntax:

```
Xor := T1 xor T2
```

Parameters

▷ **T1** (input_control) ..... integer(-array) ⇝ *integer*
Input tuple 1.

▷ **T2** (input_control) ..... integer(-array) ⇝ *integer*
Input tuple 2.

▷ **Xor** (output_control) ..... integer(-array) ⇝ *integer*
Binary exclusive or of the input tuples.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_and, tuple_or, tuple_not

See also

tuple_band, tuple_bor, tuple_bxor, tuple_bnot

Module

Foundation

28.10 Manipulation

tuple_insert (: : Tuple, Index, InsertTuple : Extended)

Inserts one or more elements into a tuple at index.

tuple_insert inserts the elements of the tuple InsertTuple at index into the tuple Tuple and returns them in the tuple Extended. In this context Index determines the start index of the elements and InsertTuple

the values to insert. Successive values of Tuple will be positioned after the inserted values of InsertTuple. The parameter Index must contain a single integer value (any floating point number must represent an integer value without fraction). Indices of tuple elements start at 0. Therefore, the first tuple element has got the index 0. If Index contains the length of Tuple as index, InsertTuple will be appended. The length of the result tuple Extended is always the sum of the two input tuples Tuple and InsertTuple. For example, if Tuple contains the values [0,1,0,1,0,1], the Index contains the value [3] and the InsertTuple contains the values [2,2,2], then the output tuple Extended will contain the values [0,1,0,2,2,2,1,0,1]. It is allowed to mix strings and numbers in the input tuples Tuple and InsertTuple.

Exception: Empty input tuples

If any of the input tuples is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_insert, which can be used in an expression in the following syntax:

```
Extended := insert(Tuple, Index, InsertTuple)
```

Parameters

- ▷ **Tuple** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple.
- ▷ **Index** (input_control) ..... integer ⇝ *integer*
  Start index of elements to be inserted.
- ▷ **InsertTuple** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Element(s) to insert at index.
- ▷ **Extended** (output_control) ..... tuple-array ⇝ *integer* / real / string
  Tuple with inserted elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_concat, tuple_replace, tuple_gen_const

See also

tuple_remove

Module

Foundation

tuple_remove (: : Tuple, Index : Reduced)

Remove elements from a tuple.

tuple_remove removes one or more elements from the tuple Tuple and returns the rest in the tuple Reduced. Index determines the indices of the elements to remove. Thus, Index may only contain integer values (any floating point number within Index must represent an integer value without fraction). Note that indices of tuple elements start at 0, i.e. the first tuple element has got the index 0. Duplicates and indices out of range are ignored.

Exception: Empty input tuple

If Tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_remove, which can be used in an expression in the following syntax:

```
Reduced := remove(Tuple, Index)
```

Parameters

- ▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Input tuple.
- ▷ **Index** (input_control) . . . . . integer(-array) ⇝ *integer*
  Indices of the elements to remove.
- ▷ **Reduced** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Reduced tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_first_n, tuple_last_n, tuple_str_bit_select, tuple_concat, tuple_insert, tuple_replace

See also

tuple_select, tuple_select_mask

Module

Foundation

tuple_replace (: : Tuple, Index, ReplaceTuple : Replaced)

Replaces one or more elements of a tuple.

tuple_replace replaces one or more elements of the input tuple Tuple and returns them with Replaced. At this, Index determines the indices of the elements and ReplaceTuple the corresponding values to replace. The parameter Index must contain one or more integer values (any floating point number must represent an integer value without fraction). Indices of tuple elements start at 0. Therefore, the first tuple element has got the index 0. If ReplaceTuple contains only one value, this value will be replaced at all indices of Index. If a value of Index is greater than the length of the input tuple Tuple, Replaced will be extended accordingly and initialized with zeros. For example, if Tuple contains [1], Index contains the values [2,4], and ReplaceTuple contains the values [3,5], Replaced will be [1,0,3,0,5].

It is allowed to mix strings and numbers in the input tuples Tuple and ReplaceTuple.

Exception: Empty input tuples

If either Index or ReplaceTuple is empty and the other is not, an exception is raised. If both are empty, the output tuple Replaced corresponds to the input Tuple. If both are empty, but the input Tuple is not, the empty tuple will be extended as described above.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_replace, which can be used in an expression in the following syntax:

```
Replaced := replace(Tuple, Index, ReplaceTuple)
```

Parameters

- ▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Input tuple.
- ▷ **Index** (input_control) . . . . . integer(-array) ⇝ *integer*
  Index/Indices of elements to be replaced.
- ▷ **ReplaceTuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
  Element(s) to replace.
- ▷ **Replaced** (output_control) . . . . . tuple-array ⇝ *integer* / real / string
  Tuple with replaced elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_select, tuple_first_n, tuple_last_n, tuple_select_mask, tuple_str_bit_select, tuple_concat, tuple_select_rank

See also

tuple_remove, tuple_insert

Module

Foundation

28.11 Selection

tuple_find (: : Tuple, ToFind : Indices)

Return the indices of all occurrences of a tuple within another tuple.

tuple_find searches Tuple sequentially for all occurrences of the values of the second tuple ToFind and returns the indices in Indices (in relation to the first input tuple Tuple). For example, if Tuple contains the values [3,4,5,6,1,2,3,4,0] and ToFind contains the values [3,4], the output tuple Indices will contain the values [0,6]. If the first tuple does not contain the second tuple as a subtuple, tuple_find returns -1 in Indices. It is allowed to mix strings and numbers in the input tuples.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_find, which can be used in an expression in the following syntax:

```
Indices := find(Tuple, ToFind)
```

Parameters

▷ **Tuple** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
Input tuple to examine.

▷ **ToFind** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
Input tuple with values to find.

▷ **Indices** (output_control) ..... integer(-array) ⇝ *integer*
Indices of the occurrences of the values to find in the tuple to examine.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_find_first, tuple_find_last, tuple_sort, tuple_inverse, tuple_equal_elem

Module

Foundation

tuple_find_first (: : Tuple, ToFind : Index)

Return the index of the first occurrence of a tuple within another tuple.

tuple_find_first searches forward through Tuple for the first occurrence of the values of the second tuple ToFind and returns the Index (in relation to the first input tuple Tuple). For example, if Tuple contains the values [3,4,5,6,1,2,3,4,0] and ToFind contains the values [3,4], the output Index is 0. If the first tuple does not contain the second tuple as a subtuple, tuple_find_first returns -1 in Index. It is allowed to mix strings and numbers in the input tuples.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_find_first, which can be used in an expression in the following syntax:

```
Index := find_first(Tuple, ToFind)
```

Parameters

- ▷ **Tuple** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple to examine.
- ▷ **ToFind** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple with values to find.
- ▷ **Index** (output_control) ..... integer ⇝ *integer*
  Index of the first occurrence of the values to find.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_find, tuple_find_last, tuple_sort, tuple_equal_elem

Module

Foundation

tuple_find_last (: : Tuple, ToFind : Index)

Return the index of the last occurrence of a tuple within another tuple.

tuple_find_last searches backward through Tuple for the first occurrence of the values of the second tuple ToFind and returns the Index (in relation to the first input tuple Tuple). For example, if Tuple contains the values [3,4,5,6,1,2,3,4,0] and ToFind contains the values [3,4], the output Index is 6. If the first tuple does not contain the second tuple as a subtuple, tuple_find_last returns -1 in Index. It is allowed to mix strings and numbers in the input tuples.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_find_last, which can be used in an expression in the following syntax:

```
Index := find_last(Tuple, ToFind)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple to examine.
▷ **ToFind** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple with values to find.
▷ **Index** (output_control) . . . . . integer ⇝ *integer*
Index of the last occurrence of the values to find.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_find, tuple_find_first, tuple_sort, tuple_equal_elem

Module

Foundation

tuple_first_n (: : Tuple, Index : Selected)

Select the first elements of a tuple up to the index "n".

tuple_first_n selects the first elements of Tuple up to the index Index and returns them in Selected. Indices of tuple elements start at 0, that means, the first tuple element has got the index 0. In total, Index+1 elements are returned.

Index must contain a single integer value (or a floating point number that represents an integer value without fraction).

Exception: Empty input tuple

If Tuple is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_first_n, which can be used in an expression in the following syntax:

```
Selected := firstn(Tuple, Index)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple.
▷ **Index** (input_control) . . . . . integer ⇝ *integer*
Index of the last element to select.
▷ **Selected** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Selected tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_last_n, tuple_select, tuple_str_bit_select, tuple_concat, tuple_select_mask

See also

tuple_remove

Module

Foundation

tuple_last_n (: : Tuple, Index : Selected)

Select all elements from index "n" to the end of a tuple.

Starting with the "n-th" element of the tuple Tuple, tuple_last_n selects every element of Tuple and returns it with Selected. Thus, Selected contains all elements of Tuple from index "n" up to the last element of Tuple (including the element at position "n"). The index "n" is determined by the input parameter Index. Thus, Index must contain a single integer value (if Index consists of a floating point number, this must represent an integer value without fraction). Indices of tuple elements start at 0, that means, the first tuple element has got the index 0.

Exception: Empty input tuple

If Tuple is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_last_n, which can be used in an expression in the following syntax:

```
Selected := lastn(Tuple, Index)
```

Parameters

- ▷ **Tuple** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple.
- ▷ **Index** (input_control) ..... integer ⇝ *integer*
  Index of the first element to select.
- ▷ **Selected** (output_control) ..... tuple(-array) ⇝ *integer* / real / string
  Selected tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_first_n, tuple_select, tuple_str_bit_select, tuple_concat, tuple_select_mask

See also

tuple_remove

Module

Foundation

tuple_select (: : Tuple, Index : Selected)

Select single elements of a tuple.

tuple_select selects one or more single elements of the tuple Tuple and returns them with Selected. At this, Index determines the indices of the elements to select. Thus, Index may only contain integer values (any floating point number within Index must represent an integer value without fraction). Indices of tuple elements start at 0, that means, the first tuple element has got the index 0.

Exception: Empty input tuples

If Tuple is empty, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_select, which can be used in an expression in the following syntax:

```
Selected := Tuple[Index]
```

```
Selected := subset(Tuple, Index)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple.

▷ **Index** (input_control) . . . . . integer(-array) ⇝ *integer*
Indices of the elements to select.

▷ **Selected** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Selected tuple element.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_first_n, tuple_last_n, tuple_str_bit_select, tuple_concat, tuple_select_range, tuple_select_rank

See also

tuple_remove

Module

Foundation

tuple_select_mask (: : Tuple, Mask : Selected)

Select in mask specified elements of a tuple.

tuple_select_mask selects one or more single elements of the tuple Tuple and returns them with Selected. For every element Mask determines the corresponding element to select. If the value is greater than 0, the appropriate element is selected. Thus the length of the two input tuples has to be equal and Mask may only contain integer or float values.

Exception: Empty input tuples

If both input tuples are empty, the operator returns an empty tuple. If only one of the input tuples is empty and the other is not, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_select_mask, which can be used in an expression in the following syntax:

```
Selected := select_mask(Tuple, Mask)
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple.

▷ **Mask** (input_control) . . . . . integer(-array) ⇝ *integer*
> 0 specifies the elements to select.

▷ **Selected** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Selected tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_first_n, tuple_last_n, tuple_str_bit_select, tuple_concat, tuple_select, tuple_select_range, tuple_select_rank

See also

tuple_remove

Module

Foundation

tuple_select_range (: : Tuple, Leftindex, Rightindex : Selected)

Select several elements of a tuple.

tuple_select_range selects several consecutive elements of the input tuple Tuple and returns them with Selected. At this, Leftindex determines the index of the first element and Rightindex determines the index of the last element to select. Thus, both parameters Leftindex and Rightindex must contain a single integer value (any floating point number must represent an integer value without fraction). Indices of tuple elements start at 0, that means, the first tuple element has got the index 0. The result tuple Selected contains every element from the tuple Tuple that has got an index between Leftindex and Rightindex (including the elements at position Leftindex and Rightindex). If the indices are equal, only one element is selected. In addition, it is possible that the right index is Leftindex - 1. In this case and if the left or right index are valid an empty tuple is returned. Hence, the index Rightindex must be greater or equal to Leftindex - 1.

Exception: Empty input tuples

If Leftindex and Rightindex are empty tuples, the operator returns an empty tuple. If only one of those parameters or Tuple is an empty tuple, an exception is raised.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_select_range, which can be used in an expression in the following syntax:

```
Selected := Tuple[Leftindex:Rightindex]
```

Parameters

- ▷ **Tuple** (input_control) ..... tuple(-array) ⇝ *integer* / real / string
  Input tuple.
- ▷ **Leftindex** (input_control) ..... integer(-array) ⇝ *integer*
  Index of first element to select.
- ▷ **Rightindex** (input_control) ..... integer(-array) ⇝ *integer*
  Index of last element to select.
- ▷ **Selected** (output_control) ..... tuple(-array) ⇝ *integer* / real / string
  Selected tuple elements.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_select, tuple_first_n, tuple_last_n, tuple_select_mask, tuple_str_bit_select, tuple_concat, tuple_select_rank

See also

tuple_remove

Module

Foundation

tuple_select_rank (: : Tuple, RankIndex : Selected)

Select the element of rank n of a tuple.

tuple_select_rank sorts the elements of the tuple Tuple and returns the element of rank n in Selected. RankIndex determines the index of the element to select. Thus, RankIndex may only contain integer values (any floating point number in RankIndex must represent an integer value without fraction). Indices of tuple elements start at 0, i.e. the lowest tuple element has the index 0.

Exception: Empty input tuples

If RankIndex is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_select_rank, which can be used in an expression in the following syntax:

```
Selected := select_rank(Tuple, RankIndex)
```

Parameters

▷ **Tuple** (input_control) ..... number(-array) ⇝ *integer* / real
Input tuple.

▷ **RankIndex** (input_control) ..... number ⇝ *integer* / real
Rank of the element to select.

▷ **Selected** (output_control) ..... number ⇝ *integer* / real
Selected tuple element.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_sort_index, tuple_sort

See also

tuple_median, tuple_select

Module

Foundation

tuple_str_bit_select (: : Tuple, Index : Selected)

Select single character or bit from a tuple.

tuple_str_bit_select selects a single character or bit from a tuple Tuple of integer numbers and/or strings. The input parameter Index determines the character or bit position to select. Index must contain a single number. If Index contains a floating point number, this may only represent an integer value (without fraction). The result tuple Selected contains a new element for each element of Tuple. Let Index contain the number "n" then each element of Selected consists of the "n-th" character (for strings) or "n-th" bit (for integers) of the corresponding element of Tuple.

If Tuple is empty, an exception is raised.

Unicode code points versus bytes

The index reference Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the index should reference the raw bytes of the string, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_str_bit_select`, which can be used in an expression in the following syntax:

```
Selected := Tuple{Index}
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *string* / integer
Input tuple.

▷ **Index** (input_control) . . . . . integer ⇝ *integer*
Position of character or bit to select.

▷ **Selected** (output_control) . . . . . tuple(-array) ⇝ *string* / integer
Tuple containing the selected characters and bits.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_select`, `tuple_first_n`, `tuple_last_n`, `tuple_concat`, `tuple_strchr`, `tuple_strrchr`, `tuple_str_first_n`, `tuple_str_last_n`, `tuple_substr`, `tuple_and`, `tuple_or`, `tuple_xor`, `tuple_not`

See also

`tuple_remove`

Module

Foundation

tuple_uniq (: : Tuple : Uniq)

Discard all but one of successive identical elements of a tuple.

`tuple_uniq` discards all but one of successive identical elements from the input tuple `Tuple` and returns the remaining elements in `Uniq`. For example, if `Tuple` contains the values [0,0,1,1,1,2,0,1], the output tuple `Uniq` will contain the values [0,1,2,0,1]. It is allowed to mix strings and numbers in the input tuple.

To get a tuple `Uniq` that contains all different entries of `Tuple` exactly once, use the operator `tuple_sort` first. Note however that in this case the output tuple `Uniq` is sorted. The result of the above example then is [0,1,2].

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_uniq`, which can be used in an expression in the following syntax:

```
Uniq := uniq(Tuple)
```

As mentioned above, you can use `tuple_sort` to truly get all unique elements of `Tuple`.

```
Uniq := uniq(sort(Tuple))
```

Parameters

▷ **Tuple** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Input tuple.

▷ **Uniq** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string
Tuple without successive identical elements.

Example

```
Tuple := [0,0,1,1,1,2,0,1]
*
tuple_uniq (Tuple, Uniq)
*
tuple_sort (Uniq, Sorted)
tuple_uniq (Sorted, Uniq1)
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

tuple_inverse, tuple_sort

Alternatives

tuple_intersection

Module

Foundation

28.12 Sets

tuple_difference (: : Set1, Set2 : Difference)

Compute the difference set of two input tuples.

tuple_difference returns the difference set from Set1 and Set2 in Difference. For example, if Set1 contains the values [0,1,3,3,5] and Set2 contains the values [2,3,5,10], the output Difference is [0,1]. The operator also allows mixed types of elements in the input tuples. However, the elements with different types will be considered as different elements, i.e. 1.0 and 1 are different. Also, this means, if Set1 and Set2 have different types of elements, their difference set could be Set1 itself. For example, if Set1 contains the values [2,5,3] and Set2 contains the values [2.0,5.0,3.0], the output Difference is [2,3,5]. Please note that the order of tuple elements resulting from tuple_difference does not necessarily conform to the order in the input tuple.

Exception: Empty input tuples

If both input tuples are empty, the operator returns an empty tuple. If Set1 is empty and Set2 is not, the operator returns an empty tuple as well. If Set2 is empty and Set1 is not, the output corresponds to Set1.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_difference, which can be used in an expression in the following syntax:

```
Difference := difference(Set1, Set2)
```

Parameters

▷ **Set1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **Set2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **Difference** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
The difference set of two input tuples.

Result

If the parameters are valid, the operator tuple_difference returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

tuple_symmdiff

See also

tuple_intersection, tuple_symmdiff, tuple_union

Module

Foundation

tuple_intersection (: : Set1, Set2 : Intersection)

Compute the intersection set of two input tuples.

tuple_intersection returns the intersection set from Set1 and Set2 in Intersection. For example, if Set1 contains the values [0,1,3,3,5] and Set2 contains the values [2,3,5,10], the output Intersection is [3,5]. The operator also allows mixed types of elements in the input tuples. However, the elements with different types will be considered as different elements, i.e. 1.0 and 1 are different. Also, this means, if Set1 and Set2 have different types of elements, their intersection set could be empty. For example, if Set1 contains the values [2,5,3] and Set2 contains the values [2.0,5.0,0,10], the output Intersection is empty. Please note that the order of tuple elements resulting from tuple_intersection does not necessarily conform to the order in the input tuple.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_intersection, which can be used in an expression in the following syntax:

```
Intersection := intersection(Set1, Set2)
```

Parameters

▷ **Set1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **Set2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **Intersection** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
The intersection set of two input tuples.

Result

If the parameters are valid, the operator tuple_intersection returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

tuple_union

See also

tuple_difference, tuple_symmdiff, tuple_union

Module

Foundation

tuple_symmdiff (: : Set1, Set2 : SymmDiff)

Compute the symmetric difference set of two input tuples.

tuple_symmdiff returns the symmetric difference set from Set1 and Set2 in SymmDiff. For example, if Set1 contains the values [0,1,3,3,5] and Set2 contains the values [2,3,5,10], the output SymmDiff is [0,1,2,10]. The operator also allows mixed types of elements in the input tuples. However, the elements with different types will be considered as different elements, i.e. 1.0 and 1 are different. Also, this means, if Set1 and Set2 have different types of elements, their symmetric difference set could be the union. For example, if Set1 contains the values [2,5,3] and Set2 contains the values [2.0,5.0], the output SymmDiff is [2,3,5,2.0,5.0]. Please note that the order of tuple elements resulting from tuple_symmdiff does not necessarily conform to the order in the input tuple.

Exception: Empty input tuples

If both of the input tuples are empty, the operator returns an empty tuple. If one of the input tuples is empty and the other is not, the output corresponds to the input tuple that is not empty.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_symmdiff, which can be used in an expression in the following syntax:

```
SymmDiff := symmdiff(Set1, Set2)
```

Parameters

- ▷ **Set1** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
  Input tuple.
- ▷ **Set2** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
  Input tuple.
- ▷ **SymmDiff** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
  The symmetric difference set of two input tuples.

Result

If the parameters are valid, the operator tuple_symmdiff returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

tuple_difference

See also

tuple_difference, tuple_intersection, tuple_union

Module

Foundation

tuple_union (: : Set1, Set2 : Union)

Compute the union set of two input tuples.

tuple_union returns the union set from Set1 and Set2 in Union. For example, if Set1 contains the values [0,1,3,3,5] and Set2 contains the values [3,4], the output Union is [0,1,3,4,5]. The operator also allows mixed types of elements in the input tuples. However, the elements with different types will be considered as different elements, i.e. 1.0 and 1 are different. For example, the union of [2,5,3] and [4.0,5.0] is [2,3,5,4.0,5.0]. Please note that the order of tuple elements resulting from tuple_union does not necessarily conform to the order in the input tuple.

Exception: Empty input tuples

If both of the input tuples are empty, the operator returns an empty tuple. If one of the input tuples is empty and the other is not, the output corresponds to the input tuple that is not empty.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_union`, which can be used in an expression in the following syntax:

```
Union := union(Set1, Set2)
```

Parameters

▷ **`Set1`** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **`Set2`** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **`Union`** (output_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
The union set of two input tuples.

Result

If the parameters are valid, the operator `tuple_union` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on internal data level.

Alternatives

`tuple_intersection`

See also

`tuple_difference`, `tuple_intersection`, `tuple_symmdiff`

Module

Foundation

28.13 String Operations

This chapter contains operators for string operations.

General Information

The HALCON library encodes strings in UTF-8 by default.

UTF-8 is a Unicode character encoding that can encode all Unicode code points with one to four bytes. 'Unicode' refers to the character set that assigns each character of a string to a code point (for example, 'U + 0041' for 'A'). UTF-8 then translates the Unicode code points into binary data. The memory requirement for all ASCII characters is usually 1 byte. Certain characters, such as German umlauts, or Greek and Cyrillic characters require 2 bytes. Asian characters occupy up to 4 bytes per character.

By default, the HALCON string operators work on the basis of Unicode code points. That is, accessing a character of a string always returns the corresponding Unicode code point of the character, regardless of how many bytes are needed to represent the code point in UTF-8. Thus, multi-byte characters, such as Asian characters or German umlauts can be uniformly translated on all systems. Please note that the Unicode standard also allows you to assemble printable characters from multiple code points (using so-called 'Combining Diacritical Marks'). This is currently not fully supported by HALCON: in HALCON, the code points are processed separately, and when strings are compared, equivalent characters are not set equal if coded with different code points.

If there are compatibility problems during execution of older programs, the string encoding of the HALCON library can be changed from *'utf8'* to *'locale'* (legacy mode). Then strings are stored depending on the locale and the string operators work – as in previous versions of HALCON – not characterwise but bytewise. If bytewise character processing is also required in UTF-8 mode, the operator `set_system` can be used to set the option *'tuple_string_operator_mode'* from *'codepoint'* to *'byte'*. Afterwards, the string operators no longer work on the basis of code points. The byte sequence of a string can be interesting for debugging, for example.

tuple_environment (: : Names : Values)

Read one or more environment variables.

tuple_environment reads the content of all environment variables that are referenced by their names in the input tuple Names and returns the content with the output tuple Values. The input tuple may only contain strings.

For general information about string operations see Tuple / String Operations.

Exception: Empty input tuples

If the input tuple is empty, the operator returns an empty tuple. Additionally, an empty string is returned for every name within Names that does not denote a valid environment variable.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_environment, which can be used in an expression in the following syntax:

```
Values := environment(Names)
```

Parameters

▷ **Names** (input_control) ..... string(-array) ⇝ *string*
Tuple containing name(s) of the environment variable(s).

▷ **Values** (output_control) ..... string(-array) ⇝ *string*
Content of the environment variable(s).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_strstr, tuple_strrstr, tuple_strchr, tuple_strrchr, tuple_strlen, tuple_str_first_n, tuple_str_last_n, tuple_split, tuple_substr

Module

Foundation

tuple_join (: : Strings, Separators : JoinedStrings)

Join strings using separator symbol(s).

tuple_join joins the strings of the input tuple Strings interleaved with the separators defined in the input tuple Separators and returns the result in the tuple JoinedStrings.

For each string element in Separators, there is one corresponding element in JoinedStrings which holds the elements of Strings joined by this separator. Thereby an empty string in Separators corresponds to simple string concatenation.

Example: Strings = *['aaa', 'bbb', 'ccc', 'ddd']* and Separators = *['++', '–', '']*. In this case the operator returns the tuple JoinedStrings = *['aaa++bbb++ccc++ddd', 'aaa–bbb–ccc–ddd', 'aaabbbcccddd']*.

Both input tuples (Strings and Separators) may only consist of strings. Otherwise tuple_join returns an error.

Exception: Empty input tuples

If either or both of the input tuples are empty, the operator returns an empty tuple.

Unicode code points versus bytes

A separator always consists of the entire string. This is true regardless of the system setting of *'tsp_tuple_string_operator_mode'*.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_join`, which can be used in an expression in the following syntax:

```
JoinedStrings := join(Strings, Separators)
```

Parameters

▷ **`Strings`** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with strings to join.

▷ **`Separators`** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with separator symbol(s).

▷ **`JoinedStrings`** (output_control) . . . . . string(-array) ⇝ *string*
Output tuple with the contained strings.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`tuple_split`

Alternatives

`tuple_strstr`, `tuple_strrstr`, `tuple_strchr`, `tuple_strrchr`, `tuple_strlen`, `tuple_str_first_n`, `tuple_str_last_n`, `tuple_environment`

Module

Foundation

`tuple_regexp_match` (: : Data, Expression : Matches)

Extract substrings using regular expressions.

`tuple_regexp_match` applies the regular expression in `Expression` to one or more input strings in `Data`, and in each case returns the first matching substring in `Matches`. Normally, one output string is returned for each input string, the output string being empty if no match was found. However, if the regular expression contains capturing groups (see below), the behavior depends on the number of input strings: If there is only a single input string, the result is a tuple of all captured submatches. If there are multiple input strings, the output strings represent the matched pattern of the first capturing group.

A summary of regular expression syntax is provided here. Basically, each character in the regular expression represents a literal to match, except for the following symbols which have a special meaning (the described syntax is compatible with Perl):

^	Matches start of string
$	Matches end of string (a trailing newline is allowed)
.	Matches any character except newline
[...]	Matches any character literal listed in the brackets. If the first character is a '^', this matches any character except those in the list. You can use the '-' character as in [A-Z0-9] to select character ranges. Other characters lose their special meaning in brackets, except '\'. Within these brackets it is possible to use the following POSIX character classes (note that the additional brackets are needed): [:alnum:] alphabetic and numeric characters [:alpha:] alphabetic characters [:blank:] space and tab [:cntrl:] control characters [:digit:] digits [:graph:] non-blank (like spaces or control characters) [:lower:] lowercase alphabetic characters [:print:] like [:graph:] but including spaces [:punct:] punctuation characters [:space:] all whitespace characters ([:blank:], newline, ...) [:upper:] uppercase alphabetic characters [:xdigit:] digits allowed in hexadecimal numbers (0-9a-fA-F).
*	Allows 0 or more repetitions of preceding literal or group
+	Allows 1 or more repetitions of preceding literal or group
?	Allows 0 or 1 repetitions of preceding literal or group
{n,m}	Allows n to m repetitions of preceding literal or group
{n}	Allows exactly n repetitions of preceding literal or group
|	Separates alternative matching expressions
()	Groups a subpattern and creates a capturing group. The substrings captured by this group will be stored separately. (?:) Groups a subpattern without creating a capturing group (?=) Positive lookahead (requested condition right to the match) (?!) Negative lookahead (forbidden condition right to the match) (?<=) Positive lookbehind (requested condition left to the match) (?<!) Negative lookbehind (forbidden condition left to the match)
\	Escapes any special symbol to treat it as a literal. *Attention:* Some host languages like HDevelop and C/C++ already use the backslash as a general escape character. In this case, '\.' matches a literal dot while '\\' matches a literal backslash. Furthermore, there are some special codes: \d Matches a digit (Negation: \D) \w Matches a letter, digit or underscore (Negation: \W) \s Matches a white space character (Negation: \S) \b Matches a word boundary (Negation: \B)

The repeat quantifiers listed in the table above are greedy by default, i.e., they attempt to maximize the length of the match. Appending '?' attempts to find a minimal match, e.g., '+?'.

If the specified expression is syntactically incorrect, you will receive an error stating that the value of control parameter 2 is wrong. Additional details are displayed in a message box if set_system('do_low_error', 'true') is set and in HDevelop's Output Console.

Furthermore, you can set some options by passing a string tuple for Expression. In this case, the first element is used as the expression, and each additional element is treated as an option.

- *'ignore_case'*: Perform case-insensitive matching
- *'multiline'*: '^' and '$' match start and end of individual lines
- *'dot_matches_all'*: Allow the '.' character to also match newlines
- *'newline_lf'*, *'newline_crlf'*, *'newline_cr'*: Specify the encoding of newlines in the input data. The default is LF on all systems (even though in Windows files usually CRLF is used as line break, when reading a file into memory the read operators return for every line break just '\n', which is the same as LF).

For general information about string operations see Tuple / String Operations.

If the input parameter `Data` is an empty tuple, the operator returns an empty tuple. If `Expression` is an empty tuple, an exception is raised.

Unicode code points versus bytes

Regular expression matching operates on Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If regular expression matching should only match on bytes, this operator can be switched to byte mode with `set_system('tsp_tuple_string_operator_mode','byte')`. If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_regexp_match`, which can be used in an expression in the following syntax:

```
Matches := regexp_match(Data, Expression)
```

Parameters

- ▷ **`Data`** (input_control) ..... string(-array) ⇝ *string*
  Input strings to match.
- ▷ **`Expression`** (input_control) ..... string(-array) ⇝ *string*
  Regular expression.
  Default: '.*'
  Suggested values: `Expression` ∈ {'.*', 'ignore_case', 'multiline', 'dot_matches_all', 'newline_lf', 'newline_crlf', 'newline_cr'}
- ▷ **`Matches`** (output_control) ..... string(-array) ⇝ *string*
  Found matches.

Example

```
tuple_regexp_match ('abba', 'a*b*', Result)
* Returns 'abb'

tuple_regexp_match ('abba', 'b*a*', Result)
* Returns 'a'

tuple_regexp_match ('abba', 'b+a*', Result)
* Returns 'bba'

tuple_regexp_match ('abba', '.a', Result)
* Returns 'ba'

tuple_regexp_match ('abba', '[ab]*', Result)
* Returns 'abba'

tuple_regexp_match (['img123','img124'], 'img(.*)', Result)
* Returns ['123','124']

tuple_regexp_match ('mydir/img001.bmp', 'img(.*)\\.(.*)', Result)
* Returns ['001','bmp']
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_strstr`

See also

tuple_regexp_replace, tuple_regexp_test, tuple_regexp_select

References

Perl Compatible Regular Expressions (PCRE), http://www.pcre.org/

Module

Foundation

tuple_regexp_replace (: : Data, Expression, Replace : Result)

Replace a substring using regular expressions.

tuple_regexp_replace applies the regular expression in Expression to one or more input strings in Data, and replaces the **first** matching substring using the Replace expression. For each input string, a processed output string is returned in Result.

Please refer to the documentation of tuple_regexp_match for syntax and options of regular expressions. Additionally, tuple_regexp_replace supports the option *'replace_all'*, which causes **all** matches within each individual string to be replaced.

The Replace expression may use the tag *'$0'* to refer to the matched substring in the input data, *'$i'* to refer to the submatch of the i-th capturing group (for $i <= 9$), and *'$$'* to refer to the *'$'* literal.

For general information about string operations see Tuple / String Operations.

If Data is an empty tuple, the operator returns an empty tuple. If Replace is an empty tuple and Data is not empty, an exception is raised.

Unicode code points versus bytes

Regular expression matching operates on Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If regular expression matching should only match on bytes, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_regexp_replace, which can be used in an expression in the following syntax:

```
Result := regexp_replace(Data, Expression, Replace)
```

Parameters

- ▷ **Data** (input_control) ..... string(-array) ⇝ *string*
  Input strings to process.
- ▷ **Expression** (input_control) ..... string(-array) ⇝ *string*
  Regular expression.
  Default: '.*'
  Suggested values: Expression ∈ {'.*', 'replace_all', 'ignore_case', 'multiline', 'dot_matches_all', 'newline_lf', 'newline_crlf', 'newline_cr'}
- ▷ **Replace** (input_control) ..... string ⇝ *string*
  Replacement expression.
- ▷ **Result** (output_control) ..... string(-array) ⇝ *string*
  Processed strings.

Example

```
tuple_regexp_replace(['img10.bmp','img11.bmp','img12.bmp'], \
                     'img(.*).bmp', 'out$1.txt', Result)
* Returns ['out10.txt','out11.txt','out12.txt']
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

tuple_regexp_match, tuple_regexp_test, tuple_regexp_select

Module

Foundation

tuple_regexp_select (: : Data, Expression : Selection)

Select tuple elements matching a regular expression.

tuple_regexp_select applies the regular expression in Expression to one or more input strings in Data, and returns the matching string elements in Selection. This is convenient, e.g., for filtering a list of files obtained using the operator list_files.

Please refer to the documentation of tuple_regexp_match for syntax and options of regular expressions. Additionally, tuple_regexp_select supports the option *'invert_match'*, which causes those input strings to be selected which do **not** match the regular expression.

For general information about string operations see Tuple / String Operations.

If the input tuple is empty, the operator returns an empty tuple.

Unicode code points versus bytes

Regular expression matching operates on Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If regular expression matching should only match on bytes, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_regexp_select, which can be used in an expression in the following syntax:

```
Selection := regexp_select(Data, Expression)
```

Parameters

▷ **Data** (input_control) ..... string(-array) ⇝ *string*
Input strings to match.

▷ **Expression** (input_control) ..... string(-array) ⇝ *string*
Regular expression.
Default: '.*'
Suggested values: Expression ∈ {'.*', 'invert_match', 'ignore_case', 'multiline', 'dot_matches_all', 'newline_lf', 'newline_crlf', 'newline_cr'}

▷ **Selection** (output_control) ..... string(-array) ⇝ *string*
Matching strings

Example

```
tuple_regexp_select (['.','..','mydir','a.png','b.txt','c.bmp','d.dat'], \
                     '.(bmp|png)', Result)
* Returns ['a.png','c.bmp']

tuple_regexp_select (Files, ['training','invert_match'], Matches)
* Returns all file names that do *not* contain the string 'training'
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Alternatives

tuple_select

See also

tuple_regexp_match, tuple_regexp_replace, tuple_regexp_test

Module

Foundation

tuple_regexp_test (: : Data, Expression : NumMatches)

Test if a string matches a regular expression.

tuple_regexp_test applies the regular expression in Expression to one or more input strings in Data, and returns the number of matching strings in NumMatches. In particular, the result for a single input string will be 1 in case of a match, and 0 otherwise.

Please refer to the documentation of tuple_regexp_match for syntax and options of regular expressions. Additionally, tuple_regexp_test supports the option *'invert_match'*, which causes those input strings to be counted which do **not** match the regular expression.

For convenient use in conditional expressions, this functionality is also available as the =~ operation in HDevelop.

For general information about string operations see Tuple / String Operations.

If the input tuple is empty, the operator returns 0.

Unicode code points versus bytes

Regular expression matching operates on Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If regular expression matching should only match on bytes, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_regexp_test, which can be used in an expression in the following syntax:

NumMatches := regexp_test(Data, Expression) or NumMatches := Data =~ Expression

Parameters

▷ **Data** (input_control) ..... string(-array) ⇝ *string*
Input strings to match.

▷ **Expression** (input_control) ..... string(-array) ⇝ *string*
Regular expression.
Default: '.*'
Suggested values: Expression ∈ {'.*', 'invert_match', 'ignore_case', 'multiline', 'dot_matches_all', 'newline_lf', 'newline_crlf', 'newline_cr'}

▷ **NumMatches** (output_control) ..... integer ⇝ *integer*
Number of matching strings

Example

```
tuple_regexp_test ('p10662599755', '[A-Z]*', Result)
* Returns 0

tuple_regexp_test ('p10662599755', ['[A-Z]*','ignore_case'], Result)
* Returns 1
```

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_strstr

See also

tuple_regexp_match, tuple_regexp_replace, tuple_regexp_select

Module

Foundation

tuple_split (: : String, Separator : Substrings)

Split strings into substrings using predefined separator symbol(s).

tuple_split searches within the strings of the input tuple String for the separators defined in the input tuple Separator. tuple_split then splits the examined strings into the substrings divided by the separators and returns them in the tuple Substrings. The behavior depends on the length of the input tuples:

- Separator contains only one string: Every string in String is split up according to the separator symbols in Separator. E.g., String = *['alpha:1', 'beta:2', 'gamma:3']* and Separator = *['a:']*. In this case, the operator returns the tuple Substrings = *['lph', '1', 'bet', '2', 'g', 'mm', '3']*.
- String contains only one string: The single string is split up for every element in Separator. E.g., String = *['alpha:1 beta:2 gamma:3']* and Separator = *[':', '123']*. In this case, the output tuple Substrings will comprise the substrings *'alpha'*, *'1'*, *'beta'*, *'2'*, *'gamma'*, and *'3'*, as the result of splitting the string of String according to the first element of Separator (*':'*) as well as *'alpha:'*, *'beta:'*, and *'gamma:'* as the result of splitting according to the second element of Separator (*'123'*).
- Both tuples consist of the same amount of strings: The search is done elementwise. I.e., tuple_split will split the first string of String according to the separator symbols in the first element of Separator, the second string of String according to the separator symbols in the second element of Separator and so on.
- Either or both of the input tuples are empty: An empty tuple is returned.

Notes to Separator: If an element of Separator contains more than one character, each character defines a separator (see the example given above). Subsequent occurrences of Separator in String are treated as one separator. Separators at the beginning and the end of String will not result in an empty string.

Please consider, both input tuples (String and Separator) may only consist of strings. Otherwise tuple_split returns an error. If both input tuples contain more than one element and the number of elements differs for the input tuples, tuple_split returns an error.

Unicode code points versus bytes

The split characters are interpreted as Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the split characters should be handled as raw bytes of the string, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_split, which can be used in an expression in the following syntax:

```
Substrings := split(String, Separator)
```

Parameters

▷ **String** (input_control) ..... string(-array) ⇝ *string*
Input tuple with string(s) to split.

▷ **Separator** (input_control) ..... string(-array) ⇝ *string*
Input tuple with separator symbol(s).

▷ **Substrings** (output_control) ..... string(-array) ⇝ *string*
Substrings after splitting the input strings.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_strstr, tuple_strrstr, tuple_strchr, tuple_strrchr, tuple_strlen, tuple_str_first_n, tuple_str_last_n, tuple_environment

Module

Foundation

tuple_str_distance (: : String1, String2, Mode : Distance)

Calculate the distance between strings.

tuple_str_distance computes the edit distance between String1 and String2 and returns the result in Distance. The distance measure used is specified in Mode. Both input tuples may only consist of strings. Otherwise, tuple_str_distance returns an error. As an exception, an empty tuple in String1 or String2 cause Distance to also be empty.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each of the input tuples String1 and String2 can be given as a tuple of length *1* or *'N'*. Parameters with tuple length *1* will be repeated internally such that the number of computed distances is always *'N'*.

Supported distance measures

- *'levenshtein'*: Uses the Levenshtein distance to quantify string distances. The Levenshtein distance is the minimum number of edit operations required to transform String1 into String2. Valid edit operations are insertion, deletion and replacement of characters.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_str_distance, which can be used in an expression in the following syntax:

```
Result := str_distance(String1, String2, Mode)
```

Parameters

▷ **String1** (input_control) ..... string(-array) ⇝ *string*
Input tuple.
Default: 'String1'

▷ **String2** (input_control) ..... string(-array) ⇝ *string*
Input tuple.
Default: 'String2'

▷ **Mode** (input_control) ..... string ⇝ *string*
Distance measure.
Default: 'levenshtein'
List of values: Mode ∈ {'levenshtein'}

▷ **Distance** (output_control) ..... integer(-array) ⇝ *integer*
Element-wise string distance.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`get_text_result`, `find_text`, `do_ocr_word_mlp`

Alternatives

`suggest_lexicon`

References

Vladimir I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Doklady Akademii Nauk SSSR, 163(4):845-848, 1965 (Russian). English translation in Soviet Physics Doklady, 10(8):707-710, 1966.

Module

Foundation

tuple_str_first_n (: : String, Position : Substring)

Cut the first characters up to position "n" out of a string tuple.

`tuple_str_first_n` cuts the first characters up to position "n" out of each string of the input tuple `String` and returns them as new strings in the output tuple `Substring` (remark: the position within strings starts with 0 for the first character of a string). The number "n" is determined by the second input tuple `Position`. If `Position` only contains one element, this element contains the number "n". If `String` and `Position` have got the same number of elements, the first element of `Position` determines the position "n" for the first string of `String`, the second element of `Position` does so for the second string of `String` and so on. If `Position` contains more than one element and `String` contains only one string, `tuple_str_first_n` cuts more than one substrings out of this string. The elements of `Position` then determine the lengths of these substrings. If both input tuples contain more than one element but differ in the number of elements, `tuple_str_first_n` returns an error.

If both input tuples are empty, the operator returns an empty tuple. Similarly, if `String` is empty and `Position` is not, the operator returns an empty tuple. In contrast, if `Position` is empty and `String` is not, an exception is raised.

Unicode code points versus bytes

The position references Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the position should reference the raw bytes of the string, this operator can be switched to byte mode with `set_system('tsp_tuple_string_operator_mode','byte')`. If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_str_first_n`, which can be used in an expression in the following syntax:

```
Substring := str_firstn(String, Position)
```

Parameters

▷ **String** (input_control) ..... string(-array) ⇝ *string*
Input tuple with string(s) to examine.

▷ **Position** (input_control) ..... number(-array) ⇝ *integer* / real
Input tuple with position(s) "n".

▷ **Substring** (output_control) ..... string(-array) ⇝ *string*
The first characters of the string(s) up to position "n".

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_str_last_n, tuple_substr, tuple_strstr, tuple_strrstr, tuple_strlen, tuple_strchr, tuple_strrchr, tuple_split, tuple_environment

Module

Foundation

tuple_str_last_n (: : String, Position : Substring)

Cut all characters starting at position "n" out of a string tuple.

tuple_str_last_n cuts all characters from position "n" to the end of the string out of each string of the input tuple String and returns them as new strings in the output tuple Substring. The position "n" is determined by the second input tuple Position. If Position only contains one element, this element contains "n". If String and Position have got the same number of elements, the first element of Position determines the start position for the first string of String, the second element of Position does so for the second string of String and so on. If Position contains more than one element and String contains only one string, tuple_str_last_n cuts more than one substrings out of this string. The elements of Position then determine the start positions for these substrings. If both input tuples contain more than one element but differ in the number of elements, tuple_str_last_n returns an error.

If both input tuples are empty, the operator returns an empty tuple. Similarly, if String is empty and Position is not, the operator returns an empty tuple. In contrast, if Position is empty and String is not, an exception is raised.

Unicode code points versus bytes

The position references Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the position should reference the raw bytes of the string, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_str_last_n, which can be used in an expression in the following syntax:

```
Substring := str_lastn(String, Position)
```

Parameters

- ▷ **String** (input_control) ..... string(-array) ⇝ *string*
  Input tuple with string(s) to examine.
- ▷ **Position** (input_control) ..... number(-array) ⇝ *integer* / real
  Input tuple with position(s) "n".
- ▷ **Substring** (output_control) ..... string(-array) ⇝ *string*
  The last characters of the string(s) starting at position "n".

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_str_first_n, tuple_substr, tuple_strstr, tuple_strrstr, tuple_strlen, tuple_strchr, tuple_strrchr, tuple_split, tuple_environment

Module

Foundation

tuple_str_replace (: : String, Before, After : Replaced)

Replace all occurrences of a substring within a string.

tuple_str_replace replaces all occurrences of Before in String with After and returns the result in Replaced. All three tuples may only consist of strings. Otherwise, tuple_str_replace returns an error. As an exception, an empty tuple in After corresponds to an empty string. Before and After must contain the same number of elements. They can either contain one element, then Before is replaced with After in all input strings of the tuple String or they can contain exactly the same number of elements as the tuple String, then the n-th entry in Before is replaced with the n-th entry in After in the n-th entry of String. The string with all replacements is returned in Replaced and contains the same number of elements as String in both cases.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_str_replace, which can be used in an expression in the following syntax:

```
Replaced := str_replace(String, Before, After)
```

Parameters

- ▷ **String** (input_control) ..... string(-array) ⇝ *string*
  Input tuple with string(s) to work on.
- ▷ **Before** (input_control) ..... string(-array) ⇝ *string*
  Input tuple with string(s) to search.
- ▷ **After** (input_control) ..... string(-array) ⇝ *string*
  Input tuple with string(s) to replace Before.
- ▷ **Replaced** (output_control) ..... string(-array) ⇝ *string*
  Output tuple with replaced strings.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_regexp_replace

Module

Foundation

tuple_strchr (: : String, ToFind : Position)

Forward search for characters within a string tuple.

tuple_strchr searches within the strings of the input tuple String for the characters of the input tuple ToFind. Both input tuples may only consist of strings. Otherwise tuple_strchr returns an error. If the elements of ToFind contain more than one character, only the first character of each element is considered for searching. If String contains only one string, all the characters defined in ToFind are searched in this string. Thus, the output tuple consists of as many elements as ToFind. Whenever a searched character has been found, the position of its first occurrence gets stored in the output tuple Position (remark: the position starts at 0 for

the first character of a string). If a character can not be found, -1 will be returned instead of its position. If both input tuples show the same number of elements, the search is done elementwise. I.e., the first character of the first element of ToFind is searched within the first string of String, the first character of the second element of ToFind is searched within the second string of String and so on. The results of the elementwise searches are returned with Position that contains as many elements as String and ToFind. If ToFind only contains one string, its first character is searched within all strings of String. Thus, in this case Position consists of as many elements as String. If both input tuples contain more than one element and the number of elements differs for the input tuples, tuple_strchr returns an error.

If either or both of the input tuples are empty, the operator returns an empty tuple.

Unicode code points versus bytes

The position references Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the position should reference the raw bytes of the string, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_strchr, which can be used in an expression in the following syntax:

```
Position := strchr(String, ToFind)
```

Parameters

- ▷ **String** (input_control) . . . . . string(-array) ⇝ *string*
  Input tuple with string(s) to examine.
- ▷ **ToFind** (input_control) . . . . . string(-array) ⇝ *string*
  Input tuple with character(s) to search.
- ▷ **Position** (output_control) . . . . . integer(-array) ⇝ *integer*
  Position of searched character(s) within the string(s).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_strrchr, tuple_strstr, tuple_strrstr, tuple_strlen, tuple_substr, tuple_str_first_n, tuple_str_last_n, tuple_split, tuple_environment

Module

Foundation

tuple_strlen (: : T1 : Length)

Determine the length of every string within a tuple of strings.

tuple_strlen checks the length of every string within the input tuple T1 and returns the length of each string with the output tuple Length. All elements of T1 may only consist of strings. Otherwise tuple_strlen returns an error.

If the input tuple is empty, the operator returns an empty tuple.

Unicode code points versus bytes

The string length references Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the length should reference the raw bytes of the string, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_strlen`, which can be used in an expression in the following syntax:

```
Length := strlen(T1)
```

Parameters

▷ **T1** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple.

▷ **Length** (output_control) . . . . . integer(-array) ⇝ *integer*
Lengths of the single strings of the input tuple.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_strstr`, `tuple_strrstr`, `tuple_strchr`, `tuple_strrchr`, `tuple_substr`, `tuple_str_first_n`, `tuple_str_last_n`, `tuple_split`, `tuple_environment`

See also

`tuple_is_string`

Module

Foundation

tuple_strrchr (: : String, ToFind : Position)

Backward search for characters within a string tuple.

`tuple_strrchr` searches within the strings of the input tuple `String` for the characters of the input tuple `ToFind`. Both input tuples may only consist of strings. Otherwise `tuple_strrchr` returns an error. In any case backward search is used, i.e., every string is examined from its last to its first character. If the elements of `ToFind` contain more than one character, only the first character of each element is considered for searching. If `String` contains only one string, all the characters defined in `ToFind` are searched in this string. Thus, the output tuple consists of as many elements as `ToFind`. Whenever a searched character has been found, the position of its first occurrence gets stored in the output tuple `Position` . If a character can not be found, -1 will be returned instead of its position (remark: the position starts at 0 for the first character of a string). If both input tuples show the same number of elements, the search is done elementwise. I.e., the first character of the first element of `ToFind` is searched within the first string of `String`, the first character of the second element of `ToFind` is searched within the second string of `String` and so on. The results of the elementwise searches are returned with `Position` that contains as many elements as `String` and `ToFind`. If `ToFind` only contains one string, its first character is searched within all strings of `String`. Thus, in this case `Position` consists of as many elements as `String`. If both input tuples contain more than one element and the number of elements differs for the input tuples, `tuple_strrchr` returns an error.

If either or both of the input tuples are empty, the operator returns an empty tuple.

Unicode code points versus bytes

The position references Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the position should reference the raw bytes of the string, this operator can be switched to byte mode with `set_system('tsp_tuple_string_operator_mode','byte')`. If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_strrchr`, which can be used in an expression in the following syntax:

```
Position := strrchr(String, ToFind)
```

Parameters

▷ **String** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with string(s) to examine.

▷ **ToFind** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with character(s) to search.

▷ **Position** (output_control) . . . . . integer(-array) ⇝ *integer*
Position of searched character(s) within the string(s).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_strchr, tuple_strstr, tuple_strrstr, tuple_strlen, tuple_substr, tuple_str_first_n, tuple_str_last_n, tuple_split, tuple_environment

Module

Foundation

tuple_strrstr (: : String, ToFind : Position)

Backward search for strings within a string tuple.

tuple_strrstr searches within the strings of the input tuple String for the strings of the input tuple ToFind. Both input tuples may only consist of strings. Otherwise tuple_strrstr returns an error. In any case backward search is used, i.e., every string is examined from its last to its first character. If String contains only one string, all strings of ToFind are searched in it. Thus, the output tuple consists of as many elements as ToFind. Whenever a searched string has been found, the position of its first occurrence gets stored in the output tuple Position (positions in strings are counted starting with 0). If a string can not be found, -1 will be returned instead of its position. If both input tuples show the same number of elements, the strings are searched elementwise. I.e., the first string of ToFind is searched within the first string of String, the second string of ToFind is searched within the second string of String and so on. The results of the elementwise searches are returned with Position that contains as many elements as String and ToFind. If ToFind only contains one string, this is searched within all strings of String. Thus, in this case Position consists of as many elements as String. If both input tuples contain more than one element and the number of elements differs for the input tuples, tuple_strrstr returns an error.

If either or both of the input tuples are empty, the operator returns an empty tuple.

Unicode code points versus bytes

The position references Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the position should reference the raw bytes of the string, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_strrstr, which can be used in an expression in the following syntax:

```
Position := strrstr(String, ToFind)
```

Parameters

▷ **String** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with string(s) to examine.

▷ **ToFind** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with string(s) to search.

▷ **Position** (output_control) . . . . . integer(-array) ⇝ *integer*
Position of searched string(s) within the examined string(s).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_strstr, tuple_strlen, tuple_strchr, tuple_strrchr, tuple_substr, tuple_str_first_n, tuple_str_last_n, tuple_split, tuple_environment

Module

Foundation

tuple_strstr (: : String, ToFind : Position)

Forward search for strings within a string tuple.

tuple_strstr searches within the strings of the input tuple String for the strings of the input tuple ToFind. Both input tuples may only consist of strings. Otherwise tuple_strstr returns an error. If String contains only one string, all strings of ToFind are searched in it. Thus, the output tuple consists of as many elements as ToFind. Whenever a searched string has been found, the position of its first occurrence gets stored in the output tuple Position (positions in strings are counted starting with 0). If a string can not be found, -1 will be returned instead of its position. If both input tuples show the same number of elements, the strings are searched elementwise. I.e., the first string of ToFind is searched within the first string of String, the second string of ToFind is searched within the second string of String and so on. The results of the elementwise searches are returned with Position that contains as many elements as String and ToFind. If ToFind only contains one string, this is searched within all strings of String. Thus, in this case Position consists of as many elements as String. If both input tuples contain more than one element and the number of elements differs for the input tuples, tuple_strstr returns an error.

If either or both of the input tuples are empty, the operator returns an empty tuple.

Unicode code points versus bytes

The position references Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the position should reference the raw bytes of the string, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_strstr, which can be used in an expression in the following syntax:

```
Position := strstr(String, ToFind)
```

Parameters

▷ **String** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with string(s) to examine.

▷ **ToFind** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with string(s) to search.

▷ **Position** (output_control) . . . . . integer(-array) ⇝ *integer*
Position of searched string(s) within the examined string(s).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_strrstr, tuple_strlen, tuple_strchr, tuple_strrchr, tuple_substr, tuple_str_first_n, tuple_str_last_n, tuple_split, tuple_environment

Module

Foundation

tuple_substr (: : String, Position1, Position2 : Substring)

Cut characters from position "n1" through "n2" out of a string tuple.

tuple_substr cuts all characters from position "n1" through "n2" out of each string of the input tuple String and returns them as new strings in the output tuple Substring. The positions "n1" and "n2" are determined by the second and third input tuples Position1 and Position2. Their length has to be equal. If Position1 and Position2 only contain one element, this element defines for all strings of String "n1" and "n2", respectively . If String, Position1 and Position2 have got the same number of elements, the first elements of Position1 and Position2 determine the start and end position for the first string of String. The second elements of Position1 and Position2 do so for the second string of String and so on. If Position1 and Position2 contain more than one element and String contains only one string, tuple_substr cuts more than one substring out of this string. The elements of Position1 and Position2 then determine the start and end positions for these substrings. If all input tuples contain more than one element but differ in the number of elements, tuple_substr returns an error.

If String is an empty tuple, the operator returns an empty tuple. If String is not empty and Position1 and/or Position2 are empty tuples, an exception is raised.

Unicode code points versus bytes

The positions reference Unicode code points. One Unicode code point may be composed of multiple bytes in the UTF-8 string. If the positions should reference the raw bytes of the string, this operator can be switched to byte mode with set_system('tsp_tuple_string_operator_mode','byte'). If *'filename_encoding'* is set to *'locale'* (legacy), this operator always uses the byte mode.

For general information about string operations see Tuple / String Operations.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_substr, which can be used in an expression in the following syntax:

```
Substring := String{Position1:Position2}
```

Parameters

▷ **String** (input_control) . . . . . string(-array) ⇝ *string*
Input tuple with string(s) to examine.

▷ **Position1** (input_control) . . . . . number(-array) ⇝ *integer* / real
Input tuple with start position(s) "n1".

▷ **Position2** (input_control) . . . . . number(-array) ⇝ *integer* / real
Input tuple with end position(s) "n2".

▷ **Substring** (output_control) . . . . . string(-array) ⇝ *string*
Characters of the string(s) from position "n1" to "n2".

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_str_first_n, tuple_str_last_n, tuple_strstr, tuple_strrstr, tuple_strlen, tuple_strchr, tuple_strrchr, tuple_split, tuple_environment

Module

Foundation

28.14 Type

tuple_is_handle (: : T : IsHandle)

Test if the internal representation of a tuple is of type handle.

tuple_is_handle tests if the internal representation of the input tuple T is of type handle. In that case the value *1* (true) is returned in IsHandle, otherwise the value *0* (false) is returned. If a tuple consists of handle elements only, IsHandle can nevertheless be 0 in case the internal representation is H_TYPE_MIXED; see tuple_type for details.

Exception: Empty input tuple

If the input tuple is empty, the operator returns *1*.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_is_handle, which can be used in an expression in the following syntax:

```
IsHandle := is_handle(T)
```

Attention

Even if all tuple elements are of type handle, tuple_is_handle returns *0* (false) if the internal representation of the tuple is of type H_TYPE_MIXED. To test if the elements of the tuple are of type handle, the operator tuple_is_handle_elem should be used.

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *handle* / real / integer / string
Input tuple.

▷ **IsHandle** (output_control) . . . . . number ⇝ *integer*
Boolean value indicating if the input tuple is of type handle.

Result

If the parameters are valid, the operator tuple_is_handle returns the value 2 (H_MSG_TRUE).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_type, tuple_is_number, tuple_is_handle_elem, tuple_is_string, tuple_sem_type

See also

tuple_is_int, tuple_is_real, tuple_is_string

Module

Foundation

tuple_is_handle_elem (: : T : IsHandle)

Test whether the elements of a tuple are of type `handle`*.*

`tuple_is_handle_elem` tests the elements of the input tuple `T` separately. For all elements of type `handle` the value *1* (true) is returned in `IsHandle`, else *0* (false) is returned.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_is_handle_elem`, which can be used in an expression in the following syntax:

```
IsHandle := is_handle_elem(T)
```

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *handle* / real / integer / string
Input tuple.

▷ **IsHandle** (output_control) . . . . . number(-array) ⇝ *integer*
Boolean values indicating if the elements of the input tuple are of type `handle`.

Result

If the parameters are valid, the operator `tuple_is_handle_elem` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_type`, `tuple_type_elem`, `tuple_is_handle`

See also

`tuple_is_mixed`, `tuple_is_int`, `tuple_is_real`, `tuple_is_string`, `tuple_is_mixed`, `tuple_is_int_elem`, `tuple_is_real_elem`, `tuple_is_number`, `tuple_is_handle`

Module

Foundation

tuple_is_int (: : T : IsInt)

Test if the internal representation of a tuple is of type `integer`*.*

`tuple_is_int` tests if the internal representation of the input tuple `T` is of type `integer`. In that case the value *1* (true) is returned in `IsInt`, otherwise the value *0* (false) is returned. If a tuple consists of `integer` elements only, `IsInt` can nevertheless be 0 in case the internal representation is H_TYPE_MIXED; see `tuple_type` for details.

Exception: Empty input tuple

If the input tuple is empty, the operator returns 1.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_is_int`, which can be used in an expression in the following syntax:

```
IsInt := is_int(T)
```

Attention

Even if all tuple elements are of type `integer`, `tuple_is_int` returns *0* (false) if the internal representation of the tuple is of type H_TYPE_MIXED. To test if the elements of the tuple are of type `integer` the operator `tuple_is_int_elem` should be used.

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **IsInt** (output_control) . . . . . number ⇝ *integer*
Is the input tuple of type `integer`?

Example

```
tuple_is_int ([3.1416,'pi',3], IsIntA)
* IsIntA = false
tuple_is_int ([1,2,3], IsIntB)
* IsIntB = true
tuple_is_int ([], IsIntC)
* IsIntC = true
```

Result

If the parameters are valid, the operator `tuple_is_int` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_type`, `tuple_is_number`, `tuple_is_int_elem`

See also

`tuple_is_real`, `tuple_is_string`, `tuple_is_handle`

Module

Foundation

tuple_is_int_elem (: : T : IsInt)

Test whether the types of the elements of a tuple are of type `integer`.

`tuple_is_int_elem` tests the elements of the input tuple `T` separately. For all elements of type `integer` the value *1* (true) is returned in `IsInt`, else *0* (false) is returned.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_is_int_elem`, which can be used in an expression in the following syntax:

```
IsInt := is_int_elem(T)
```

Attention

`tuple_is_int_elem` returns `2` (`H_MSG_TRUE`) if the internal data type of the elements of the tuple is `integer`. In contrast to `tuple_is_number` it does not return whether the elements of a tuple could be represented as `integer` values.

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *integer* / real / string / handle
Input tuple.

▷ **IsInt** (output_control) . . . . . number(-array) ⇝ *integer*
Are the elements of the input tuple of type `integer`?

Example

```
tuple_is_int_elem ([3.1416,'pi',3,3.0], IsInt)
* IsInt = [false, false, true, false]
```

Result

If the parameters are valid, the operator `tuple_is_int_elem` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_type`, `tuple_type_elem`, `tuple_is_int`

See also

`tuple_is_mixed`, `tuple_is_int`, `tuple_is_real`, `tuple_is_string`, `tuple_is_mixed`, `tuple_is_real_elem`, `tuple_is_string_elem`, `tuple_is_number`, `tuple_is_handle_elem`, `tuple_is_handle`

Module

Foundation

tuple_is_mixed (: : T : IsMixed)

Test whether a tuple is of type `mixed`.

`tuple_is_mixed` tests the input tuple `T`. If the type is `mixed`, the value *1* (true) is returned in `IsMixed`, else *0* (false) is returned. If a tuple consists of elements with equal data types only, `IsMixed` can nevertheless be 1 in case the internal representation is H_TYPE_MIXED; see `tuple_type` for details.

If the type of the tuple is `mixed` and you need all elements of the tuple `T` to be of one data type only, you can explicitly convert the tuple to a better fitting representation by using `tuple_int`, `tuple_real` or `tuple_string` with Format='s'. This improves processing speed for the converted tuple which is especially useful if the tuple is used many times afterwards.

Exception: Empty input tuple

If the input tuple is empty, the operator returns 1.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_is_mixed`, which can be used in an expression in the following syntax:

```
IsMixed := is_mixed(T)
```

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *string* / real / integer / handle
Input tuple.

▷ **IsMixed** (output_control) . . . . . number ⇝ *integer*
Is the input tuple of type `mixed`?

Example

```
tuple_is_mixed ([3.1416,'pi',3], IsMixedA)
* IsMixedA = true
tuple_is_mixed (['a','b','111'], IsMixedB)
* IsMixedB = false
tuple_is_mixed ([], IsMixedC)
* IsMixedC = true
```

Result

If the parameters are valid, the operator `tuple_is_mixed` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_type`, `tuple_is_number`

See also

`tuple_is_int`, `tuple_is_real`

Module

Foundation

tuple_is_nan_elem (: : T : IsNaN)

Check a tuple whether it represents NaNs (Not-a-number).

`tuple_is_nan_elem` checks each element of the input tuple `T` whether it represents a NaN. If it does, 1 is returned for that element, otherwise 0.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_is_nan_elem`, which can be used in an expression in the following syntax:

```
IsNaN := is_nan_elem(T)
```

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *real* / integer
Input tuple.

▷ **IsNaN** (output_control) . . . . . number ⇝ *integer*
Tuple with Boolean numbers.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_type`, `tuple_is_number`, `tuple_sem_type`

See also

`tuple_is_int`, `tuple_is_real`, `tuple_is_string`

Module

Foundation

tuple_is_number (: : T : IsNumber)

Check a tuple (of strings) whether it represents numbers.

tuple_is_number checks each element of the input tuple T whether it represents a number. If the element is a number (real or integer), 1 is returned for that element. If the element is a string, it is checked whether the string represents a number. If so, 1 is returned, otherwise 0.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_is_number, which can be used in an expression in the following syntax:

```
IsNumber := is_number(T)
```

Parameters

- ▷ **T** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string
  Input tuple.
- ▷ **IsNumber** (output_control) . . . . . integer(-array) ⇝ *integer*
  Tuple with Boolean numbers.

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_is_int, tuple_is_real, tuple_type, tuple_is_nan_elem

See also

tuple_number

Module

Foundation

tuple_is_real (: : T : IsReal)

Test if the internal representation of a tuple is of type real.

tuple_is_real tests if the internal representation of the input tuple T is of type real. In that case the value *1* (true) is returned in IsReal, otherwise the value *0* (false) is returned. If a tuple consists of real elements only, IsReal can nevertheless be 0 in case the internal representation is H_TYPE_MIXED; see tuple_type for details.

Exception: Empty input tuple

If the input tuple is empty, the operator returns 1.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_is_real, which can be used in an expression in the following syntax:

```
IsReal := is_real(T)
```

Attention

Even if all tuple elements are of type real, tuple_is_real returns *0* (false) if the internal representation of the tuple is of type H_TYPE_MIXED. To test if the elements of the tuple are of type real the operator tuple_is_real_elem should be used.

Parameters

- ▷ **T** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string / handle
  Input tuple.
- ▷ **IsReal** (output_control) . . . . . number ⇝ *integer*
  Is the input tuple of type real?

Example

```
tuple_is_real ([3.1416,'pi',3], IsRealA)
* IsRealA = false
tuple_is_real ([1.0,2.0,3.0], IsRealB)
* IsRealB = true
tuple_is_real ([], IsRealC)
* IsRealC = true
```

Result

If the parameters are valid, the operator `tuple_is_real` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_type`, `tuple_is_number`, `tuple_is_real_elem`

See also

`tuple_is_int`, `tuple_is_string`, `tuple_is_handle`

Module

Foundation

`tuple_is_real_elem` (: : T : IsReal)

Test whether the types of the elements of a tuple are of type `real`.

`tuple_is_real_elem` tests the elements of the input tuple `T` separately. For all elements of type `real` the value *1* (true) is returned in `IsReal`, else *0* (false) is returned.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_is_real_elem`, which can be used in an expression in the following syntax:

```
IsReal := is_real_elem(T)
```

Attention

`tuple_is_real_elem` returns `2` (`H_MSG_TRUE`) if the internal data type of the elements of the tuple is `real`. In contrast to `tuple_is_number` it does not return whether the elements of a tuple could be represented as `real` values.

Parameters

▷ **`T`** (input_control) ..... tuple(-array) ⇝ *real* / integer / string / handle
Input tuple.

▷ **`IsReal`** (output_control) ..... number(-array) ⇝ *integer*
Are the elements of the input tuple of type `real`?

Example

```
tuple_is_real_elem ([3.1416,'pi',3,3.0], IsReal)
* IsReal = [true, false, false, true]
```

Result

If the parameters are valid, the operator `tuple_is_real_elem` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_type`, `tuple_type_elem`, `tuple_is_real`

See also

`tuple_is_mixed`, `tuple_is_int`, `tuple_is_real`, `tuple_is_string`, `tuple_is_mixed`, `tuple_is_int_elem`, `tuple_is_string_elem`, `tuple_is_number`, `tuple_is_handle_elem`, `tuple_is_handle`

Module

Foundation

tuple_is_string (: : T : IsString)

Test if the internal representation of a tuple is of type `string`.

`tuple_is_string` tests if the internal representation of the input tuple `T` is of type `string`. In that case the value *1* (true) is returned in `IsString`, otherwise the value *0* (false) is returned. If a tuple consists of `string` elements only, `IsString` can nevertheless be 0 in case the internal representation is H_TYPE_MIXED; see `tuple_type` for details.

Exception: Empty input tuple

If the input tuple is empty, the operator returns 1.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_is_string`, which can be used in an expression in the following syntax:

```
IsString := is_string(T)
```

Attention

Even if all tuple elements are of type `string`, `tuple_is_string` returns *0* (false) if the internal representation of the tuple is of type H_TYPE_MIXED. To test if the elements of the tuple are of type `string` the operator `tuple_is_string_elem` should be used.

Parameters

▷ **T** (input_control) ..... tuple(-array) ⇝ *string* / real / integer / handle
Input tuple.

▷ **IsString** (output_control) ..... number ⇝ *integer*
Is the input tuple of type `string`?

Example

```
tuple_is_string ([3.1416,'pi',3], IsStringA)
* IsStringA = false
tuple_is_string (['a','b','111'], IsStringB)
* IsStringB = true
tuple_is_string ([], IsStringC)
* IsStringC = true
```

Result

If the parameters are valid, the operator `tuple_is_string` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_type, tuple_is_number, tuple_is_string_elem

See also

tuple_is_int, tuple_is_real, tuple_is_handle

Module

Foundation

tuple_is_string_elem (: : T : IsString)

Test whether the types of the elements of a tuple are of type `string`.

`tuple_is_string_elem` tests the elements of the input tuple `T` separately. For all elements of type `string` the value *1* (true) is returned in `IsString`, else *0* (false) is returned.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_is_string_elem`, which can be used in an expression in the following syntax:

```
IsString := is_string_elem(T)
```

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *string* / real / integer / handle
Input tuple.

▷ **IsString** (output_control) . . . . . number(-array) ⇝ *integer*
Are the elements of the input tuple of type `string`?

Example

```
tuple_is_string_elem ([3.1416,'pi',3], IsString)
* IsString = [false, true, false]
```

Result

If the parameters are valid, the operator `tuple_is_string_elem` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_type, tuple_type_elem, tuple_is_string

See also

tuple_is_mixed, tuple_is_int, tuple_is_real, tuple_is_string, tuple_is_mixed, tuple_is_int_elem, tuple_is_real_elem, tuple_is_number, tuple_is_handle_elem, tuple_is_handle

Module

Foundation

tuple_is_valid_handle (: : Handle : IsValid)

Check if a handle is valid.

tuple_is_valid_handle returns *1* in IsValid if Handle is a valid handle that has not been cleared, and *0* otherwise.

If multiple handles are passed in Handle, a validity value will be returned for each of them in IsValid.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

Parameters

▷ **Handle** (input_control) . . . . . tuple(-array) ⇝ *handle*
The handle to check for validity.

▷ **IsValid** (output_control) . . . . . integer(-array) ⇝ *integer*
The validity of the handle, *1* or *0*.

Result

If the parameters are valid, the operator tuple_is_valid_handle returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

tuple_sem_type (: : T : SemType)

Return the semantic type of a tuple.

tuple_sem_type returns the semantic type of the input tuple T. The type is returned as a string value in the output parameter SemType.

If T contains only valid handles of the same type, the corresponding semantic type of the handles is returned (e.g., *'matrix'*). If it contains only valid or invalid handles, or handles of different type, *'handle'* is returned. If T contains elements of different types, *'any'* is returned.

Exception: Empty input tuple

If the input tuple is empty, the operator returns *'any'*.

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_sem_type, which can be used in an expression in the following syntax:

```
SemType := sem_type(T)
```

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *handle* / real / integer / string
Input tuple.

▷ **SemType** (output_control) . . . . . string ⇝ *string*
Semantic type of the input tuple as a string.

Example

```
create_matrix (3, 3, 0, MatrixID)
tuple_sem_type (MatrixID, SemType)
```

```
* SemType == 'matrix' (Handle of specific type)
clear_handle (MatrixID)
tuple_sem_type (MatrixID, SemType)
* SemType == 'handle' (Cleared handle)
tuple_sem_type ([MatrixID,123], SemType)
* SemType == 'any' (Mixed types)
```

Result

If the parameters are valid, the operator `tuple_sem_type` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_is_int`, `tuple_is_number`, `tuple_is_real`, `tuple_is_string`, `tuple_type`

See also

`get_obj_class`

Module

Foundation

tuple_sem_type_elem (: : T : SemTypes)

Return the semantic type of the elements of a tuple.

`tuple_sem_type_elem` returns the semantic type of each element of the input tuple `T`. The semantic type is returned as a tuple of string values in the output parameter `SemTypes`.

If an element is a valid handle, the corresponding semantic type of the handle is returned. For an invalid or cleared handle, *'handle'* is returned.

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_sem_type_elem`, which can be used in an expression in the following syntax:

```
SemTypes := sem_type_elem(T)
```

Parameters

- ▷ **T** (input_control) . . . . . tuple(-array) ⇝ *handle* / real / integer / string
  Input tuple.
- ▷ **SemTypes** (output_control) . . . . . string(-array) ⇝ *string*
  Semantic types of the elements of the input tuple as strings.

Example

```
create_matrix (3, 3, 0, MatrixID)
tuple_sem_type_elem ([MatrixID,1,1.0,'HALCON',HNULL], SemTypeElem)
* SemTypeElem == ['matrix','integer','real','string','handle']
```

Result

If the parameters are valid, the operator `tuple_sem_type_elem` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

tuple_type, tuple_is_mixed, tuple_is_int, tuple_is_real, tuple_is_string, tuple_is_mixed, tuple_is_int_elem, tuple_is_real_elem, tuple_is_string_elem, tuple_type_elem, tuple_sem_type

See also

get_obj_class

Module

Foundation

tuple_type (: : T : Type)

Return the type of a tuple.

tuple_type returns the type of the input tuple T. The type is returned as an integer value in the output parameter Type. In HDevelop corresponding constants are defined:

- H_TYPE_INT (1).
- H_TYPE_REAL (2).
- H_TYPE_STRING (4).
- H_TYPE_MIXED (8).
- H_TYPE_HANDLE (16).
- H_TYPE_ANY (31).

H_TYPE_MIXED is returned in the following two cases:

- some elements of the tuple have different data types, e.g., real and integer.
- the tuple T has undergone operations which modified the data type of single elements. Such tuples will generally not be optimized automatically (because of runtime reasons) even if the data types of all elements become equal in subsequent operations. See tuple_is_mixed and the following example on how to optimize such tuples.

Exception: Empty input tuple

If the input tuple is empty, the operator returns 31 (H_TYPE_ANY).

HDevelop In-line Operation

HDevelop provides an in-line operation for tuple_type, which can be used in an expression in the following syntax:

```
Type := type(T)
```

Attention

tuple_type returns the internal data type of the tuple. In contrast to tuple_is_number it does not return whether a tuple could be represented as a tuple of a certain type.

Parameters

▷ **T** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string / handle
Input tuple.

▷ **Type** (output_control) . . . . . number ⇝ *integer*
Type of the input tuple as an integer number.

Example

```
tuple_type ([3.1416,'pi',3], TypeA)
* TypeA = H_TYPE_MIXED
tuple_type (['a','b','111'], TypeB)
* TypeB = H_TYPE_STRING
tuple_type ([], TypeC)
* TypeC = H_TYPE_ANY
tuple_type (HNULL, TypeD)
* TypeD = H_TYPE_HANDLE

TupleInt := [1,2,3,4]
TupleReal := [42.0]
TupleConcat := [TupleInt, TupleReal]
tuple_type (TupleConcat, TypeConcat)
* TypeConcat = H_TYPE_MIXED
* Now set 42.0 to 42
TupleConcat[4] := 42
tuple_type (TupleConcat, TypeConcat2)
* TypeConcat2 = H_TYPE_MIXED
* TupleConcat now consists of integers only, but the
* internal representation hasn't been updated. Optimize
* it by converting the tuple explicitly to an integer
* tuple.
tuple_int (TupleConcat, TupleConcatInt)
tuple_type (TupleConcatInt, TypeConcatInt)
* TypeConcatInt = H_TYPE_INT
```

Result

If the parameters are valid, the operator `tuple_type` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_is_int`, `tuple_is_number`, `tuple_is_real`, `tuple_is_string`, `tuple_sem_type`

See also

`get_obj_class`

Module

Foundation

tuple_type_elem (: : T : Types)

Return the types of the elements of a tuple.

`tuple_type_elem` returns the types of the elements of the input tuple `T`. The types are returned separately as integer values in the output parameter `Types`. In HDevelop the corresponding constants are defined:

- H_TYPE_INT (1).
- H_TYPE_REAL (2).
- H_TYPE_STRING (4).
- H_TYPE_HANDLE (16).

Exception: Empty input tuple

If the input tuple is empty, the operator returns an empty tuple.

HDevelop In-line Operation

HDevelop provides an in-line operation for `tuple_type_elem`, which can be used in an expression in the following syntax:

```
Types := type_elem(T)
```

Parameters

▷ **`T`** (input_control) . . . . . tuple(-array) ⇝ *real* / integer / string / handle
Input tuple.

▷ **`Types`** (output_control) . . . . . number(-array) ⇝ *integer*
Types of the elements of the input tuple as integer values.

Example

```
tuple_type_elem ([3.1416,'pi',3], Types)
* Types = [H_TYPE_REAL, H_TYPE_STRING, H_TYPE_INT]
```

Result

If the parameters are valid, the operator `tuple_type_elem` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: independent (runs in parallel even with exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Alternatives

`tuple_type`, `tuple_is_mixed`, `tuple_is_int`, `tuple_is_real`, `tuple_is_string`, `tuple_is_mixed`, `tuple_is_int_elem`, `tuple_is_real_elem`, `tuple_is_string_elem`

See also

`get_obj_class`, `tuple_is_number`

Module

Foundation

Chapter 29

XLD

29.1 Access

get_contour_xld (Contour : : : Row, Col)

Return the coordinates of an XLD contour.

get_contour_xld returns the following values of the XLD contour Contour:

Row	Row coordinate of the contour's points
Col	Column coordinate of the contour's points

Parameters

▷ **Contour** (input_object) . . . . . xld_cont ⇝ *object*
Input XLD contour.

▷ **Row** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinate of the contour's points.

▷ **Col** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinate of the contour's points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix

See also

get_contour_attrib_xld, query_contour_attribs_xld, get_contour_global_attrib_xld, query_contour_global_attribs_xld, gen_contour_polygon_xld

Module

Foundation

get_lines_xld (Polygon : : : BeginRow, BeginCol, EndRow, EndCol, Length, Phi)

Return an XLD polygon's data (as lines).

get_lines_xld returns the XLD polygon Polygon as a set of lines. The following values are returned:

BeginRow:	Rows coordinates of the lines' start points
BeginCol:	Columns coordinates of the lines' start points
EndRow:	Row coordinates of the lines' end points
EndCol:	Column coordinates of the lines' end points
Length:	Lengths of the line segments
Phi:	Angles to the normal vectors of the line segments

Parameters

- ▷ **Polygon** (input_object) ..... xld_poly(-array) ⇝ *object*
  Input XLD polygons.
- ▷ **BeginRow** (output_control) ..... line.begin.y-array ⇝ *real*
  Row coordinates of the lines' start points.
- ▷ **BeginCol** (output_control) ..... line.begin.x-array ⇝ *real*
  Column coordinates of the lines' start points.
- ▷ **EndRow** (output_control) ..... line.end.y-array ⇝ *real*
  Column coordinates of the lines' end points.
- ▷ **EndCol** (output_control) ..... line.end.x-array ⇝ *real*
  Column coordinates of the lines' end points.
- ▷ **Length** (output_control) ..... real-array ⇝ *real*
  Lengths of the line segments.
- ▷ **Phi** (output_control) ..... angle.rad-array ⇝ *real*
  Angles to the normal vectors of the line segments.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_polygons_xld

Alternatives

get_polygon_xld

Module

Foundation

get_parallels_xld (Parallels : : : Row1, Col1, Length1, Phi1, Row2, Col2, Length2, Phi2)

Return an XLD parallel's data (as lines).

get_parallels_xld returns for the first XLD parallel in Parallels the following values:

Row1:	Row coordinates of the points on polygon P1
Col1:	Column coordinates of the points on polygon P1
Length1:	Lengths of the line segments on polygon P1
Phi1:	Angles to the normal vectors of the line segments on polygon P1
Row2:	Row coordinates of the points on polygon P2
Col2:	Column coordinates of the points on polygon P2
Length2:	Lengths of the line segments on polygon P2
Phi2:	Angles to the normal vectors of the line segments on polygon P2

Parameters

▷ **Parallels** (input_object) . . . . . xld ⇝ *object*
Input XLD parallels.

▷ **Row1** (output_control) . . . . . polygon.y-array ⇝ *real*
Row coordinates of the points on polygon P1.

▷ **Col1** (output_control) . . . . . polygon.x-array ⇝ *real*
Column coordinates of the points on polygon P1.

▷ **Length1** (output_control) . . . . . real-array ⇝ *real*
Lengths of the line segments on polygon P1.

▷ **Phi1** (output_control) . . . . . angle.rad-array ⇝ *real*
Angles to the normal vectors of the line segments on polygon P1.

▷ **Row2** (output_control) . . . . . polygon.y-array ⇝ *real*
Row coordinates of the points on polygon P2.

▷ **Col2** (output_control) . . . . . polygon.x-array ⇝ *real*
Column coordinates of the points on polygon P2.

▷ **Length2** (output_control) . . . . . real-array ⇝ *real*
Lengths of the line segments on polygon P2.

▷ **Phi2** (output_control) . . . . . angle.rad-array ⇝ *real*
Angles to the normal vectors of the line segments on polygon P2.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_parallels_xld

See also

get_polygon_xld, get_lines_xld

Module

Foundation

get_polygon_xld (Polygon : : : Row, Col, Length, Phi)

Return an XLD polygon's data.

get_polygon_xld returns the following values of the XLD polygon Polygon:

Row	Row coordinates of the polygons' points
Col	Column coordinates of the polygons' points
Length	Lengths of the line segments between points i and $i+1$, respectively
Phi	Angles to the normal vectors of the line segments between points i and $i+1$, respectively

Parameters

▷ **Polygon** (input_object) . . . . . xld_poly ⇝ *object*
Input XLD polygon.

▷ **Row** (output_control) . . . . . point.y-array ⇝ *real*
Row coordinates of the polygons' points.

▷ **Col** (output_control) . . . . . point.x-array ⇝ *real*
Column coordinates of the polygons' points.

▷ **Length** (output_control) . . . . . real-array ⇝ *real*
Lengths of the line segments.

▷ **Phi** (output_control) . . . . . angle.rad-array ⇝ *real*
Angles to the normal vectors of the line segments.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_polygons_xld

Alternatives

get_lines_xld

Module

Foundation

29.2 Creation

gen_circle_contour_xld (: ContCircle : Row, Column, Radius, StartPhi, EndPhi, PointOrder, Resolution :)

Create XLD contours corresponding to circles or circular arcs.

gen_circle_contour_xld creates one or more circular arcs or closed circles. Circles are specified by their center (Row, Column) and their Radius. In addition to that, circular arcs are characterized by the angle of the start point StartPhi, the angle of the end point EndPhi, and the PointOrder along the boundary. The resolution of the resulting contours ContCircle is controlled via Resolution containing the Euclidean distance in pixel between neighboring contour points. In general, the distance between the second to last and the last point is smaller than Resolution.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

▷ **ContCircle** (output_object) . . . . . xld_cont(-array) ⇝ *object*
Resulting contours.

▷ **Row** (input_control) . . . . . circle.center.y(-array) ⇝ *real*
Row coordinate of the center of the circles or circular arcs.
Default: 200.0

▷ **Column** (input_control) . . . . . circle.center.x(-array) ⇝ *real*
Column coordinate of the center of the circles or circular arcs.
Default: 200.0

▷ **Radius** (input_control) . . . . . circle.radius(-array) ⇝ *real*
Radius of the circles or circular arcs.
Default: 100.0
Restriction: Radius > 0

▷ **StartPhi** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Angle of the start points of the circles or circular arcs [rad].
Default: 0.0

▷ **EndPhi** (input_control) . . . . . angle.rad(-array) ⇝ *real*
Angle of the end points of the circles or circular arcs [rad].
Default: 6.28318

▷ **PointOrder** (input_control) . . . . . string(-array) ⇝ *string*
Point order along the circles or circular arcs.
Default: 'positive'
List of values: PointOrder ∈ {'positive', 'negative'}

▷ **Resolution** (input_control) . . . . . real ⇝ *real*
Distance between neighboring contour points.
Default: 1.0
Restriction: Resolution >= 0.00001

Example

```
draw_circle (WindowHandle, Row, Column, Radius)
gen_circle_contour_xld (ContCircle, Row, Column, Radius, 0, \
                        rad(360) , 'positive', 1.0)
gen_region_contour_xld (ContCircle, Region, 'filled')
```

Result

`gen_circle_contour_xld` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`draw_circle`

Possible Successors

`disp_xld`

Alternatives

`gen_ellipse_contour_xld`

Module

Foundation

gen_contour_nurbs_xld (: Contour : Rows, Cols, Knots, Weights, Degree, MaxError, MaxDistance :)

Transform a NURBS curve into an XLD contour.

`gen_contour_nurbs_xld` generates an XLD `Contour` that approximates a NURBS curve (non-uniform rational B-Spline curve). The NURBS curve is specified by its `Degree`, a control polygon (via `Rows` and `Cols`), a `Weights` vector and a `Knots` vector.

The `Degree` of the curve determines the grade of differentiability of the curve. The curve lies within the convex hull of its control polygon (`Rows`,`Cols`). The weights in `Weights` determine how much the curve is attracted by the individual control points (`Rows`,`Cols`). A `Weights` vector with equal weights for all control points is generated if *'auto'* is chosen. The `Knots` vector describes the domain of the curve. Typically, this parameter can be set to *'auto'*.

The accuracy of the generated `Contour` depends on the two parameters `MaxDistance` and `MaxError`. `MaxDistance` limits the maximum distance of two subsequent `Contour` points. The maximum distance between the generated `Contour` and the actual NURBS curve is limited to `MaxError`. By default the approximation must fulfill both constraints, but it is possible to set one of these parameters to *'omit'*. The lower `MaxError` and `MaxDistance` are chosen the better is the approximation of the curve. Note that all `Contour` points lie exactly on the curve (except for numerical inaccuracies).

Definition

A NURBS curve C of degree p is given by

$$C(u) = \frac{\sum_{i=0}^{n} N_{i,p}(u) w_i P_i}{\sum_{i=0}^{n} N_{i,p}(u) w_i}, u \in [u_p, u_n].$$

The basis functions $N_{i,p}$ are defined by

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)$$

Thus, each NURBS curve is determined by its `Degree` p, its control Polygon $P_0, ..., P_{n-1}$ (`Rows` and `Cols`), its weight vector $w = (w_0, ..., w_{n-1})$ (`Weights`), and its knot vector $u = (u_0, ..., u_{n+p})$ (`Knots`).

The grade of differentiability of the curve is set via the degree p. At least $p + 1$ control points P_i are required to form a valid curve. The knot vector u describes the domain of the curve. Its length must be $n + p + 1$ and its entries must be monotonically increasing. Every entry u_i stands for a knot between the curve segments $[u_{i-1}, u_i)$ and $[u_i, u_{i+1})$. Each curve segment $[u_i, u_{i+1})$ lies within the convex hull of the control points $P_{i-p}, .., P_i$. For every curve segment of length zero, the differentiability at knot k decreases. The number of occurrences of each knot in the knot vector is called multiplicity. The maximum multiplicity of one knot is p. Only the start and the end point's multiplicity can be $p + 1$ (which in fact is the case for most curves, because in this case start and end point of the curve correspond to the first respectively last point of the control polygon). If start and/or end point do not have full multiplicity, the domain of the curve is restricted to $[u_p, u_n]$. The weights w_i determine the influence of the individual control points P_i on the curve.

Closed Curves: If start and end point are equal ($P_0 = P_{n-1}$) and `Knots` is set to *'auto'*, the curve is closed automatically such that all knots feature multiplicity one. Note that in this case the length of `Weights` is required to be $n - 1$.

Parameters

▷ **`Contour`** (output_object) . . . . . xld_cont ⇝ *object*
The contour that approximates the NURBS curve.

▷ **`Rows`** (input_control) . . . . . coordinates.y-array ⇝ *real*
Row coordinates of the control polygon.
Number of elements: Rows == Cols && Rows >= Degree + 1

▷ **`Cols`** (input_control) . . . . . coordinates.x-array ⇝ *real*
Column coordinates of the control polygon.
Number of elements: Cols == Rows && Cols >= Degree + 1

▷ **`Knots`** (input_control) . . . . . string(-array) ⇝ *string* / real
The knot vector u.
Default: 'auto'
Suggested values: `Knots` ∈ {'auto'}

▷ **`Weights`** (input_control) . . . . . string(-array) ⇝ *string* / real
The weight vector w.
Number of elements: Weights == Rows
Default: 'auto'
Suggested values: `Weights` ∈ {'auto'}
Restriction: Weights > 0.0

▷ **`Degree`** (input_control) . . . . . integer ⇝ *integer*
The degree p of the NURBS curve.
Default: 3
Suggested values: `Degree` ∈ {2, 3, 4, 5}
Restriction: Degree >= 2

▷ **`MaxError`** (input_control) . . . . . real ⇝ *real* / string
Maximum distance between the NURBS curve and its approximation.
Default: 1.0
Suggested values: `MaxError` ∈ {'omit', 1.0, 2.0, 3.0, 4.0, 5.0}
Restriction: MaxError > 0.0

▷ **`MaxDistance`** (input_control) . . . . . real ⇝ *real* / string
Maximum distance between two subsequent `Contour` points.
Default: 5.0
Suggested values: `MaxDistance` ∈ {'omit', 1.0, 2.0, 3.0, 4.0, 5.0}
Restriction: MaxDistance > 0.0

Example

```
* use a polygon XLD contour as control polygon and approximate
* the NURBS curve as contour

get_contour_xld(Polygon, Row, Col)
gen_contour_nurbs_xld(NURBSContour, Row, Col, 'auto', 'auto', 3, 1.0, 10.0)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_nurbs_interp

Possible Successors

gen_polygons_xld

References

L. Piegl, W. Tiller: "The NURBS Book", 2nd Edition, Springer, 1997.

Module

Foundation

gen_contour_polygon_rounded_xld (: Contour : Row, Col, Radius, SamplingInterval :)

Generate an XLD contour with rounded corners from a polygon (given as tuples).

gen_contour_polygon_rounded_xld generates an XLD contour Contour with rounded corners from a polygon given in the tuples Row and Col. The rounded corners are created in form of arcs defined by Radius. For every specified point of the polygon there must be a corresponding rounding radius defined. In case of a closed polygon the first and the last point have to be defined identical with the radii of these points being equal. In contrast, the radii of the first and last point of open polygons are ignored. Finally, the SamplingInterval parameter defines the distance of the control points of the formed contour Contour.

Parameters

▷ **Contour** (output_object) . . . . . xld_cont ⇝ *object*
Resulting contour.

▷ **Row** (input_control) . . . . . coordinates.y-array ⇝ *real* / integer
Row coordinates of the polygon.
Default: [20,80,80,20,20]
Suggested values: Row ∈ {0, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500}

▷ **Col** (input_control) . . . . . coordinates.x-array ⇝ *real* / integer
Column coordinates of the polygon.
Default: [20,20,80,80,20]
Suggested values: Col ∈ {0, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500}

▷ **Radius** (input_control) . . . . . number-array ⇝ *real* / integer
Radii of the rounded corners.
Default: [20,20,20,20,20]
Suggested values: Radius ∈ {0, 1, 2, 5, 10, 20, 50}

▷ **SamplingInterval** (input_control) . . . . . number ⇝ *real* / integer
Distance of the samples.
Default: 1.0
Suggested values: SamplingInterval ∈ {0.5, 1.0, 2.0, 5.0}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_region_contour

Possible Successors

smooth_contours_xld, gen_polygons_xld

See also

gen_contour_polygon_xld, gen_contours_skeleton_xld

Module

Foundation

gen_contour_polygon_xld (: Contour : Row, Col :)

Generate an XLD contour from a polygon (given as tuples).

gen_contour_polygon_xld generates an XLD contour Contour from a polygon given in the tuples Row and Col. This operator is useful if contours have been obtained from routines outside the core library, but higher level operators, e.g., polygon approximation and extraction of parallels, are to be performed on the contours.

Parameters

▷ **Contour** (output_object) ..... xld_cont ⇝ *object*
Resulting contour.

▷ **Row** (input_control) ..... coordinates.y-array ⇝ *real* / integer
Row coordinates of the polygon.
Default: [0,1,2,2,2]
Suggested values: Row ∈ {0, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500}

▷ **Col** (input_control) ..... coordinates.x-array ⇝ *real* / integer
Column coordinates of the polygon.
Default: [0,0,0,1,2]
Suggested values: Col ∈ {0, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

get_region_contour

Possible Successors

smooth_contours_xld, gen_polygons_xld

See also

gen_contours_skeleton_xld, get_contour_xld

Module

Foundation

gen_contour_region_xld (Regions : Contours : Mode :)

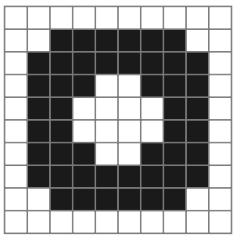
Generate XLD contours from regions.

gen_contour_region_xld generates XLD contours Contours from the regions given in Regions. This operator is useful if regions have been obtained from segmentation operations, but higher level operators, e.g.,

polygon approximation and extraction of parallels, are to be performed on their boundaries. For each connected component of the input regions a closed contour of the boundary is generated. The parameter Mode can take on the following values:

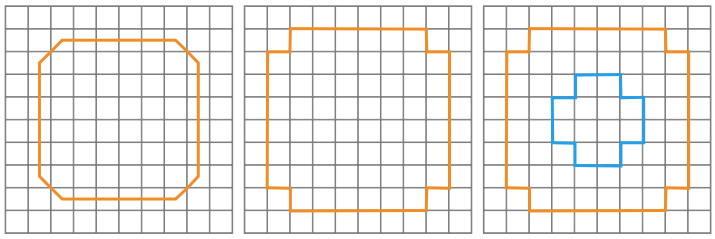
- *'center'*: The centers of the border pixels are used as contour points.
- *'border'*: The outer border of the border pixels is used as contour points.
- *'border_holes'*: In addition to the outer border of the input region you get the contours of all holes.

The difference between the modes is demonstrated using the following region:



Example region.

Computing the contour with the different values available for Mode returns the following result contours:



(1) (2) (3)

Result for *'center'* (1), *'border'* (2), and *'border_holes'* mode (3).

This means, for example, that contours generated with *'border'* will in general have a much larger Euclidean length (see length_xld) than contours generated with *'center'*. This results from the fact that for diagonal border elements *'border'* uses two contour segments of length 1 each, whereas *'center'* uses a single segment of length $\sqrt{2}$. Other features, e.g., the area (see area_center_xld), will obviously also have different values.

Parameters

▷ **Regions** (input_object) . . . . . region(-array) ⇝ *object*
Input regions.

▷ **Contours** (output_object) . . . . . xld_cont(-array) ⇝ *object*
Resulting contours.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Mode of contour generation.
Default: 'border'
List of values: Mode ∈ {'border', 'border_holes', 'center'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

smooth_contours_xld, gen_polygons_xld

Alternatives

gen_contour_polygon_xld, get_region_contour

See also

gen_contours_skeleton_xld

Module

Foundation

gen_contours_skeleton_xld (Skeleton : Contours : Length, Mode :)

Convert a skeleton into XLD contours.

gen_contours_skeleton_xld converts the input skeleton (e.g., edges) Skeleton, which is assumed to contain mostly one pixel wide regions (see skeleton), into XLD contours.

The algorithm first attempts to transform the regions to contain only line segments in 8-neighborhood. In a second step, the junction points are labeled. After this, gen_contours_skeleton_xld tries to generate contours which end in junction or end points, in particular for junction points of the following configurations (in all four possible rotations):

1	0	1	1	0	1	1	0	0	1	0	0	0	1	0	0	1	0
0	2	0	0	2	0	0	2	1	0	2	1	0	2	1	1	2	1
0	0	1	1	0	1	0	1	0	1	0	0	0	1	0	0	1	0

where 0 = background, 1 = foreground, and 2 = junction point.

After this, all contours having at least Length points are returned. Since contours are split at junction points, possibly long contours may be split into several short segments because of short adjacent lines, even if they are longer than Length points, if Mode = *'filter'* was selected. This can be avoided by setting Mode = *'generalize1'*. In this case, the contours are generated as if the segments shorter than Length were not contained in the input region. In order to preserve line segments, which are split into very short segments by the crossing of short lines, Mode = *'generalize2'* can be selected. In this case, line segments are preserved if they end in two junction points, even if they are shorter than Length.

Parameters

▷ **Skeleton** (input_object) . . . . . region ⇝ *object*
Skeleton of which the contours are to be determined.

▷ **Contours** (output_object) . . . . . xld_cont-array ⇝ *object*
Resulting contours.

▷ **Length** (input_control) . . . . . integer ⇝ *integer*
Minimum number of points a contour has to have.
Default: 1
Suggested values: Length ∈ {1, 2, 3, 5, 10, 20}

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Contour filter mode.
Default: 'filter'
List of values: Mode ∈ {'filter', 'generalize1', 'generalize2'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

skeleton

Possible Successors

smooth_contours_xld, get_contour_xld, gen_polygons_xld

See also

edges_image, threshold, get_region_contour

Module

Foundation

gen_cross_contour_xld (: Cross : Row, Col, Size, Angle :)

Generate one XLD contour in the shape of a cross for each input point.

gen_cross_contour_xld generates an XLD contour in the shape of a cross for each input point (Row,Col). Conceptually, the contour consists of two lines of length Size, which intersect exactly in the input point. Their orientation is determined by Angle. The cross is returned in Cross.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

▷ **Cross** (output_object) ..... xld_cont(-array) ⇝ *object*
Generated XLD contours.

▷ **Row** (input_control) ..... point.y(-array) ⇝ *real* / integer
Row coordinates of the input points.

▷ **Col** (input_control) ..... point.x(-array) ⇝ *real* / integer
Column coordinates of the input points.
Restriction: number(Col) == number(Row)

▷ **Size** (input_control) ..... number ⇝ *real* / integer
Length of the cross bars.
Default: 6.0
Suggested values: Size ∈ {4.0, 6.0, 8.0, 10.0}
Restriction: 0.0 <= Size

▷ **Angle** (input_control) ..... angle.rad ⇝ *real*
Orientation of the crosses.
Default: 0.785398
Suggested values: Angle ∈ {0.0, 0.785398}

Result

gen_cross_contour_xld returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

points_foerstner, points_harris, critical_points_sub_pix, local_max_sub_pix, local_min_sub_pix, saddle_points_sub_pix

Alternatives

disp_cross

Module

Foundation

gen_ellipse_contour_xld (: ContEllipse : Row, Column, Phi, Radius1, Radius2, StartPhi, EndPhi, PointOrder, Resolution :)

Create an XLD contour that corresponds to an elliptic arc.

gen_ellipse_contour_xld creates one or more elliptic arcs or closed ellipses. Ellipses are specified by their center (Row, Column), the orientation of the main axis Phi, the length of the larger half axis Radius1, and the length of the smaller half axis Radius2. In addition to that, elliptic arcs are characterized by the angle of the start point StartPhi, the angle of the end point EndPhi, and the point order PointOrder along the boundary. Both the angle of the start point and the angle of the end point are measured from the positive main

axis specified with Phi. They correspond to the smallest surrounding circle of the ellipse. The actual start and end point of the ellipse is the intersection of the ellipse with the orthogonal projection of the corresponding circle point onto the main axis. Both angles refer to the coordinate system of the ellipse, i.e. relative to the main axis and in a mathematical positive direction. Thus, the two main poles correspond to the angles 0 and π, the two minor poles to the angles $\pi/2$ and $3\pi/2$. To create a closed ellipse the values 0 and 2π (with positive point order) have to be passed to the operator. All angles Phi, StartPhi, EndPhi take arbitrary values and are mapped internally to the interval $[0, 2\pi]$. The resolution of the resulting contours ContEllipse is controlled via Resolution containing the maximum Euclidean distance between neighboring contour points. The resolution is set to the smallest valid value, if the input value falls below this value.

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

- ▷ **ContEllipse** (output_object) ..... xld_cont(-array) ⇝ *object*
  Resulting contour.
- ▷ **Row** (input_control) ..... ellipse.center.y(-array) ⇝ *real*
  Row coordinate of the center of the ellipse.
  Default: 200.0
- ▷ **Column** (input_control) ..... ellipse.center.x(-array) ⇝ *real*
  Column coordinate of the center of the ellipse.
  Default: 200.0
- ▷ **Phi** (input_control) ..... ellipse.angle.rad(-array) ⇝ *real*
  Orientation of the main axis [rad].
  Default: 0.0
- ▷ **Radius1** (input_control) ..... ellipse.radius1(-array) ⇝ *real*
  Length of the larger half axis.
  Default: 100.0
  Restriction: Radius1 > 0
- ▷ **Radius2** (input_control) ..... ellipse.radius2(-array) ⇝ *real*
  Length of the smaller half axis.
  Default: 50.0
  Restriction: Radius2 >= 0 && Radius2 <= Radius1
- ▷ **StartPhi** (input_control) ..... angle.rad(-array) ⇝ *real*
  Angle of the start point on the smallest surrounding circle [rad].
  Default: 0.0
- ▷ **EndPhi** (input_control) ..... angle.rad(-array) ⇝ *real*
  Angle of the end point on the smallest surrounding circle [rad].
  Default: 6.28318
- ▷ **PointOrder** (input_control) ..... string(-array) ⇝ *string*
  point order along the boundary.
  Default: 'positive'
  List of values: PointOrder ∈ {'positive', 'negative'}
- ▷ **Resolution** (input_control) ..... real ⇝ *real*
  Resolution: Maximum distance between neighboring contour points.
  Default: 1.5
  Restriction: Resolution >= 1.192e-7

Example

```
draw_ellipse(WindowHandle,Row,Column,Phi,Radius1,Radius2)
gen_ellipse_contour_xld(Ellipse,Row,Column,Phi,Radius1,Radius2,0,6.28319, \
                        'positive',1.5)
length_xld(Ellipse,Length)
* Transform StartPhi so that the start point of the ellipse actually
* intersects the line through origin at the angle StartPhi measured from
* the positive main axis
affine_trans_point_2d ([1.0,0.0,0.0,0.0,1.0*Radius2/Radius1,0.0], \
```

```
                        sin(StartPhi), cos(StartPhi), Qx, Qy)
StartPhiEllipse := atan2(Qx,Qy)
```

Result

`gen_ellipse_contour_xld` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`draw_ellipse`

Possible Successors

`disp_xld`, `get_points_ellipse`

Alternatives

`gen_circle_contour_xld`

Module

Foundation

gen_nurbs_interp (: : Rows, Cols, Tangents, Degree : CtrlRows, CtrlCols, Knots)

Create control data of a NURBS curve that interpolates given points.

`gen_nurbs_interp` creates the NURBS control data `CtrlRows`, `CtrlCols`, and `Knots` of a NURBS curve that interpolates the input points (`Rows`, `Cols`). If the input point list is not closed (i.e., the first point is identical to the last point), the tangents of the first and last point must be given in `Tangents` in the order $[drow\_0, dcol\_0, drow\_n - 1, dcol\_n - 1]$ (for closed point lists, `Tangents` must be an empty tuple). Furthermore the `Degree` of the NURBS curve must be specified.

The output of `gen_nurbs_interp` can be used directly with `gen_contour_nurbs_xld` with the weights vector set to *'auto'*.

See the documentation of `gen_contour_nurbs_xld` for more information on NURBS curves.

Parameters

▷ **Rows** (input_control) . . . . . coordinates.y-array ⇝ *real*
Row coordinates of input point list.
Number of elements: Rows == Cols

▷ **Cols** (input_control) . . . . . coordinates.x-array ⇝ *real*
Column coordinates of input point list.
Number of elements: Cols == Rows

▷ **Tangents** (input_control) . . . . . real-array ⇝ *real*
Tangents at first and last point.
Default: []

▷ **Degree** (input_control) . . . . . integer ⇝ *integer*
Order of the output curve.
Default: 3
Suggested values: `Degree` ∈ {2, 3, 4, 5}
Restriction: Degree >= 2

▷ **CtrlRows** (output_control) . . . . . coordinates.y-array ⇝ *real*
Row coordinates of the control polygon.

▷ **CtrlCols** (output_control) . . . . . coordinates.x-array ⇝ *real*
Column coordinates of the control polygon.
Number of elements: Cols == Rows

▷ **Knots** (output_control) . . . . . real-array ⇝ *real*
The knot vector of the output curve.

Result

If all input parameters are correct `gen_nurbs_interp` returns the value `2` (`H_MSG_TRUE`).

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Successors

`gen_contour_nurbs_xld`

See also

`draw_nurbs_interp`, `draw_nurbs_interp_mod`

References

L. Piegl, W. Tiller: "The NURBS Book", 2nd Edition, Springer, 1997.

Module

Foundation

gen_parallels_xld (Polygons : Parallels : Len, Dist, Alpha, Merge :)

Extract parallel XLD polygons.

`gen_parallels_xld` examines the XLD polygons passed in `Polygons` for parallelism. The resulting parallel polygons are returned in `Parallels`. If the parameter `Merge` is set to *'true'*, adjacent parallel polygons are returned in a single parallel relation. Otherwise, one parallel relation is returned for each pair of parallel line segments.

Only polygon segments longer than `Len` are taken into account. Whether two polygon segments are parallel depends on a maximum allowed angle difference `Alpha` (in radians). Furthermore, the projections of the two segments onto their angle bisector have to overlap. Finally, the overlap on the angle bisector is reprojected onto the two segments. If the maximal distance of the reprojections is smaller than `Dist`, the segments are stored in `Parallels`.

As a side effect, a quality factor is calculated for each pair of parallels, compare `mod_parallels_xld`. It is based on the normalized angular difference and the normalized length of the overlap:

$$q = \frac{\pi/2 - \delta\alpha}{\pi/2} \frac{2overlap}{len_1 + len_2} \qquad \text{with} \quad 0 \leq q \leq 1$$

Here, $\delta\alpha$ is the angle difference of the polygon segments, $overlap$ is the length of the overlap, len_1 and len_2 the lengths of the polygon segments, and q the resulting quality factor. The quality factor is a measure of parallelism (the larger its value, the "more parallel" the polygons). Finally, the quality factors of all parallel polygon segments contained in a single polygon are added, weighted with their length of the overlapping area. Note that you can query the range for the quality factor with `info_parallels_xld`.

Parameters

▷ **Polygons** (input_object) . . . . . xld_poly-array ⇝ *object*
Input polygons.

▷ **Parallels** (output_object) . . . . . xld_para-array ⇝ *object*
Parallel polygons.

▷ **Len** (input_control) . . . . . number ⇝ *real* / integer
Minimum length of the individual polygon segments.
Default: 10.0
Suggested values: `Len` ∈ {5.0, 10.0, 15.0, 20.0}
Restriction: Len > 0.0

▷ **Dist** (input_control) . . . . . number ⇝ *real* / integer
Maximum distance between the polygon segments.
Default: 30.0
Suggested values: Dist ∈ {20.0, 25.0, 30.0, 40.0, 50.0, 75.0}
Restriction: Dist > 0.0

▷ **Alpha** (input_control) . . . . . number ⇝ *real* / integer
Maximum angle difference of the polygon segments.
Default: 0.15
Suggested values: Alpha ∈ {0.05, 0.10, 0.15, 0.20, 0.30}
Restriction: 0 <= Alpha && Alpha <= pi / 2

▷ **Merge** (input_control) . . . . . string ⇝ *string*
Should adjacent parallel relations be merged?
Default: 'true'
List of values: Merge ∈ {'true', 'false'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_polygons_xld

Possible Successors

mod_parallels_xld, get_parallels_xld, info_parallels_xld

Module

Foundation

gen_polygons_xld (Contours : Polygons : Type, Alpha :)

Approximate XLD contours by polygons.

gen_polygons_xld approximates XLD contours (Contours) by polygons. The type of the approximation can be set by Type. The threshold for the approximation is set via Alpha. The procedure is able to process open as well as closed contours. The resulting approximating XLD polygons are returned in Polygons.

Contours can be approximated by the algorithm of Ramer, which approximates contours such that the Euclidean distance of the approximating polygon to the contour is at most Alpha pixel units.

Parameters

▷ **Contours** (input_object) . . . . . xld_cont-array ⇝ *object*
Contours to be approximated.

▷ **Polygons** (output_object) . . . . . xld_poly-array ⇝ *object*
Approximating polygons.

▷ **Type** (input_control) . . . . . string ⇝ *string*
Type of approximation.
Default: 'ramer'
List of values: Type ∈ {'ramer'}

▷ **Alpha** (input_control) . . . . . number ⇝ *real* / integer
Threshold for the approximation.
Default: 2.0
Suggested values: Alpha ∈ {1.0, 1.5, 2.0, 3.0, 4.0}
Restriction: Alpha > 0.0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix

Possible Successors

gen_parallels_xld, split_contours_xld

See also

get_region_polygon

Module

Foundation

gen_rectangle2_contour_xld (: Rectangle : Row, Column, Phi, Length1, Length2 :)

Create an XLD contour in the shape of a rectangle.

gen_rectangle2_contour_xld creates one or more XLD contours in the shape of a rectangle with arbitrary orientation. The rectangle has the center (Row, Column), the orientation Phi, and the half edge lengths Length1 and Length2. The angle Phi must be given in radians and specifies the angle between the horizontal axis and the edge with the half length Length1 in the mathematically positive direction (counterclockwise).

Parameter Broadcasting

This operator supports parameter broadcasting. This means that each parameter can be given as a tuple of length *1* or *N*. Parameters with tuple length *1* will be repeated internally such that the number of created items is always *N*.

Parameters

▷ **Rectangle** (output_object) ..... xld_cont(-array) ⇝ *object*
Rectangle contour.

▷ **Row** (input_control) ..... rectangle2.center.y(-array) ⇝ *real*
Row coordinate of the center of the rectangle.
Default: 300.0

▷ **Column** (input_control) ..... rectangle2.center.x(-array) ⇝ *real*
Column coordinate of the center of the rectangle.
Default: 200.0

▷ **Phi** (input_control) ..... rectangle2.angle.rad(-array) ⇝ *real*
Orientation of the main axis of the rectangle [rad].
Default: 0.0
Restriction: - pi / 2 < Phi && Phi <= pi / 2

▷ **Length1** (input_control) ..... rectangle2.hwidth(-array) ⇝ *real*
First radius (half length) of the rectangle.
Default: 100.5

▷ **Length2** (input_control) ..... rectangle2.hheight(-array) ⇝ *real*
Second radius (half width) of the rectangle.
Default: 20.5

Result

gen_rectangle2_contour_xld returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fit_rectangle2_contour_xld

Alternatives

gen_rectangle2

See also

gen_contour_polygon_xld

Module

Foundation

mod_parallels_xld (Parallels, Image : ModParallels, ExtParallels : Quality, MinGray, MaxGray, MaxStandard :)

Extract parallel XLD polygons enclosing a homogeneous area.

mod_parallels_xld selects XLD parallels enclosing homogeneous areas from the input parallels Parallels. The parameter Image contains the corresponding gray value image.

Only parallels having a quality factor larger than Quality are examined. The algorithm performs parallel cross sections one pixel apart, and parallel to the line segments in the area of overlap between two parallel line segments.

In the first iteration, the mean gray value for each of the lines in the cross section is calculated. In the second iteration, the standard deviations of the gray values along each line are computed.

If the mean gray value of an area between parallels lies in the interval [MinGray,MaxGray], and if the mean of all standard deviations is smaller than the upper threshold MaxStandard, the corresponding parallels are returned as modified parallels in ModParallels.

In a second step, all polygon segments adjacent to parallels bordering homogeneous areas are checked for homogeneity. To do so, a rectangle having the width of the last area enclosed by a modified parallel is constructed, and checked for homogeneity using the algorithm described above. This process is continued as long as there are adjacent polygon segments. The polygons thus found are returned as extended parallels in ExtParallels.

Parameters

- ▷ **Parallels** (input_object) ..... xld_para-array ⇝ *object*
  Input XLD parallels.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object : byte*
  Corresponding gray value image.
- ▷ **ModParallels** (output_object) ..... xld_mod_para-array ⇝ *object*
  Modified XLD parallels.
- ▷ **ExtParallels** (output_object) ..... xld_ext_para-array ⇝ *object*
  Extended XLD parallels.
- ▷ **Quality** (input_control) ..... number ⇝ *real / integer*
  Minimum quality factor (measure of parallelism).
  Default: 0.4
  Suggested values: Quality ∈ {0.1, 0.2, 0.3, 0.4, 0.5, 0.6}
  Restriction: 0.0 <= Quality && Quality <= 1.0
- ▷ **MinGray** (input_control) ..... integer ⇝ *integer*
  Minimum mean gray value.
  Default: 160
  Suggested values: MinGray ∈ {80, 100, 120, 140, 160, 180}
  Restriction: 0 <= MinGray && MinGray <= 255
- ▷ **MaxGray** (input_control) ..... integer ⇝ *integer*
  Maximum mean gray value.
  Default: 220
  Suggested values: MaxGray ∈ {140, 160, 180, 200, 220, 240}
  Restriction: 0 <= MaxGray && MaxGray <= 255 && MaxGray >= MinGray

▷ **MaxStandard** (input_control) . . . . . number ⇝ *real / integer*
Maximum allowed standard deviation.
Default: 10.0
Suggested values: MaxStandard ∈ {5.0, 10.0, 15.0, 20.0}
Restriction: MaxStandard >= 0.0

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_parallels_xld

Possible Successors

max_parallels_xld

See also

info_parallels_xld

Module

Foundation

29.3 Features

area_center_points_xld (XLD : : : Area, Row, Column)

Area and center of gravity (centroid) of contours and polygons treated as point clouds.

area_center_points_xld calculates the area and center of gravity (centroid) of the point clouds given by contours or polygons XLD (i.e., the order of the points in the contour or polygon is not taken into account). The area corresponds to the number of points in the point cloud. The centroid is given by the arithmetic mean of all points. If the contour or polygon is closed (end point = start point), the end point of the contour or polygon is not taken into account to avoid that it receives twice the weight of the other points.

area_center_points_xld should be used if the contour XLD intersects itself or if it is not possible to close the contour using a line from end to start point without self-intersection, because in this case area_center_xld does not produce useful results. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used.

If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD.

Attention

Even if the contour or polygon XLD is not intersecting itself, the result of area_center_points_xld significantly differs from the result of area_center_xld as it is calculated from the point cloud and not from the enclosed region.

Parameters

▷ **XLD** (input_object) . . . . . xld(-array) ⇝ *object*
Point clouds to be examined in form of contours or polygons.

▷ **Area** (output_control) . . . . . real(-array) ⇝ *real*
Area of the point cloud.

▷ **Row** (output_control) . . . . . point.y(-array) ⇝ *real*
Row coordinate of the centroid.

▷ **Column** (output_control) . . . . . point.x(-array) ⇝ *real*
Column coordinate of the centroid.

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

area_center_points_xld returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behavior can be set via set_system('no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, smooth_contours_xld, gen_polygons_xld

Alternatives

area_center_xld

See also

moments_points_xld, moments_any_points_xld, area_center, moments_region_2nd

Module

Foundation

area_center_xld (XLD : : : Area, Row, Column, PointOrder)

Area and center of gravity (centroid) of contours and polygons.

area_center_xld calculates the area and center of gravity (centroid) of the regions enclosed by the contours or polygons XLD as well as the order of the points along the boundary. The area and centroid are computed by applying Green's theorem using only the points on the contour or polygon, i.e., no region is generated explicitly for the purpose of calculating the features. If the points are arranged counterclockwise (i.e., in a positive mathematical sense) in the contour or polygon, PointOrder will be *'positive'*. It is assumed that the contours or polygons are closed. If this is not the case area_center_xld will artificially close the contour respectively polygon.

It should be noted that area_center_xld only returns useful results if the contour or polygon encloses a region in the plane. In particular, the contour or polygon must not intersect itself. This is particularly important if open contours or polygons are passed because they are closed automatically, which can produce a self-intersection. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used. If the contour or polygon intersects itself, a useful value for the center of gravity can be calculated with area_center_points_xld.

If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Area** (output_control) ..... real(-array) ⇝ *real*
Area enclosed by the contour or polygon.

▷ **Row** (output_control) ..... point.y(-array) ⇝ *real*
Row coordinate of the centroid.

▷ **Column** (output_control) ..... point.x(-array) ⇝ *real*
Column coordinate of the centroid.

▷ **PointOrder** (output_control) ..... string(-array) ⇝ *string*
point order along the boundary ('positive'/'negative').

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

area_center_xld returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behavior can be set via set_system('no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, smooth_contours_xld, gen_polygons_xld

Alternatives

area_center_points_xld

See also

moments_xld, moments_any_xld, area_center, moments_region_2nd

Module

Foundation

circularity_xld (XLD : : : Circularity)

Shape factor for the circularity (similarity to a circle) of contours or polygons.

The operator circularity_xld calculates the similarity of each input contour or polygon with a circle. The input contour or polygon must not intersect itself, otherwise the resulting parameter is not meaningful (Whether the input contour or polygon intersects itself or not can be determined with test_self_intersection_xld). If the input contour or polygon is not closed it will be closed automatically.

Calculation: If F is the enclosed area of the contour or polygon and max is the maximum distance from the center to all contour or polygon pixels, the shape factor Circularity is defined as:

$$\text{Circularity} = \frac{F}{(max^2 * \pi)}$$

The shape factor Circularity of a circle is 1. If the contour or polygon encloses an elongated area Circularity is smaller than 1. The operator circularity_xld especially responds to large bulges.

If more than one contour or polygon is passed, the numerical values of the shape factor are stored in a tuple in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Circularity** (output_control) ..... real(-array) ⇝ *real*
Roundness of the input contours or polygons.
Assertion: 0 <= Circularity && Circularity <= 1.0

Result

The operator circularity_xld returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contour available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld, test_self_intersection_xld

Alternatives

compactness_xld, convexity_xld, eccentricity_xld, rectangularity_xld

See also

area_center_xld, select_shape_xld

Module

Foundation

compactness_xld (XLD : : : Compactness)

Shape factor for the compactness of contours or polygons.

The operator compactness_xld calculates the compactness of each input contour or polygon in XLD. The input contour or polygon must not intersect itself, otherwise the resulting parameter is not meaningful (Whether the input contour or polygon intersects itself or not can be determined with test_self_intersection_xld). If the input contour or polygon is not closed it will be closed automatically.

Calculation: If L is the length and F the enclosed area of the contour or polygon the shape factor Compactness is defined as:

$$\text{Compactness} = \frac{L^2}{4F\pi}$$

The shape factor Compactness of a circle is 1. If the contour or polygon encloses an elongated area Compactness is larger than 1. The operator compactness_xld responds to the course of the contour or polygon (roughness). If more than one contour or polygon is passed, the shape factors are stored in a tuple in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Compactness** (output_control) ..... real(-array) ⇝ *real*
Compactness of the input contours or polygons.
Assertion: Compactness >= 1.0 || Compactness == 0

Result

The operator compactness_xld returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contours available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld, test_self_intersection_xld

Alternatives

circularity_xld, convexity_xld, eccentricity_xld, rectangularity_xld

See also

area_center_xld, select_shape_xld

Module

Foundation

contour_point_num_xld (Contour : : : Length)

Return the number of points in an XLD contour.

contour_point_num_xld returns the length of the input contour Contour, i.e., its number of points, in Length.

Parameters

▷ **Contour** (input_object) ..... xld_cont(-array) ⇝ *object*
Input XLD contour.

▷ **Length** (output_control) ..... integer(-array) ⇝ *integer*
Number of contour points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix

Possible Successors

get_contour_xld, get_contour_attrib_xld

Alternatives

get_regress_params_xld

See also

query_contour_attribs_xld

Module

Foundation

convexity_xld (XLD : : : Convexity)

Shape factor for the convexity of contours or polygons.

The operator convexity_xld calculates the convexity for each input contour or polygon. The input contour or polygon must not intersect itself, otherwise the resulting parameter is not meaningful (Whether the input contour or polygon intersects itself or not can be determined with test_self_intersection_xld). If the input contour or polygon is not closed it will be closed automatically.

Calculation: If F_c is the area of the convex hull and F_o the area enclosed by the original contour or polygon the shape factor Convexity is defined as:

$$\texttt{Convexity} = \frac{F_o}{F_c}$$

The shape factor Convexity is 1 if the contour or polygon is convex (e.g., a rectangle, circle, etc.). If there are indentations Convexity is smaller than 1.

If more than one contour or polygon is passed, the numerical results of the shape factor are stored in a tuple in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Convexity** (output_control) ..... real(-array) ⇝ *real*
Convexity of the input contours or polygons.
Assertion: Convexity <= 1

Result

The operator convexity_xld returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contour available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld, test_self_intersection_xld

See also

area_center_xld, select_shape_xld, shape_trans_xld

Module

Foundation

diameter_xld (XLD : : : Row1, Column1, Row2, Column2, Diameter)

Maximum distance between two contour or polygon points.

The operator diameter_xld calculates the maximum distance between two points of each input contour or polygon. The coordinates of these two extremes and the distance between them will be returned. The input contour or polygon must not intersect itself, otherwise the resulting parameters are not meaningful (Whether the input contour or polygon intersects itself or not can be determined with test_self_intersection_xld). If the input contour or polygon is not closed it will be closed automatically. If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD.

Attention

If the contour or polygon is empty, the results of Row1, Column1, Row2 and Column2 (all of them = 0) may lead to confusion.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Row1** (output_control) ..... line.begin.y(-array) ⇝ *real*
Row coordinate of the first extreme point of the contours or polygons.

▷ **Column1** (output_control) ..... line.begin.x(-array) ⇝ *real*
Column coordinate of the first extreme point of the contours or polygons.

▷ **Row2** (output_control) ..... line.end.y(-array) ⇝ *real*
Row coordinate of the second extreme point of the contour or polygons.

▷ **Column2** (output_control) ..... line.end.x(-array) ⇝ *real*
Column coordinate of the second extreme point of the contours or polygons.

▷ **Diameter** (output_control) ..... number(-array) ⇝ *real*
Distance of the two extreme points of the contours or polygons.

Result

The operator diameter_xld returns the value 2 (H_MSG_TRUE), if the input is not empty. The reaction to empty input (no input contours are available) may be determined with the help of the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld, test_self_intersection_xld

Alternatives

smallest_rectangle2_xld

See also

area_center_xld

Module

Foundation

dist_ellipse_contour_points_xld (Contour : : DistanceMode, ClippingEndPoints, Row, Column, Phi, Radius1, Radius2 : Distances)

Compute the distances of all contour points to an ellipse.

The operator dist_ellipse_contour_points_xld determines the distances between points of a contour Contour and an ellipse specified by the center (Row, Column), the orientation of the main axis Phi, the length of the larger half axis Radius1, and the length of the smaller half axis Radius2.

The distance measure is the geometric distance, sometimes also called orthogonal distance. The distances Distances are returned for all points of the contour. Normally, the distance is a positive value. In this case DistanceMode equal to *'unsigned'* must be chosen. If this option is switched to *'signed'* the distances can take positive or negative sign, depending on whether the contour points lie outside or inside the ellipse respectively.

Because of artifacts in the preprocessing, the start and end points of a contour might be faulty. In this case, the operator fit_ellipse_contour_xld is typically called with the parameter ClippingEndPoints set to a value greater than zero to exclude points at the beginning and the end of the contour from the computation. In order to get the geometric distances of the same set of points as used in the fitting process ClippingEndPoints should take the same value.

Parameters

▷ **Contour** (input_object) . . . . . xld_cont ⇝ *object*
Input contours.

▷ **DistanceMode** (input_control) . . . . . string ⇝ *string*
Mode for unsigned or signed distance values.
Default: 'unsigned'
List of values: DistanceMode ∈ {'unsigned', 'signed'}

▷ **ClippingEndPoints** (input_control) . . . . . integer ⇝ *integer*
Number of points at the beginning and the end of the contours to be ignored for the computation of distances.
Default: 0
Restriction: ClippingEndPoints >= 0

▷ **Row** (input_control) . . . . . ellipse.center.y ⇝ *real*
Row coordinate of the center of the ellipse.

▷ **Column** (input_control) . . . . . ellipse.center.x ⇝ *real*
Column coordinate of the center of the ellipse.

▷ **Phi** (input_control) . . . . . ellipse.angle.rad ⇝ *real*
Orientation of the main axis in radian.
Restriction: Phi >= 0 && Phi <= 6.283185307

▷ **Radius1** (input_control) . . . . . ellipse.radius1 ⇝ *real*
Length of the larger half axis.
Restriction: Radius1 > 0

▷ **Radius2** (input_control) . . . . . ellipse.radius2 ⇝ *real*
Length of the smaller half axis.
Restriction: Radius2 >= 0 && Radius2 <= Radius1

▷ **Distances** (output_control) . . . . . real-array ⇝ *real*
Distances of the contour points to the ellipse.

Result

`dist_ellipse_contour_points_xld` returns `2` (`H_MSG_TRUE`) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`fit_ellipse_contour_xld`

Alternatives

`dist_ellipse_contour_xld`

Module

Foundation

dist_ellipse_contour_xld (Contours : : Mode, MaxNumPoints, ClippingEndPoints, Row, Column, Phi, Radius1, Radius2 : MinDist, MaxDist, AvgDist, SigmaDist)

Compute the distance of contours to an ellipse.

`dist_ellipse_contour_xld` determines the distance between the contours in `Contours` and an ellipse specified by the center (`Row`, `Column`), the orientation of the main axis `Phi`, the length of the larger half axis `Radius1`, and the length of the smaller half axis `Radius2`. Different measures for the distance of a contour point $X = (x_i, y_i)$ to the ellipse are available (`Mode`):

'geometric' The underlying distance measure is the geometric distance between a point on the contour and the ellipse. This measure is also called orthogonal or Euclidean distance.

'algebraic' The distance is measured by the algebraic distance
$ax_i^2 + bx_iy_i + cy_i^2 + dx_i + ey_i + f$, where the parameters a - f describing the ellipse are normalized in order to obtain `Radius2` as distance of the center of the ellipse. This measure shows a *high curvature bias*: Near points of high curvature on the ellipse (like the poles on the main axis) the distance is smaller than near points with low curvature.

'focpoints' The distance is measured by the deviation $XF_1 + XF_2 - 2a$, where F_1,F_2 are the focal points and a corresponds to `Radius1`. This measure shows a *low curvature bias*: Near points of high curvature on the ellipse (like the poles on the main axis) the distance is larger than near points with low curvature.

'bisec' The distance is measured by the distance between X and the intersection of the angular bisector of the two lines through X and the focal points with the ellipse. This is a good approximation of the orthogonal distance from X to the ellipse. The accuracy of the approximation depends on both the aspect ratio of the ellipse and the distance.

The operator returns the minimum absolute distance `MinDist`, the maximum absolute distance `MaxDist`, the average absolute distance `AvgDist`, and the standard deviation of the absolute distances `SigmaDist` of all contour points.

To reduce the computational load, the computation of the distances can be restricted to a subset of the contour points: If a value other than -1 is assigned to `MaxNumPoints`, only up to `MaxNumPoints` points - uniformly distributed over the contour - are used. Due to artifacts in the pre-processing the start and end points of a contour might be faulty. Therefore, it is possible to exclude `ClippingEndPoints` points at the beginning and at the end of a contour from the computation.

Parameters

- ▷ **Contours** (input_object) . . . . . xld_cont(-array) ⇝ *object*
  Input contours.
- ▷ **Mode** (input_control) . . . . . string ⇝ *string*
  Method for the determination of the distances.
  Default: 'geometric'
  List of values: Mode ∈ {'geometric', 'algebraic', 'focpoints', 'bisec'}
- ▷ **MaxNumPoints** (input_control) . . . . . integer ⇝ *integer*
  Maximum number of contour points used for the computation (-1 for all points).
  Default: -1
  Restriction: MaxNumPoints >= 3
- ▷ **ClippingEndPoints** (input_control) . . . . . integer ⇝ *integer*
  Number of points at the beginning and the end of the contours to be ignored for the computation of distances.
  Default: 0
  Restriction: ClippingEndPoints >= 0
- ▷ **Row** (input_control) . . . . . ellipse.center.y ⇝ *real*
  Row coordinate of the center of the ellipse.
- ▷ **Column** (input_control) . . . . . ellipse.center.x ⇝ *real*
  Column coordinate of the center of the ellipse.
- ▷ **Phi** (input_control) . . . . . ellipse.angle.rad ⇝ *real*
  Orientation of the main axis in radian.
  Restriction: Phi >= 0 && Phi <= 6.283185307
- ▷ **Radius1** (input_control) . . . . . ellipse.radius1 ⇝ *real*
  Length of the larger half axis.
  Restriction: Radius1 > 0
- ▷ **Radius2** (input_control) . . . . . ellipse.radius2 ⇝ *real*
  Length of the smaller half axis.
  Restriction: Radius2 >= 0 && Radius2 <= Radius1
- ▷ **MinDist** (output_control) . . . . . real(-array) ⇝ *real*
  Minimum distance.
- ▷ **MaxDist** (output_control) . . . . . real(-array) ⇝ *real*
  Maximum distance.
- ▷ **AvgDist** (output_control) . . . . . real(-array) ⇝ *real*
  Mean distance.
- ▷ **SigmaDist** (output_control) . . . . . real(-array) ⇝ *real*
  Standard deviation of the distance.

Example

```
read_image (Image, 'caltab')
find_caltab (Image, CalPlate, 'caltab_big.descr', 3, 112, 5)
reduce_domain (Image, CalPlate, ImageReduced)
edges_sub_pix (ImageReduced, Edges, 'lanser2', 0.5, 20, 40)
select_contours_xld (Edges, EdgesClosed, 'closed', 0, 2.0, 0, 0)
select_contours_xld (EdgesClosed, EdgesMarks, 'contour_length', 20, 100, \
                     0, 0)
fit_ellipse_contour_xld (EdgesMarks, 'fitzgibbon', -1, 2, 0, 200, 3, 2.0, \
                         Row, Column, Phi, Radius1, Radius2, StartPhi, \
                         EndPhi, PointOrder)
for i := 0 to |Row|-1 by 1
    select_obj (EdgesMarks, ObjectSelected, i+1)
    dist_ellipse_contour_xld (ObjectSelected, 'bisec', -1, 0, Row[i], \
                              Column[i], Phi[i], Radius1[i], Radius2[i], \
                              MinDist, MaxDist, AvgDist, SigmaDist)
endfor
```

Result

dist_ellipse_contour_xld returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fit_ellipse_contour_xld

Alternatives

dist_ellipse_contour_points_xld

Module

Foundation

dist_rectangle2_contour_points_xld (
Contour : : ClippingEndPoints, Row, Column, Phi, Length1,
Length2 : Distances)

Compute the distances of all contour points to a rectangle.

dist_rectangle2_contour_points_xld determines the Euclidean distances between the points of a contour Contour and a rectangle of arbitrary orientation specified by the center (Row, Column), the orientation Phi, and the half edge lengths Length1 and Length2. The angle Phi must be given in radians and specifies the angle between the horizontal axis and the edge with the half length Length1 in the mathematically positive direction (counterclockwise).

Depending on the processing used to create Contour, the start and end points of a contour may contain positional errors. In this case, the operator fit_rectangle2_contour_xld is typically called with the parameter ClippingEndPoints set to a value greater than zero to exclude points at the beginning and the end of the contour from the computation. To compute the distances of the same set of points as used in the fitting process, ClippingEndPoints should be set to the same value as in fit_rectangle2_contour_xld.

Parameters

▷ **Contour** (input_object) . . . . . xld_cont ⇝ *object*
Input contour.

▷ **ClippingEndPoints** (input_control) . . . . . integer ⇝ *integer*
Number of points at the beginning and the end of the contours to be ignored for the computation of distances.
Default: 0
Suggested values: ClippingEndPoints ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Restriction: ClippingEndPoints >= 0

▷ **Row** (input_control) . . . . . rectangle2.center.y ⇝ *real*
Row coordinate of the center of the rectangle.

▷ **Column** (input_control) . . . . . rectangle2.center.x ⇝ *real*
Column coordinate of the center of the rectangle.

▷ **Phi** (input_control) . . . . . rectangle2.angle.rad ⇝ *real*
Orientation of the main axis of the rectangle [rad].

▷ **Length1** (input_control) . . . . . rectangle2.hwidth ⇝ *real*
First radius (half length) of the rectangle.
Restriction: Length1 > 0

▷ **Length2** (input_control) . . . . . rectangle2.hheight ⇝ *real*
Second radius (half width) of the rectangle.
Restriction: Length2 >= 0

▷ **Distances** (output_control) . . . . . real-array ⇝ *real*
Distances of the contour points to the rectangle.

Result

dist_rectangle2_contour_points_xld returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

fit_rectangle2_contour_xld

Module

Foundation

eccentricity_points_xld (XLD : : : Anisometry)

Anisometry of contours or polygons treated as point clouds.

The operator eccentricity_points_xld calculates the shape feature Anisometry derived from the geometric moments of the point cloud given by the contour or polygon XLD (i.e., the order of the points in the contour or polygon is not taken into account). If the contour or polygon is closed (end point = start point), the end point of the contour or polygon is not taken into account to avoid that it receives twice the weight of the other points.

Calculation: If the ellipse radii Ra, Rb (elliptic_axis_points_xld) are given, the following applies:

$$\texttt{Anisometry} = \frac{Ra}{Rb}$$

The anisometry of a circle is 1.0.

eccentricity_points_xld should be used if the contour XLD intersects itself or if it is not possible to close the contour using a line from end to start point without self-intersection, because in this case eccentricity_xld does not produce useful results. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used.

If more than one contour or polygon is passed, the values of the anisometry are stored in a tuple in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Anisometry** (output_control) ..... real(-array) ⇝ *real*
Anisometry of the contours or polygons.
Assertion: Anisometry >= 1.0

Result

The operator eccentricity_points_xld returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contours available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld, test_self_intersection_xld

Alternatives

eccentricity_xld

See also

elliptic_axis_points_xld, moments_region_2nd, select_shape_xld

Module

Foundation

eccentricity_xld (XLD : : : Anisometry, Bulkiness, StructureFactor)

Shape features derived from the ellipse parameters of contours or polygons.

The operator eccentricity_xld calculates the three shape features Anisometry, Bulkiness, and StructureFactor derived from the geometric moments for each input contour or polygon. If the input contour or polygon is not closed it will be closed automatically.

Calculation: If the ellipse radii Ra, Rb (elliptic_axis_xld) and the enclosed area A of the contour or polygon are given, the following applies:

$$\begin{aligned} \texttt{Anisometry} &= \frac{Ra}{Rb} \\ \texttt{Bulkiness} &= \frac{\pi \cdot Ra \cdot Rb}{F} \\ \texttt{StructureFactor} &= \texttt{Anisometry} \cdot \texttt{Bulkiness} - 1 \end{aligned}$$

The anisometry of a circle is 1.0.

It should be noted that eccentricity_xld only returns useful results if the contour or polygon encloses a region in the plane. In particular, the contour or polygon must not intersect itself. This is particularly important if open contours or polygons are passed because they are closed automatically, which can produce a self-intersection. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used. If the contour or polygon intersects itself, a useful value for the anisometry can be calculated with eccentricity_points_xld.

If more than one contour is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a contour in the input.

Parameters

- ▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
  Contours or polygons to be examined.
- ▷ **Anisometry** (output_control) ..... real(-array) ⇝ *real*
  Anisometry of the contours or polygons.
  Assertion: Anisometry >= 1.0
- ▷ **Bulkiness** (output_control) ..... real(-array) ⇝ *real*
  Bulkiness of the contours or polygons.
- ▷ **StructureFactor** (output_control) ..... real(-array) ⇝ *real*
  Structure factor of the contours or polygons.

Result

The operator eccentricity_xld returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contours available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld, test_self_intersection_xld

Alternatives

eccentricity_points_xld

See also

elliptic_axis_xld, moments_region_2nd, select_shape_xld, area_center_xld

Module

Foundation

elliptic_axis_points_xld (XLD : : : Ra, Rb, Phi)

Parameters of the equivalent ellipse of contours or polygons treated as point clouds.

The operator elliptic_axis_points_xld calculates the radii (Ra, Rb) and the orientation (Phi, in radians) of the ellipse having the same orientation and the same aspect ratio as the point cloud given by the contour or polygon XLD (i.e., the order of the points in the contour or polygon is not taken into account). If the contour or polygon is closed (end point = start point), the end point of the contour or polygon is not taken into account to avoid that it receives twice the weight of the other points.

Calculation: If the moments M_{20}, M_{02} and M_{11} are normalized to the area (see moments_points_xld), the major radius Ra and the minor radius Rb are calculated as:

$$\mathtt{Ra} = \frac{\sqrt{8(M_{20} + M_{02} + \sqrt{(M_{20} - M_{02})^2 + 4M_{11}^2})}}{2}$$

$$\mathtt{Rb} = \frac{\sqrt{8(M_{20} + M_{02} - \sqrt{(M_{20} - M_{02})^2 + 4M_{11}^2})}}{2}$$

The orientation Phi, i.e., the angle between the major axis and the x (column) axis, is defined by:

$$\mathtt{Phi} = -0.5 \operatorname{atan2}(2M_{11}, M_{02} - M_{20})$$

elliptic_axis_points_xld should be used if the contour XLD intersects itself or if it is not possible to close the contour using a line from end to start point without self-intersection, because in this case elliptic_axis_xld does not produce useful results. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used.

If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD.

Parameters

- ▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
  Contours or polygons to be examined.
- ▷ **Ra** (output_control) ..... real(-array) ⇝ *real*
  Major radius.
  Assertion: Ra >= 0.0
- ▷ **Rb** (output_control) ..... real(-array) ⇝ *real*
  Minor radius.
  Assertion: Rb >= 0.0 && Rb <= Ra
- ▷ **Phi** (output_control) ..... angle.rad(-array) ⇝ *real*
  Angle between the major axis and the column axis (radians).
  Assertion: - pi / 2 < Phi && Phi <= pi / 2

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

elliptic_axis_points_xld returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld, test_self_intersection_xld

Possible Successors

area_center_points_xld, gen_ellipse_contour_xld

Alternatives

elliptic_axis_xld, smallest_rectangle2

See also

moments_xld, smallest_circle_xld, smallest_rectangle1_xld, smallest_rectangle2_xld, shape_trans_xld

References

R. Haralick, L. Shapiro "Computer and Robot Vision" Addison-Wesley, 1992, pp. 73-75

Module

Foundation

elliptic_axis_xld (XLD : : : Ra, Rb, Phi)

Parameters of the equivalent ellipse of contours or polygons.

The operator elliptic_axis_xld calculates the radii and the orientations of the ellipses having the same orientation and the same aspect ratio as the input contours or polygons. The length of the major radius Ra and the minor radius Rb as well as the orientation of the main axis with regard to the horizontal (Phi) are determined. The angle is indicated in radians. It is assumed that the contours or polygons are closed. If this is not the case elliptic_axis_xld will artificially close the contours or polygons.

Calculation: If the moments M_{20}, M_{02} and M_{11} are normalized and passed to the area (see moments_xld), the radii Ra and Rb are calculated as:

$$\texttt{Ra} = \frac{\sqrt{8(M_{20}+M_{02}+\sqrt{(M_{20}-M_{02})^2+4M_{11}^2})}}{2}$$

$$\texttt{Rb} = \frac{\sqrt{8(M_{20}+M_{02}-\sqrt{(M_{20}-M_{02})^2+4M_{11}^2})}}{2}$$

The orientation Phi is defined by:

$$\texttt{Phi} = -0.5\text{atan2}(2M_{11}, M_{02}-M_{20})$$

It should be noted that elliptic_axis_xld only returns useful results if the contour or polygon encloses a region in the plane. In particular, the contour or polygon must not intersect itself. This is particularly important if open contours or polygons are passed because they are closed automatically, which can produce a self-intersection. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used. If the contour or polygon intersects itself, useful values for the ellipse parameters can be calculated with elliptic_axis_points_xld.

If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Ra** (output_control) ..... real(-array) ⇝ *real*
Major radius.
Assertion: Ra >= 0.0

▷ **Rb** (output_control) ..... real(-array) ⇝ *real*
Minor radius.
Assertion: Rb >= 0.0 && Rb <= Ra

▷ **Phi** (output_control) ..... angle.rad(-array) ⇝ *real*
Angle between the major axis and the x axis (radians).
Assertion: - pi / 2 < Phi && Phi <= pi / 2

Complexity

If N is the number of contour or polygon points, the runtime complexity is $O(N)$.

Result

`elliptic_axis_xld` returns `2` (`H_MSG_TRUE`) if the input is not empty. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`gen_contours_skeleton_xld`, `edges_sub_pix`, `threshold_sub_pix`, `gen_contour_polygon_xld`, `test_self_intersection_xld`

Possible Successors

`area_center_xld`, `gen_ellipse_contour_xld`

Alternatives

`elliptic_axis_points_xld`, `smallest_rectangle2`

See also

`moments_xld`, `smallest_circle_xld`, `smallest_rectangle1_xld`, `smallest_rectangle2_xld`, `shape_trans_xld`

References

R. Haralick, L. Shapiro “Computer and Robot Vision” Addison-Wesley, 1992, pp. 73-75

Module

Foundation

fit_circle_contour_xld (Contours : : Algorithm, MaxNumPoints, MaxClosureDist, ClippingEndPoints, Iterations, ClippingFactor : Row, Column, Radius, StartPhi, EndPhi, PointOrder)

Approximate XLD contours by circles.

`fit_circle_contour_xld` approximates the XLD contours `Contours` by circles. It does not perform a segmentation of the input contours. Thus, one has to make sure that each contour corresponds to one and only one circle. The operator returns for each contour the center (`Row`, `Column`), and the `Radius`.

The algorithm used for the fitting of circles can be selected via `Algorithm`:

'algebraic' This approach minimizes the algebraic distance between the contour points and the resulting circle.

'ahuber' Similar to *'algebraic'*. Here the contour points are weighted to decrease the impact of outliers based on the approach of Huber (see below).

'atukey' Similar to *'algebraic'*. Here the contour points are weighted and outliers are ignored based on the approach of Tukey (see below).

'geometric' This approach minimizes the geometric distance between the contour points and the resulting circle. The distance measure is statistically optimal, but takes more computational time. This option is recommended if the contour points are considerably distorted by noise.

'geohuber' Similar to *'geometric'*. Here the contour points are weighted to decrease the impact of outliers based on the approach of Huber (see below).

'geotukey' Similar to *'geometric'*. Here the contour points are weighted and outliers are ignored based on the approach of Tukey (see below).

For '*huber' and '*tukey' a robust error statistics is used to estimate the standard deviation of the distances from the contour points *without* outliers from the approximating circle. The parameter `ClippingFactor` (a scaling factor for the standard deviation) controls the amount of outliers: the smaller the value chosen for `ClippingFactor` the more outliers are detected. In the Tukey algorithm outliers are removed, whereas in the Huber algorithm outliers are only damped, or more precisely they are weighted linearly. Without any robust weighting the squares of the distances are taken as error values in the optimization, i.e., a least square formulation. In practice, the approach of Tukey is recommended.

The parameter `Iterations` specifies the number of iterations for the algorithms *'algebraic'*, *'ahuber'* and *'atukey'*. This parameter is ignored for the algorithms *'geometric'*, *'geohuber'* and *'geotukey'*. If `Iterations` is set to zero, the algorithm does not perform iterative improvements on the fitted circle, but only checks if the initial guess was already close enough depending on the chosen treatment of outliers.

To reduce the computational load, the fitting of circles can be restricted to a subset of the contour points: If a value other than *-1* is assigned to `MaxNumPoints`, only up to `MaxNumPoints` points - uniformly distributed over the contour - are used.

For circular arcs, the points on the circle closest to the start points and end points of the original contours are chosen as start and end points. The corresponding angles referring to the X-axis are returned in `StartPhi` and `EndPhi`, see also `gen_ellipse_contour_xld`. Contours, for which the distance between their start points and their end points is less or equal `MaxClosureDist` are considered to be closed. Thus, they are approximated by circles instead of circular arcs. Due to artifacts in the pre-processing the start and end points of a contour might be faulty. Therefore, it is possible to exclude `ClippingEndPoints` points at the beginning and at the end of a contour from the fitting of circles. However, they are still used for the determination of `StartPhi` and `EndPhi`.

The minimum necessary number of contour points for fitting a circle is three. Therefore, it is required that the number of contour points is at least $3 + 2 \cdot$ `ClippingEndPoints`.

Parameters

▷ **`Contours`** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contours.

▷ **`Algorithm`** (input_control) ..... string ⇝ *string*
Algorithm for the fitting of circles.
Default: 'algebraic'
List of values: `Algorithm` ∈ {'algebraic', 'ahuber', 'atukey', 'geometric', 'geohuber', 'geotukey'}

▷ **`MaxNumPoints`** (input_control) ..... integer ⇝ *integer*
Maximum number of contour points used for the computation (-1 for all points).
Default: -1
Restriction: MaxNumPoints >= 3

▷ **`MaxClosureDist`** (input_control) ..... real ⇝ *real*
Maximum distance between the end points of a contour to be considered as 'closed'.
Default: 0.0
Restriction: MaxClosureDist >= 0.0

▷ **`ClippingEndPoints`** (input_control) ..... integer ⇝ *integer*
Number of points at the beginning and at the end of the contours to be ignored for the fitting.
Default: 0
Restriction: ClippingEndPoints >= 0

▷ **`Iterations`** (input_control) ..... integer ⇝ *integer*
Maximum number of iterations for the robust weighted fitting.
Default: 3
Restriction: Iterations >= 0

▷ **ClippingFactor** (input_control) ..... real ⇝ *real*
Clipping factor for the elimination of outliers (typical: 1.0 for Huber and 2.0 for Tukey).
Default: 2.0
Suggested values: ClippingFactor ∈ {1.0, 1.5, 2.0, 2.5, 3.0}
Restriction: ClippingFactor > 0

▷ **Row** (output_control) ..... circle.center.y(-array) ⇝ *real*
Row coordinate of the center of the circle.

▷ **Column** (output_control) ..... circle.center.x(-array) ⇝ *real*
Column coordinate of the center of the circle.

▷ **Radius** (output_control) ..... circle.radius(-array) ⇝ *real*
Radius of circle.

▷ **StartPhi** (output_control) ..... angle.rad(-array) ⇝ *real*
Angle of the start point [rad].

▷ **EndPhi** (output_control) ..... angle.rad(-array) ⇝ *real*
Angle of the end point [rad].

▷ **PointOrder** (output_control) ..... string(-array) ⇝ *string*
Point order along the boundary.
List of values: PointOrder ∈ {'positive', 'negative'}

Result

fit_circle_contour_xld returns 2 (H_MSG_TRUE) if all parameter values are correct, and circles could be fitted to the input contours. If the input is empty the behavior can be set via set_system ('no_object_result',<Result>). If necessary, an exception is raised. If the parameter ClippingFactor is chosen too small, i.e., all points are classified as outliers, the error 3264 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix, smooth_contours_xld

Possible Successors

gen_ellipse_contour_xld, disp_circle, get_points_ellipse

See also

fit_ellipse_contour_xld, fit_line_contour_xld, fit_rectangle2_contour_xld

Module

Foundation

fit_ellipse_contour_xld (Contours : : Algorithm, MaxNumPoints, MaxClosureDist, ClippingEndPoints, VossTabSize, Iterations, ClippingFactor : Row, Column, Phi, Radius1, Radius2, StartPhi, EndPhi, PointOrder)

Approximate XLD contours by ellipses or elliptic arcs.

fit_ellipse_contour_xld approximates the XLD contours Contours by elliptic arcs or closed ellipses. It does not perform a segmentation of the input contours. Thus, one has to make sure that each contour corresponds to one and only one elliptic structure. The operator returns for each contour the center (Row, Column), the orientation of the main axis Phi, the length of the larger half axis Radius1, and the length of the smaller half axis Radius2 of the underlying ellipse. In addition to that, the angle corresponding to the start point and the end point StartPhi, EndPhi, and the point order along the boundary PointOrder is returned for elliptic arcs. These parameters are set to 0, 2π, and 'positive' for closed ellipses. The algorithm used for the fitting of ellipses can be selected via Algorithm:

'fitzgibbon' This approach minimizes the algebraic distance $ax_i^2+bx_iy_i+cy_i^2+dx_i+ey_i+f$ between the contour points (x_i, y_i) and the resulting ellipse. The constraint $4ac-b^2=1$ guarantees that the resulting polynomial characterizes an ellipse (instead of a hyperbola or a parabola).

'fhuber' Similar to *'fitzgibbon'*. Here the contour points are weighted to decrease the impact of outliers based on the approach of Huber (see below).

'ftukey' Similar to *'fitzgibbon'*. Here the contour points are weighted and outliers are ignored based on the approach of Tukey (see below).

'geometric' This approach minimizes the geometric distance between the contour points and the resulting ellipse. The distance measure is statistically optimal, but takes more computational time, because the computation is an iterative process. This option is recommended if the contour points are considerably distorted by noise.

'geohuber' Similar to *'geometric'*. Here the contour points are weighted to decrease the impact of outliers based on the approach of Huber (see below).

'geotukey' Similar to *'geometric'*. Here the contour points are weighted and outliers are ignored based on the approach of Tukey (see below).

'voss' Each input contour is transformed in an affine standard position. Based on the moments of the transformed contour (that is of the enclosed image region) the standard circular segment is chosen whose standard position matches best with the standard position of the contour. The ellipse corresponding to the standard position of the selected circular segment is re-transformed based on the affine transformation which produced the standard position of the contour resulting in the ellipse matching the original contour. `VossTabSize` standard circular segments are used for this computation. To speed up the process the corresponding moments and other data is stored in a table which is created during the first call (with a specific value for `VossTabSize`) to `fit_ellipse_contour_xld`.

'focpoints' Each point P on an ellipse satisfies the constraint that the sum of distances to the focal points F_1,F_2 equals twice the length of the larger half axis a. In this approach, the deviation PF_1+PF_2-2a is minimized for all contour points by a least squares optimization.

'fphuber' Similar to *'focpoints'*. Here a *weighted* least squares optimization is done to decrease the impact of outliers based on the approach of Huber (see below).

'fptukey' Similar to *'focpoints'*. Here a *weighted* least squares optimization is done and outliers are ignored based on the approach of Tukey (see below).

For *'*huber'* and *'*tukey'* a robust error statistics is used to estimate the standard deviation of the distances from the contour points *without* outliers from the approximating ellipse. The parameter `ClippingFactor` (a scaling factor for the standard deviation) controls the amount of outliers: the smaller the value chosen for `ClippingFactor` the more points are classified as outliers. Every contour point is individually weighted and contributes according to its weight in the fitting process. The bundle of robust weighting and fitting can be iterated. The overall number of iterations is given by `Iterations`. This parameter is of relevance for all algorithms except the two based on the geometric distance, i.e., *'geohuber'* and *'geotukey'*, where this parameter is ignored. If the algorithm does not find a fitting ellipse within this number of iterations, it fits a line; in this case `Radius2` is zero. In the Tukey algorithm outliers are removed, whereas in the Huber algorithm outliers are only damped, or more precisely they are weighted linearly. Without any robust weighting the squares of the distances are taken as error values in the optimization, i.e., a least square formulation. In practice, the approach of Tukey is recommended.

To reduce the computational load, the fitting of ellipses can be restricted to a subset of the contour points: If a value other than *-1* is assigned to `MaxNumPoints`, only up to `MaxNumPoints` points - uniformly distributed over the contour - are used.

For elliptic arcs, the points on the ellipse closest to the start points and end points of the original contours are chosen as start and end points. The corresponding angles referring to the main axis of the ellipse are returned in `StartPhi` and `EndPhi`, see also `gen_ellipse_contour_xld`. Contours, for which the distance between their start points and their end points is less or equal `MaxClosureDist` are considered to be closed. Thus, they are approximated by ellipses instead of elliptic arcs. The *'focpoints'* algorithms determine the initial parameters for the optimization for closed contours in a faster and less accurate way than for open contours. As a result, in some cases more `Iterations` may be necessary when applied to closed contours. Due to artifacts in the pre-processing the start and end points of a contour might be faulty. Therefore, it is possible to exclude `ClippingEndPoints` points at the beginning and at the end of a contour from the fitting of ellipses. However, they are still used for the determination of `StartPhi` and `EndPhi`.

The minimum necessary number of contour points for fitting an ellipse is five. Therefore, it is required that the number of contour points is at least $5 + 2 \cdot$ `ClippingEndPoints`.

Parameters

▷ **`Contours`** (input_object) . . . . . xld_cont(-array) ⇝ *object*
Input contours.

▷ **`Algorithm`** (input_control) . . . . . string ⇝ *string*
Algorithm for the fitting of ellipses.
Default: 'fitzgibbon'
List of values: `Algorithm` ∈ {'fitzgibbon', 'fhuber', 'ftukey', 'geometric', 'geohuber', 'geotukey', 'voss', 'focpoints', 'fphuber', 'fptukey'}

▷ **`MaxNumPoints`** (input_control) . . . . . integer ⇝ *integer*
Maximum number of contour points used for the computation (-1 for all points).
Default: -1
Restriction: MaxNumPoints >= 5

▷ **`MaxClosureDist`** (input_control) . . . . . real ⇝ *real*
Maximum distance between the end points of a contour to be considered as 'closed'.
Default: 0.0
Restriction: MaxClosureDist >= 0.0

▷ **`ClippingEndPoints`** (input_control) . . . . . integer ⇝ *integer*
Number of points at the beginning and at the end of the contours to be ignored for the fitting.
Default: 0
Restriction: ClippingEndPoints >= 0

▷ **`VossTabSize`** (input_control) . . . . . integer ⇝ *integer*
Number of circular segments used for the Voss approach.
Default: 200
Restriction: VossTabSize >= 25 && VossTabSize <= 5000

▷ **`Iterations`** (input_control) . . . . . integer ⇝ *integer*
Maximum number of iterations for the robust weighted fitting.
Default: 3
Restriction: Iterations >= 0

▷ **`ClippingFactor`** (input_control) . . . . . real ⇝ *real*
Clipping factor for the elimination of outliers (typical: 1.0 for '*huber' and 2.0 for '*tukey').
Default: 2.0
Suggested values: `ClippingFactor` ∈ {1.0, 1.5, 2.0, 2.5, 3.0}
Restriction: ClippingFactor > 0

▷ **`Row`** (output_control) . . . . . ellipse.center.y(-array) ⇝ *real*
Row coordinate of the center of the ellipse.

▷ **`Column`** (output_control) . . . . . ellipse.center.x(-array) ⇝ *real*
Column coordinate of the center of the ellipse.

▷ **`Phi`** (output_control) . . . . . ellipse.angle.rad(-array) ⇝ *real*
Orientation of the main axis [rad].

▷ **`Radius1`** (output_control) . . . . . ellipse.radius1(-array) ⇝ *real*
Length of the larger half axis.

▷ **`Radius2`** (output_control) . . . . . ellipse.radius2(-array) ⇝ *real*
Length of the smaller half axis.

▷ **`StartPhi`** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Angle of the start point [rad].

▷ **`EndPhi`** (output_control) . . . . . angle.rad(-array) ⇝ *real*
Angle of the end point [rad].

▷ **`PointOrder`** (output_control) . . . . . string(-array) ⇝ *string*
point order along the boundary.
List of values: `PointOrder` ∈ {'positive', 'negative'}

Example

```
read_image (Image, 'caltab')
```

```
find_caltab (Image, CalPlate, 'caltab_800mm.descr', 3, 112, 5)
reduce_domain (Image, CalPlate, ImageReduced)
edges_sub_pix (ImageReduced, Edges, 'lanser2', 0.5, 20, 40)
select_contours_xld (Edges, EdgesClosed, 'closed', 0, 2.0, 0, 0)
select_contours_xld (EdgesClosed, EdgesMarks, 'length', 20, 80, 0, 0)
fit_ellipse_contour_xld (EdgesMarks, 'fitzgibbon', -1, 2, 0, 200, 3, 2.0, \
                         Row, Column, Phi, Radius1, Radius2, StartPhi, \
                         EndPhi, PointOrder)
gen_ellipse_contour_xld (EllMarks, Row, Column, Phi, Radius1, Radius2, \
                         StartPhi, EndPhi, PointOrder, 1.5)
length_xld(EllMarks,Length)
```

Result

`fit_ellipse_contour_xld` returns `2` (`H_MSG_TRUE`) if all parameter values are correct, and ellipses could be fitted to the input contours. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised. If the parameter `ClippingFactor` is chosen too small, i.e., all points are classified as outliers, the error 3264 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_contours_skeleton_xld`, `lines_gauss`, `lines_facet`, `edges_sub_pix`, `smooth_contours_xld`

Possible Successors

`gen_ellipse_contour_xld`, `disp_ellipse`, `get_points_ellipse`

See also

`fit_line_contour_xld`, `fit_circle_contour_xld`, `fit_rectangle2_contour_xld`

Module

Foundation

fit_line_contour_xld (Contours : : Algorithm, MaxNumPoints, ClippingEndPoints, Iterations, ClippingFactor : RowBegin, ColBegin, RowEnd, ColEnd, Nr, Nc, Dist)

Approximate XLD contours by line segments.

`fit_line_contour_xld` approximates the XLD contours `Contours` by line segments. It does not perform a segmentation of the input contours. Thus, one has to make sure that each contour corresponds to one and only one line segment. The operator returns for each contour the start point (`RowBegin`, `ColBegin`), the end point (`RowEnd`, `ColEnd`), and the regression line to the contour given by the normal vector (`Nr`, `Nc`) of the line and its distance `Dist` from the origin, i.e., the line equation is given by $r \cdot \mathtt{Nr} + c \cdot \mathtt{Nc} - \mathtt{Dist} = 0$.

The algorithm used for the fitting of lines can be selected via `Algorithm`:

'regression': Standard 'least squares' line fitting.

'huber': Weighted 'least squares' line fitting, where the impact of outliers is decreased based on the approach of Huber (see below).

'tukey': Weighted 'least squares' line fitting, where outliers are ignored based on the approach of Tukey (see below).

'drop': 'least squares' line fitting, where outliers are ignored. In particular, all contour points further away from the contour than the mean distance to the regression line multiplied with the `ClippingFactor` (see below) are ignored for the calculation of the undistorted regression line.

'gauss': Weighted 'least squares' line fitting, where the impact of outliers is decreased based on the mean value and the standard deviation of the distances of all contour points from the approximating line.

For *'huber'*, *'tukey'*, and *'drop'* a robust error statistics is used to estimate the standard deviation of the distances from the contour points *without* outliers from the approximating line. The parameter `ClippingFactor` (a scaling factor for the standard deviation) controls the amount of outliers: The smaller the value chosen for `ClippingFactor` the more outliers are detected. The detection of outliers is repeated. The parameter `Iterations` specifies the number of iterations. In the mode *'regression'* this value is ignored. Note that in the approach of Tukey (*'tukey'*), the outliers are removed before performing the approximation and all other points are weighted, whereas in the approach of Huber (*'huber'*), the outliers still have a small influence. Particularly, for outliers the optimization is influenced linearly and for points with a smaller distance it is influenced to the square. In practice, the approach of Tukey is recommended.

To reduce the computational load, the fitting of lines can be restricted to a subset of the contour points: If a value other than -1 is assigned to `MaxNumPoints`, only up to `MaxNumPoints` points - uniformly distributed over the contour - are used.

The start point and the end point of a line segment is determined by projecting the first and the last point of the corresponding contour to the approximating line. Due to artifacts in the pre-processing the start and end points of a contour might be faulty. Therefore, it is possible to exclude `ClippingEndPoints` points at the beginning and at the end of a contour from the line fitting. However, they are still used for the determination of the start point and the end point of the line segment.

The minimum necessary number of contour points for fitting a line is two. Therefore, it is required that the number of contour points is at least $2 + 2 \cdot$ `ClippingEndPoints`.

Parameters

- ▷ **`Contours`** (input_object) ..... xld_cont(-array) ⇝ *object*
  Input contours.
- ▷ **`Algorithm`** (input_control) ..... string ⇝ *string*
  Algorithm for the fitting of lines.
  Default: 'tukey'
  List of values: `Algorithm` ∈ {'regression', 'huber', 'tukey', 'gauss', 'drop'}
- ▷ **`MaxNumPoints`** (input_control) ..... integer ⇝ *integer*
  Maximum number of contour points used for the computation (-1 for all points).
  Default: -1
  Restriction: MaxNumPoints >= 2
- ▷ **`ClippingEndPoints`** (input_control) ..... integer ⇝ *integer*
  Number of points at the beginning and at the end of the contours to be ignored for the fitting.
  Default: 0
  Restriction: ClippingEndPoints >= 0
- ▷ **`Iterations`** (input_control) ..... integer ⇝ *integer*
  Maximum number of iterations (unused for *'regression'*).
  Default: 5
  Restriction: Iterations >= 0
- ▷ **`ClippingFactor`** (input_control) ..... real ⇝ *real*
  Clipping factor for the elimination of outliers (typical values: *1.0* for *'huber'* and *'drop'* and *2.0* for *'tukey'*).
  Default: 2.0
  Suggested values: `ClippingFactor` ∈ {1.0, 1.5, 2.0, 2.5, 3.0}
  Restriction: ClippingFactor > 0
- ▷ **`RowBegin`** (output_control) ..... line.begin.y(-array) ⇝ *real*
  Row coordinates of the starting points of the line segments.
- ▷ **`ColBegin`** (output_control) ..... line.begin.x(-array) ⇝ *real*
  Column coordinates of the starting points of the line segments.
- ▷ **`RowEnd`** (output_control) ..... line.end.y(-array) ⇝ *real*
  Row coordinates of the end points of the line segments.
- ▷ **`ColEnd`** (output_control) ..... line.end.x(-array) ⇝ *real*
  Column coordinates of the end points of the line segments.
- ▷ **`Nr`** (output_control) ..... number(-array) ⇝ *real*
  Line parameter: Row coordinate of the normal vector

▷ **Nc** (output_control) . . . . . number(-array) ⇝ *real*
Line parameter: Column coordinate of the normal vector

▷ **Dist** (output_control) . . . . . number(-array) ⇝ *real*
Line parameter: Distance of the line from the origin

Example

```
read_image (Image, 'mreut')
edges_sub_pix (Image, Edges, 'lanser2', 0.5, 20, 40)
gen_polygons_xld (Edges, Polygons, 'ramer', 2)
split_contours_xld (Polygons, Contours, 'polygon', 1, 5)
fit_line_contour_xld (Contours, 'regression', -1, 0, 5, 2, RowBegin, \
                      ColBegin, RowEnd, ColEnd, Nr, Nc, Dist)
```

Result

`fit_line_contour_xld` returns `2` (`H_MSG_TRUE`) if all parameter values are correct, and line segments could be fitted to the input contours. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised. If the parameter `ClippingFactor` is chosen too small, i.e., all points are classified as outliers, the error 3264 is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_contours_skeleton_xld`, `lines_gauss`, `lines_facet`, `edges_sub_pix`, `smooth_contours_xld`

Possible Successors

`disp_line`, `line_orientation`

See also

`regress_contours_xld`, `get_regress_params_xld`

Module

Foundation

fit_rectangle2_contour_xld (Contours : : Algorithm, MaxNumPoints, MaxClosureDist, ClippingEndPoints, Iterations, ClippingFactor : Row, Column, Phi, Length1, Length2, PointOrder)

Fit rectangles to XLD contours.

`fit_rectangle2_contour_xld` fits rectangles to the rectangular XLD contours given by `Contours` and returns the parameters of the rectangles in `Row`, `Column` (center), `Phi` (orientation), as well as `Length1` and `Length2` (half edge lengths). The angle `Phi` is returned in radians and specifies the angle between the horizontal axis and the edge with the half length `Length1` in the mathematically positive direction (counterclockwise). In addition, the point order of the contour is returned in `PointOrder`. `PointOrder` = *'positive'* means that the contour is traversed in the mathematically positive direction (counterclockwise).

The algorithm used for the fitting of the rectangles can be selected via `Algorithm`:

'regression' Standard least-squares line fitting.

'huber' Weighted least-squares line fitting, where the impact of outliers is decreased based on the approach of Huber (see below).

'tukey' Weighted least-squares line fitting, where outliers are ignored based on the approach of Tukey(see below).

For *'huber'* and *'tukey'*, a robust error statistics is used to estimate the standard deviation of the distances of the contour points from the approximating sides of the rectangle while ignoring outliers. The standard deviation is computed separately for each side of the rectangle to allow the processing of rectangles whose sides are not exactly perpendicular to each other. The parameter `ClippingFactor` (a scaling factor relative to the standard deviation) controls the amount of outliers: The smaller the value chosen for `ClippingFactor` the more outliers are detected. The detection of outliers is iterated. The parameter `Iterations` specifies the number of iterations. For `Algorithm` = *'regression'*, the values of the last two parameters are ignored. Note that in the approach of Tukey (*'tukey'*), the outliers are removed before performing the approximation and all other points are weighted, whereas in the approach of Huber (*'huber'*), the outliers still have a small influence. Particularly, for outliers the optimization is influenced linearly and for points with a smaller distance it is influenced to the square. For the algebraic approach, all distances of the points influence the optimization to the square and thus are not robust against outliers. In practice, the approach of Tukey is recommended.

To reduce the computational load, the fitting of rectangles can be restricted to a subset of the contour points: If a value other than *-1* is assigned to `MaxNumPoints`, only up to `MaxNumPoints` points, uniformly distributed across the contour, are used.

Depending on the processing used to create `Contours`, the start and end points of a contour may contain positional errors. Therefore, it is possible to exclude `ClippingEndPoints` points at the beginning and at the end of a contour from the rectangle fitting.

Contours, for which the distance between their start points and their end points is $\leq$ `MaxClosureDist` are considered to be closed. For closed contours, the end point of the contour is not used for the rectangle fitting because it would receive twice the weight of the remaining points in the fit.

The fitting of the rectangle to the contour is based on finding the correspondence between the contour points and the four sides of the rectangle. To enable a successful fit, there must be at least one point that lies in the interior of the line segment that represents the respective rectangle side, i.e., the point must not lie at the ends of the line segment. Because of this, at least eight contour points are necessary to fit the rectangle. A point is internally assigned to the side of the rectangle to which it has the minimum distance. For this, the currently optimal rectangle parameters, i.e., the parameters used for the current iteration step, are used internally. If no corresponding points are found for at least one side of the rectangle, the rectangle parameters cannot be determined uniquely. In this case, the error 3266 is returned. Because of this, the caller of `fit_rectangle2_contour_xld` must ensure that the input contours sufficiently resemble a rectangle. In particular, none of the interior angles of the contour, if the contour was approximated by four lines, should be smaller than 45 degrees or larger than 135 degrees. Because of the assignment of contour points to the closest side of the rectangle this would mean that at least one side of the rectangle would have no corresponding points. Furthermore, `ClippingFactor` should not be chosen too small to avoid that the outlier suppression creates rectangle sides without corresponding contour points. This can only happen for `Algorithm` = *'tukey'*. If the above conditions are observed, `fit_rectangle2_contour_xld` returns highly accurate rectangle parameters. If the outlier suppression according to Tukey is used, `fit_rectangle2_contour_xld` can be used to robustly fit rectangles, e.g., to rectangular contours with rounded corners.

Parameters

▷ **`Contours`** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contours.

▷ **`Algorithm`** (input_control) ..... string ⇝ *string*
Algorithm for fitting the rectangles.
Default: 'regression'
List of values: `Algorithm` ∈ {'regression', 'huber', 'tukey'}

▷ **`MaxNumPoints`** (input_control) ..... integer ⇝ *integer*
Maximum number of contour points used for the computation (-1 for all points).
Default: -1
Restriction: MaxNumPoints == -1 || MaxNumPoints >= 8

▷ **`MaxClosureDist`** (input_control) ..... real ⇝ *real*
Maximum distance between the end points of a contour to be considered as closed.
Default: 0.0
Restriction: MaxClosureDist >= 0.0

▷ **ClippingEndPoints** (input_control) . . . . . integer ⇝ *integer*
Number of points at the beginning and at the end of the contours to be ignored for the fitting.
Default: 0
Suggested values: ClippingEndPoints ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
Restriction: ClippingEndPoints >= 0

▷ **Iterations** (input_control) . . . . . integer ⇝ *integer*
Maximum number of iterations (not used for *'regression'*).
Default: 3
Restriction: Iterations >= 0

▷ **ClippingFactor** (input_control) . . . . . real ⇝ *real*
Clipping factor for the elimination of outliers (typical values: *1.0* for *'huber'* and *2.0* for *'tukey'*).
Default: 2.0
Suggested values: ClippingFactor ∈ {1.0, 1.5, 2.0, 2.5, 3.0}
Restriction: ClippingFactor > 0

▷ **Row** (output_control) . . . . . rectangle2.center.y(-array) ⇝ *real*
Row coordinate of the center of the rectangle.

▷ **Column** (output_control) . . . . . rectangle2.center.x(-array) ⇝ *real*
Column coordinate of the center of the rectangle.

▷ **Phi** (output_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real*
Orientation of the main axis of the rectangle [rad].

▷ **Length1** (output_control) . . . . . rectangle2.hwidth(-array) ⇝ *real*
First radius (half length) of the rectangle.

▷ **Length2** (output_control) . . . . . rectangle2.hheight(-array) ⇝ *real*
Second radius (half width) of the rectangle.

▷ **PointOrder** (output_control) . . . . . string(-array) ⇝ *string*
Point order of the contour.
List of values: PointOrder ∈ {'positive', 'negative'}

Result

fit_rectangle2_contour_xld returns 2 (H_MSG_TRUE) if all parameter values are correct, and rectangles could be fitted to the input contours. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised. If the parameter ClippingFactor is chosen too small, i.e., all points are classified as outliers, the error 3266 is raised. If no points could be found for at least one side of the rectangle, the error 3266 is raised as well.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix, smooth_contours_xld, union_collinear_contours_xld, union_collinear_contours_ext_xld, union_adjacent_contours_xld

Possible Successors

dist_rectangle2_contour_points_xld, gen_rectangle2_contour_xld

Alternatives

smallest_rectangle2_xld

See also

fit_line_contour_xld, fit_circle_contour_xld, fit_ellipse_contour_xld

Module

Foundation

get_contour_angle_xld (Contour : : AngleMode, CalcMode, Lookaround : Angles)

Calculate the direction of an XLD contour for each contour point.

get_contour_angle_xld calculates for each point of the XLD contour Contour the direction of its tangent. Two modes for the output values can be chosen: By passing *'abs'* for AngleMode, the resulting angles are returned relative to the horizontal axis as angles between 0 and 2π (counter-clockwise). By passing *'rel'*, the angle difference to the previous contour point is returned. In this case the range of values is between $-\pi$ and π, where negative values indicate a right turn and positive values indicate a left turn.

There are three different ways of calculating the tangent direction (CalcMode) of the contour point i using the contour points in the interval from $i -$ Lookaround to $i +$ Lookaround. By using *'range'*, the angle of the line segment between the first and last point of the interval is used. For *'mean'*, the average of all angles between consecutive points of the contour is used. Finally, for *'regress'*, the direction of the regression line (the least squares fit of a line to the contour points in the interval) is used. Lookaround is a measure of how strongly the contour is smoothed. The angles are returned in Angles in radians.

Due to the interval given by Lookaround, the input contour must have at least $2 *$ Lookaround $+ 2$ points. This minimum number of points is required so that the operator always delivers reasonable output values for the different parameter settings.

Parameters

▷ **Contour** (input_object) ..... xld_cont ⇝ *object*
Input contour.

▷ **AngleMode** (input_control) ..... string ⇝ *string*
Return type of the angles.
Default: 'abs'
List of values: AngleMode ∈ {'abs', 'rel'}

▷ **CalcMode** (input_control) ..... string ⇝ *string*
Method for computing the angles.
Default: 'range'
List of values: CalcMode ∈ {'range', 'mean', 'regress'}

▷ **Lookaround** (input_control) ..... integer ⇝ *integer*
Number of points to take into account.
Default: 3
Restriction: Lookaround > 0

▷ **Angles** (output_control) ..... real-array ⇝ *real*
Direction of the tangent to the contour points.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix

See also

get_contour_xld, get_contour_attrib_xld

Module

Foundation

get_contour_attrib_xld (Contour : : Name : Attrib)

Return point attribute values of an XLD contour.

get_contour_attrib_xld returns the values of the attribute Name of the XLD contour Contour in Attrib. Contour point attributes are additional values defined for each contour point and describe local features. query_contour_attribs_xld can be used to query which attributes are set for a particular contour.

The following list contains information about the different contour point attributes and the operators that add them to XLD contours (for exceptions see the respective operator references):

'angle'

The direction of the normal vectors of the contour is described by *'angle'* [rad] (see image below). It is oriented such that the normal vectors point to the right side of the contour as the contour is traversed from start to end point (the angles are defined counterclockwise with respect to the row axis of the image).

The attribute *'angle'* is added by the following operators:

edges_color_sub_pix, edges_sub_pix, lines_color, lines_facet, lines_gauss

'response'

'response' contains the magnitude of the edge gradient (see image below).

The attribute *'response'* is added by the following operators:

edges_color_sub_pix, edges_sub_pix, lines_color, lines_facet, lines_gauss

'width_right'

The line width to the right of the line (as the contour is traversed from start to end point) is described by *'width_right'* [px] (see image below).

The attribute *'width_right'* is added by the following operators:

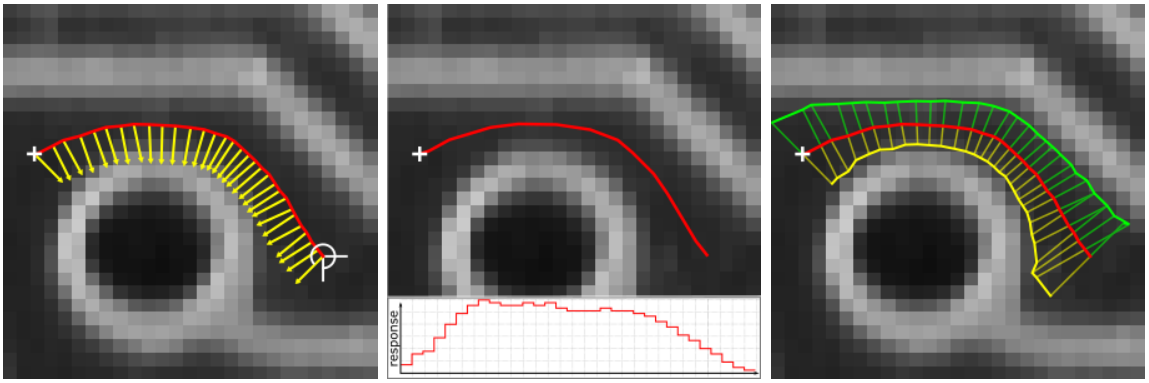
lines_color, lines_gauss

'width_left'

The line width to the left of the line (as the contour is traversed from start to end point) is described by *'width_left'* [px] (see image below).

The attribute *'width_left'* is added by the following operators:

lines_color, lines_gauss



(1) (2) (3)

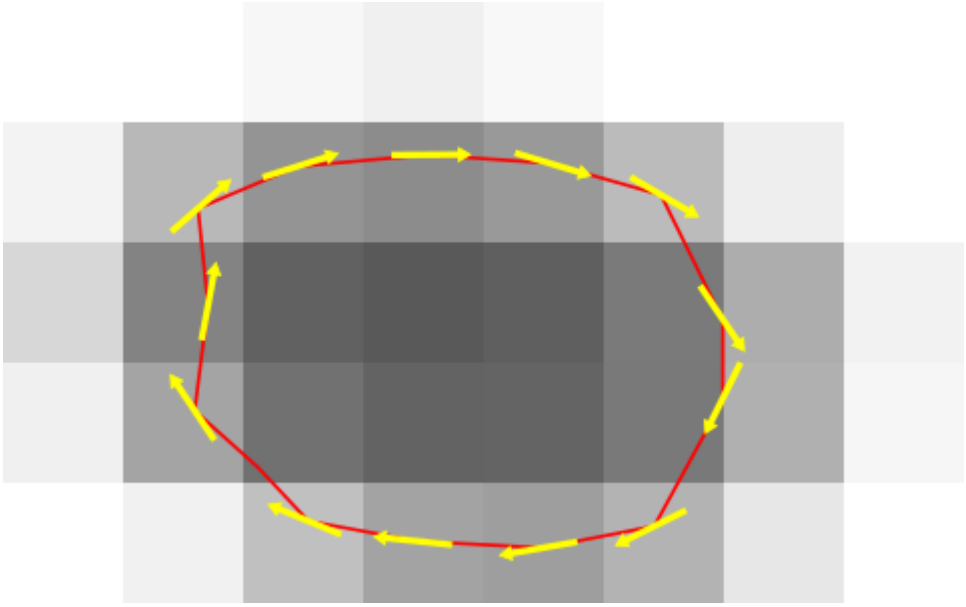
Visualization of different point attributes of a contour (red). The starting point of the contour is marked with a white cross. (1) Vectors (yellow), plotted at *'angle'* (with respect to the row axis), representing the normal for every point of the contour, (2) behavior of the attribute *'response'* along a contour, and (3) visualization of the calculated attributes *'width_right'* (yellow) and *'width_left'* (green).

'edge_direction'

Gives the direction of the edge (not of the XLD contour), calculated from image gradients in direction of rows and columns. The angles are given with respect to the column axis of the image.

The attribute *'edge_direction'* [rad] is added by the following operators:

edges_color_sub_pix, edges_sub_pix



Vectors plotted in *'edge_direction'* (yellow) for every point of a contour (red).

'asymmetry'

The contour attribute *'asymmetry'* describes the image gradient on both sides of the edge. It is positive if the asymmetric part, i.e., the part with the weaker gradient, is on the right side of the line, while it is negative if the asymmetric part is on the left side of the line (see image below).

The attribute *'asymmetry'* is added by the following operator:

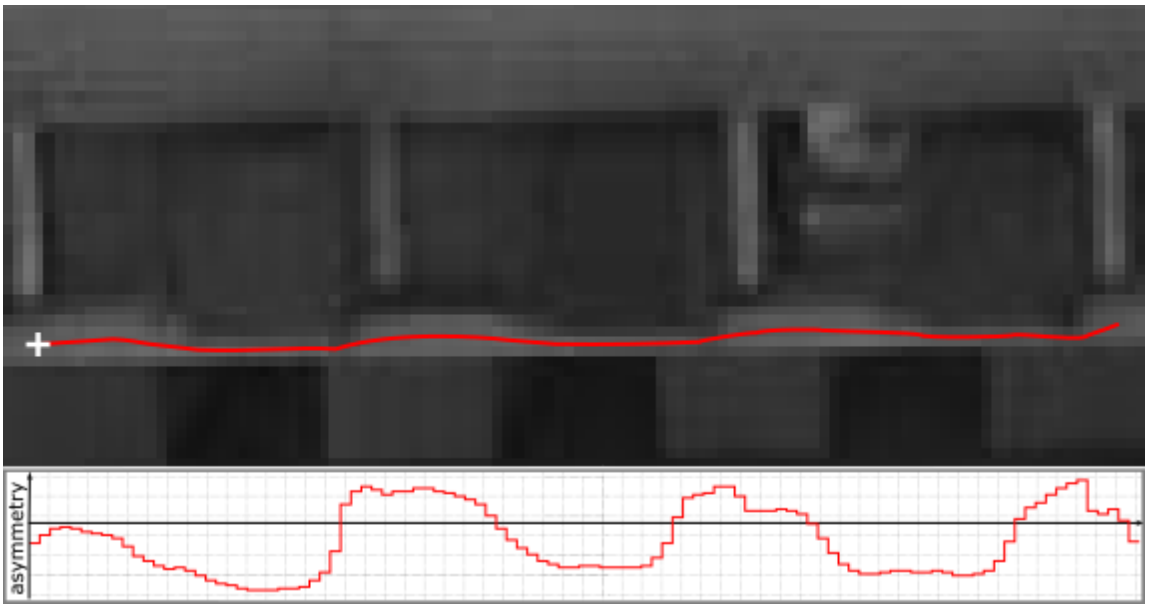
`lines_gauss`

'contrast'

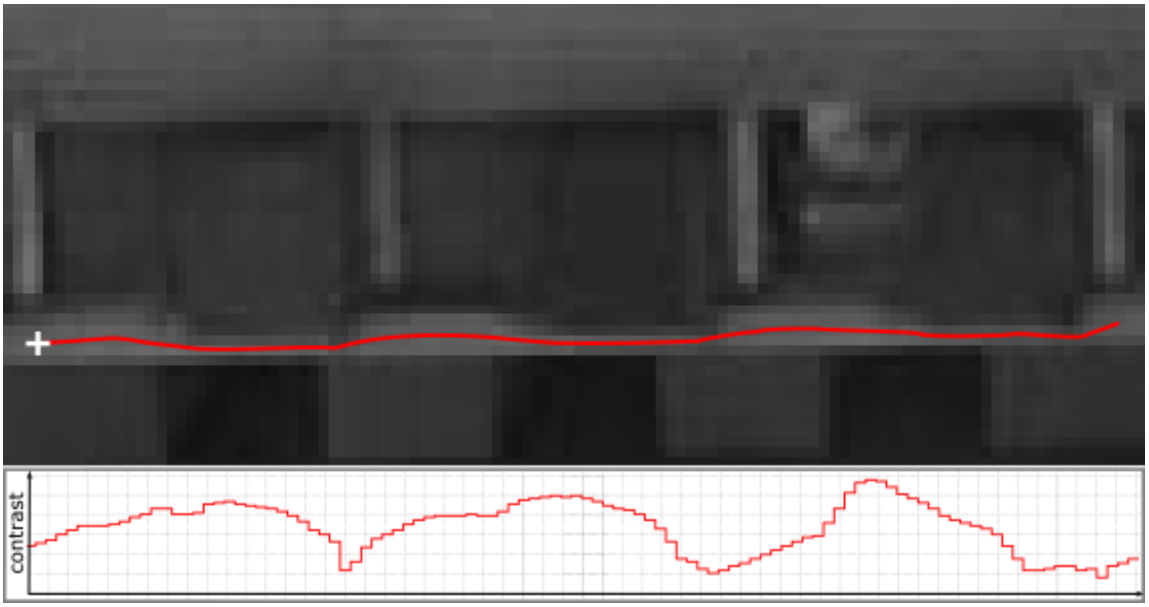
The contrast of a contour describes the difference between the gray values of the line and the gray values of the local background. It is positive if bright lines are extracted, while it is negative for dark lines (see image below).

The attribute *'contrast'* is added by the following operator:

`lines_gauss`



(1)



(2)

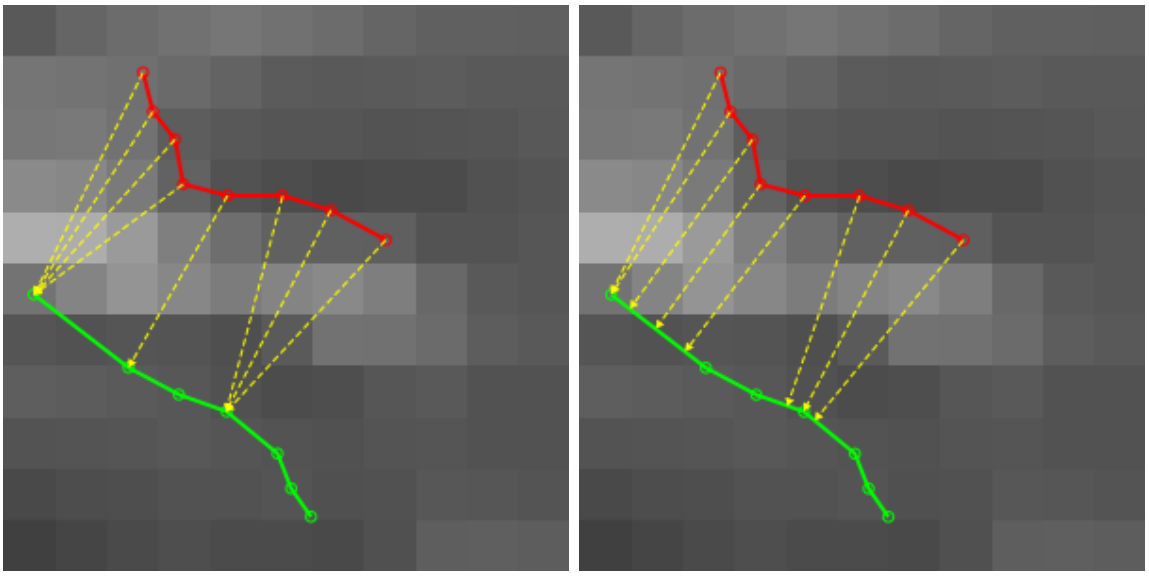
behavior of the attributes (1) *'asymmetry'* and (2) *'contrast'* for a contour along an image structure (the starting point of the contour is marked with a white cross).

'distance'

The minimum distance to any point or segment of the reference contour (depending on the mode chosen for the calculation) is given in the attribute *'distance'* [px] for all points of a contour.

The attribute *'distance'* is added by the following operators:

`apply_distance_transform_xld`, `distance_contours_xld`



(1) (2)

(1) *'distance'* of a contour (red) to the points of a reference contour (green) and (2) *'distance'* to any segment of a reference contour (green).

For information about global contour attributes see the operator reference of `get_contour_global_attrib_xld`.

Parameters

▷ **`Contour`** (input_object) ..... xld_cont ⇝ *object*
Input XLD contour.

▷ **`Name`** (input_control) ..... string ⇝ *string*
Name of the attribute.
Default: 'angle'
Suggested values: `Name` ∈ {'angle', 'edge_direction', 'width_right', 'width_left', 'response', 'contrast', 'asymmetry', 'distance'}

▷ **`Attrib`** (output_control) ..... real-array ⇝ *real*
Attribute values.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`lines_gauss`, `lines_facet`, `edges_sub_pix`, `distance_contours_xld`, `apply_distance_transform_xld`

See also

`query_contour_attribs_xld`, `get_contour_global_attrib_xld`, `query_contour_global_attribs_xld`

Module

Foundation

`get_contour_global_attrib_xld` (`Contour` : : `Name` : `Attrib`)

Return global attributes values of an XLD contour.

`get_contour_global_attrib_xld` returns the values of the global attribute `Name` of the XLD contour `Contour` in `Attrib`. Global attributes are additional values defined for each contour.

query_contour_global_attribs_xld can be used to query which global attributes are set for a particular contour.

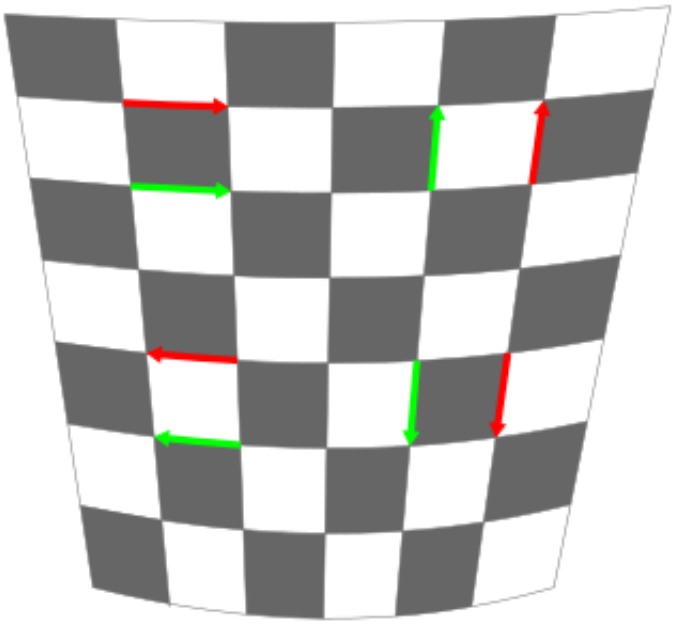
The following list contains information about the different global contour attributes and the operators that add them to XLD contours:

'bright_dark'

The type of transition for each output XLD contour is stored in the attribute *'bright_dark'*. It is set to *1.0*, if the connecting line forms a bright-dark transition (left to right, viewed from start point to end point), otherwise it is set to *0.0*.

The attribute *'bright_dark'* is added by the following operator:

connect_grid_points



Exemplary XLD contours (red and green) along the edges of a rectification grid (direction from start to end point indicated by arrow head). The transition type, stored in the attribute *'bright_dark'*, is *1.0* for red contours and *0.0* for green contours.

'cont_approx'

The best way to approximate a contour is stated by the attribute *'cont_approx'*: for *'cont_approx'=-1.0*, the contour is best approximated by a line segment, for *'cont_approx'=0.0*, by an elliptic arc, and for *'cont_approx'=1.0*, by a circular arc.

The attribute *'cont_approx'* is added by the following operator:

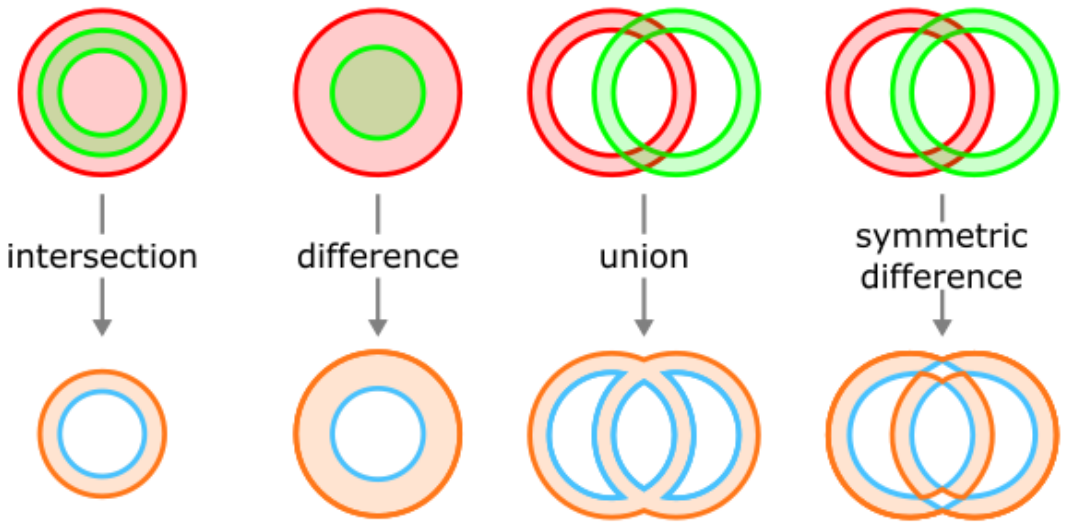
segment_contours_xld

'is_hole'

For boundaries that enclose holes, the global contour attribute *'is_hole'* is set to *1.0*, otherwise, it is set to *0.0*.

The attribute *'is_hole'* is added by the following operators:

symm_difference_closed_contours_xld, difference_closed_contours_xld, intersection_closed_contours_xld, union2_closed_contours_xld



Applying different set operations to XLD contours or contour sets adds the global attribute *'is_hole'* to the resulting contour(s). In the example above, for two sets (red and green) of closed contours different set operations are performed. If a resulting boundary encloses a hole (blue contours), *'is_hole'* is set to *1.0*, otherwise the value of *'is_hole'* is *0.0* (orange contours).

'regr_dev_dist'

'regr_dev_dist' [px] gives the standard deviation of the (Euclidean) distances between the contour points and the regression line (see image below).

The attribute *'regr_dev_dist'* is added by the following operator:

`regress_contours_xld`

'regr_dist'

'regr_dist' [px] gives the minimal distance of the regression line from the origin of the image coordinate system (see image below).

The attribute *'regr_dist'* is added by the following operator:

`regress_contours_xld`

'regr_mean_dist'

The attribute *'regr_mean_dist'* [px] contains the mean of the Euclidean distances between each contour point and the regression line (see image below).

The attribute *'regr_mean_dist'* is added by the following operator:

`regress_contours_xld`

'regr_norm_col'

'regr_norm_col' [px] is the column coordinate of the unit normal vector of the regression line with the normal vector pointing from the origin to the line (see image below).

The attribute *'regr_norm_col'* is added by the following operator:

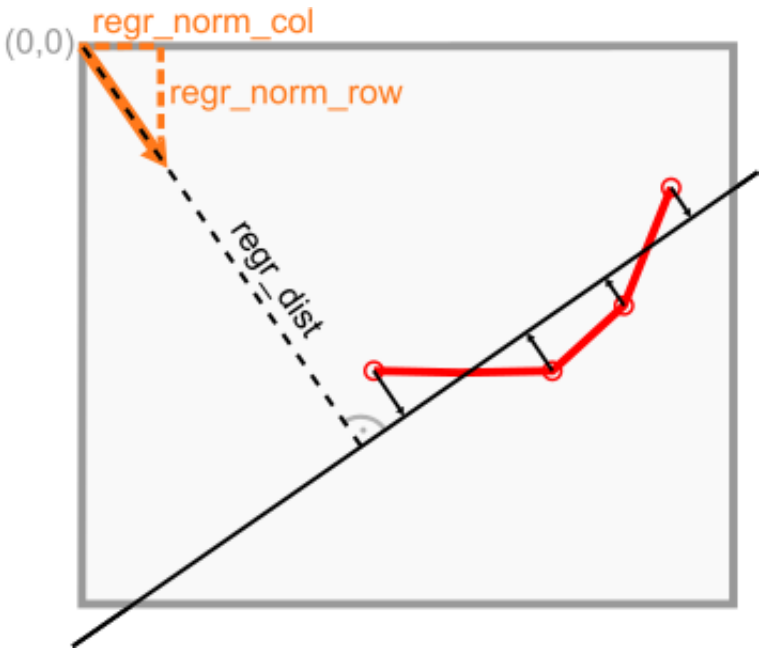
`regress_contours_xld`

'regr_norm_row'

'regr_norm_row' [px] is the row coordinate of the unit normal vector of the regression line with the normal vector pointing from the origin to the line (see image below).

The attribute *'regr_norm_row'* is added by the following operator:

`regress_contours_xld`



Sketch of the attributes *'regr_dist'*, *'regr_norm_col'*, *'regr_norm_row'* for the regression of an XLD contour (red). *'regr_mean_dist'* and *'regr_dev_dist'* are computed from the distances (black arrows) between the contour points and the regression line (solid black line).

Parameters

▷ **Contour** (input_object) ..... xld_cont ⇝ *object*
Input XLD contour.

▷ **Name** (input_control) ..... string(-array) ⇝ *string*
Name of the attribute.
Default: 'regr_norm_row'
Suggested values: Name ∈ {'regr_norm_row', 'regr_norm_col', 'regr_mean_dist', 'regr_dev_dist', 'cont_approx', 'bright_dark', 'is_hole'}

▷ **Attrib** (output_control) ..... real-array ⇝ *real*
Attribute values.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

lines_gauss, lines_facet, edges_sub_pix, segment_contours_xld

Possible Successors

fit_circle_contour_xld, fit_ellipse_contour_xld, fit_line_contour_xld, fit_rectangle2_contour_xld

See also

query_contour_global_attribs_xld, get_contour_attrib_xld, query_contour_attribs_xld

Module

Foundation

get_regress_params_xld (Contours : : : Length, Nx, Ny, Dist, Fpx, Fpy, Lpx, Lpy, Mean, Deviation)

Return XLD contour parameters.

get_regress_params_xld returns the following parameters for all XLD contours given in Contours:

- the number of contour points Length,
- the coordinates Nx and Ny of the normal vector of the regression line (i.e., least-squares approximating line),
- the distance Dist of the regression line to the origin
- the sub-pixel precise coordinates Fpx and Fpy of the perpendicular projection of the start point of the contour onto the regression line,
- the sub-pixel precise coordinates Lpx and Lpy of the perpendicular projection of the end point of the contour onto the regression line,
- the mean of the Euclidean distance of the contour points from the regression line,
- the standard deviation of these distances.

Attention

Before the contour parameters can be returned by get_regress_params_xld, the parameters of the regression line to the contour must be calculated by calling regress_contours_xld.

Parameters

▷ **Contours** (input_object) ..... xld_cont-array ⇝ *object*
Input XLD contours.

▷ **Length** (output_control) ..... integer-array ⇝ *integer*
Number of contour points.

▷ **Nx** (output_control) ..... point.x-array ⇝ *real*
X-coordinate of the normal vector of the regression line.

▷ **Ny** (output_control) ..... point.y-array ⇝ *real*
Y-coordinate of the normal vector of the regression line.

▷ **Dist** (output_control) ..... number-array ⇝ *real*
Distance of the regression line from the origin.

▷ **Fpx** (output_control) ..... point.x-array ⇝ *real*
X-coordinate of the projection of the start point of the contour onto the regression line.

▷ **Fpy** (output_control) ..... point.y-array ⇝ *real*
Y-coordinate of the projection of the start point of the contour onto the regression line.

▷ **Lpx** (output_control) ..... point.x-array ⇝ *real*
X-coordinate of the projection of the end point of the contour onto the regression line.

▷ **Lpy** (output_control) ..... point.y-array ⇝ *real*
Y-coordinate of the projection of the end point of the contour onto the regression line.

▷ **Mean** (output_control) ..... real-array ⇝ *real*
Mean distance of the contour points from the regression line.

▷ **Deviation** (output_control) ..... real-array ⇝ *real*
Standard deviation of the distances from the regression line.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

regress_contours_xld

Possible Successors

disp_line, line_orientation

See also

fit_line_contour_xld, get_contour_global_attrib_xld, query_contour_global_attribs_xld, get_contour_xld, get_contour_attrib_xld, gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix

Module

Foundation

height_width_ratio_xld (XLD : : : Height, Width, Ratio)

Compute the width, height, and aspect ratio of the enclosing rectangle parallel to the coordinate axes of contours or polygons.

The operator height_width_ratio_xld calculates the enclosing rectangle (parallel to the coordinate axes) for each input contour or polygon. The enclosing rectangle is described by the coordinates of the corner pixels (Row1,Column1,Row2,Column2) (see smallest_rectangle1_xld). Based on these values, height_width_ratio_xld computes the width, height, and aspect ratio of the smallest surrounding rectangle as follows:

$$\begin{aligned} \texttt{Width} &= \texttt{Column2} - \texttt{Column1} \\ \texttt{Height} &= \texttt{Row2} - \texttt{Row1} \\ \texttt{Ratio} &= \texttt{Height}/\texttt{Width} \end{aligned}$$

If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD. In case of an empty contour all parameters have the value 0 if no other behavior was set (see set_system).

Attention

If Width = 0, Ratio is set to DBL_MAX (1.797e308).

Parameters

- ▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
  Contours or polygons to be examined.
- ▷ **Height** (output_control) ..... extent.y(-array) ⇝ *real*
  Height of the enclosing rectangle.
- ▷ **Width** (output_control) ..... extent.x(-array) ⇝ *real*
  Width of the enclosing rectangle.
- ▷ **Ratio** (output_control) ..... real(-array) ⇝ *real*
  Aspect ratio of the enclosing rectangle.

Complexity

If N is the number of contour points, the runtime complexity is $O(N)$.

Result

height_width_ratio_xld returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld

Possible Successors

gen_polygons_xld

Alternatives

smallest_rectangle1_xld, smallest_rectangle2_xld, shape_trans_xld

See also

shape_trans_xld, smallest_circle_xld, elliptic_axis_xld, area_center_xld

Module

Foundation

info_parallels_xld (Parallels, Image : : : QualityMin, QualityMax, GrayMin, GrayMax, StandardMin, StandardMax)

Return information about the gray values of the area enclosed by XLD parallels.

info_parallels_xld calculates various gray value features of the area enclosed by the XLD parallels Parallels. The input image Image is used to get the gray values needed for this. The algorithm used in this operator is very similar to the one used in mod_parallels_xld. The operator returns ranges for the quality factor (QualityMin and QualityMax, see gen_parallels_xld for the corresponding calculation), the mean gray value (GrayMin and GrayMax), and the standard deviation with respect to the mean gray value (StandardMin and StandardMax).

This operator serves to determine appropriate thresholds for mod_parallels_xld.

Parameters

- ▷ **Parallels** (input_object) ..... xld_para-array ⇝ *object*
  Input XLD Parallels.
- ▷ **Image** (input_object) ..... singlechannelimage ⇝ *object :* byte
  Corresponding gray value image.
- ▷ **QualityMin** (output_control) ..... real ⇝ *real*
  Minimum quality factor.
- ▷ **QualityMax** (output_control) ..... real ⇝ *real*
  Maximum quality factor.
- ▷ **GrayMin** (output_control) ..... integer ⇝ *integer*
  Minimum mean gray value.
- ▷ **GrayMax** (output_control) ..... integer ⇝ *integer*
  Maximum mean gray value.
- ▷ **StandardMin** (output_control) ..... real ⇝ *real*
  Minimum standard deviation.
- ▷ **StandardMax** (output_control) ..... real ⇝ *real*
  Maximum standard deviation.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_parallels_xld

Possible Successors

mod_parallels_xld

See also

intensity, min_max_gray

Module

Foundation

length_xld (XLD : : Length)

Length of contours or polygons.

length_xld calculates the length of the contours or polygons XLD. The length is calculated as the sum of the Euclidean distances of successive points on the contour or polygon. If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) . . . . . xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Length** (output_control) . . . . . real(-array) ⇝ *real*
Length of the contour or polygon.
Assertion: Length >= 0

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

`length_xld` returns `2` (`H_MSG_TRUE`) if the input is not empty. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`gen_contours_skeleton_xld`, `smooth_contours_xld`, `gen_polygons_xld`

See also

`area_center_xld`, `moments_any_xld`, `moments_xld`, `contlength`

Module

Foundation

local_max_contours_xld (Contours, Image : LocalMaxContours : MinPercent, MinDiff, Distance :)

Select XLD contours with a local maximum of gray values.

`local_max_contours_xld` selects XLD contours from the contours passed in `Contours`, which have a local maximum in gray values across the direction of the contour. In order to be selected, at least `MinPercent` of the contour points must have a local maximum perpendicular to the direction of the contour. The contours' direction is determined by fitting a regression line through five neighboring points of the contour. In order to decide whether there is a local maximum for a contour point, a gray value profile that is `Distance` points wide and perpendicular to the contour is computed on both sides of the contour. If the gray value at the contour point is at least `MinDiff` larger than the gray value at at least one point on each side of the profile, the contour point is labeled as a local maximum. The selected contours are returned in `LocalMaxContours`.

Parameters

▷ **Contours** (input_object) . . . . . xld_cont-array ⇝ *object*
XLD contours to be examined.

▷ **Image** (input_object) . . . . . singlechannelimage ⇝ *object :* byte
Corresponding gray value image.

▷ **LocalMaxContours** (output_object) . . . . . xld_cont-array ⇝ *object*
Selected contours.

▷ **MinPercent** (input_control) . . . . . number ⇝ *integer* / real
Minimum percentage of maximum points.
Default: 70
Suggested values: `MinPercent` ∈ {60, 70, 75, 80, 85, 90, 95}
Restriction: 0.0 <= MinPercent && MinPercent <= 100.0

▷ **MinDiff** (input_control) . . . . . integer ⇝ *integer*
Minimum amount by which the gray value at the maximum must be larger than in the profile.
Default: 15
Suggested values: `MinDiff` ∈ {5, 8, 10, 12, 15, 20}
Restriction: 0 <= MinDiff && MinDiff <= 255

▷ **Distance** (input_control) . . . . . integer ⇝ *integer*
Maximum width of profile used to check for maxima.
Default: 4
Suggested values: Distance ∈ {2, 3, 4, 5, 6}
Restriction: Distance >= 1

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix

Possible Successors

gen_polygons_xld

See also

smooth_contours_xld

Module

Foundation

max_parallels_xld (ExtParallels : MaxPolygons : :)

Join modified XLD parallels lying on the same polygon.

max_parallels_xld joins all modified XLD parallels in ExtParallels into a polygon if they lie on the same original polygon segment. This means that polygons exhibiting parallelism and enclosing homogeneous areas in several places are joined into one long polygon (from the first parallel line to the last parallel line). The resulting polygons are returned in MaxPolygons.

Parameters

▷ **ExtParallels** (input_object) . . . . . xld_ext_para-array ⇝ *object*
Extended XLD parallels.
▷ **MaxPolygons** (output_object) . . . . . xld_poly-array ⇝ *object*
Maximally extended parallels.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

mod_parallels_xld

Possible Successors

get_polygon_xld, get_lines_xld

Module

Foundation

moments_any_points_xld (XLD : : Mode, Area, CenterRow, CenterCol, P, Q : M)

Arbitrary geometric moments of contours or polygons treated as point clouds.

moments_any_points_xld calculates arbitrary moments M of the point clouds given by the contours or polygons XLD (i.e., the order of the points in the contour or polygon is not taken into account).

The computed moments are normalized depending on the desired mode Mode:

'unnormalized': No normalization. The computed moment $M_{p,q}$ is:

$$M_{p,q} = \sum_{i=0}^{n-1} r_i^p c_i^q$$

'unnormalized_central': Shift the contour points by their centroid $[\bar{r}, \bar{c}] = [\texttt{CenterRow}, \texttt{CenterCol}]$:

$$M_{p,q} = \sum_{i=0}^{n-1} (r_i - \bar{r})^p (c_i - \bar{c})^q$$

'normalized': Normalization by the area $A = \texttt{Area}$ from XLD:

$$M_{p,q} = \frac{1}{A} \sum_{i=0}^{n-1} r_i^p c_i^q$$

'central': Normalization by the area $A = \texttt{Area}$ and a shift of the contour points by their centroid $[\bar{r}, \bar{c}] = [\texttt{CenterRow}, \texttt{CenterCol}]$:

$$M_{p,q} = \frac{1}{A} \sum_{i=0}^{n-1} (r_i - \bar{r})^p (c_i - \bar{c})^q$$

For the normalization of the moments three specific values are used: The area Area and the coordinates CenterRow,CenterCol of it's centroid (this values can be computed with area_center_points_xld).

If the contour or polygon is closed (end point = start point), the end point of the contour or polygon is not taken into account to avoid that it receives twice the weight of the other points.

moments_any_xld should be used if the contour XLD intersects itself or if it is not possible to close the contour using a line from end to start point without self-intersection, because in this case moments_any_xld does not produce useful results. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used.

If more than one contour or polygon is passed, M contains all desired moments of the first contour/polygon followed by all the moments of the second contour/polygon and so forth.

Parameters

- ▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
  Contours or polygons to be examined.
- ▷ **Mode** (input_control) ..... string ⇝ *string*
  Computation mode.
  Default: 'unnormalized'
  Suggested values: Mode ∈ {'unnormalized', 'unnormalized_central', 'normalized', 'central'}
- ▷ **Area** (input_control) ..... real(-array) ⇝ *real*
  Area enclosed by the contour or polygon.
- ▷ **CenterRow** (input_control) ..... point.y(-array) ⇝ *real*
  Row coordinate of the centroid.
- ▷ **CenterCol** (input_control) ..... point.x(-array) ⇝ *real*
  Column coordinate of the centroid.
- ▷ **P** (input_control) ..... point.x(-array) ⇝ *integer*
  First index of the desired moments $M_{p,q}$.
  Default: 1
  Suggested values: P ∈ {0, 1, 2, 3, 4}
- ▷ **Q** (input_control) ..... point.x(-array) ⇝ *integer*
  Second index of the desired moments $M_{p,q}$.
  Default: 1
  Suggested values: Q ∈ {0, 1, 2, 3, 4}
- ▷ **M** (output_control) ..... real(-array) ⇝ *real*
  The computed moments.

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

`moments_any_points_xld` returns `2` (`H_MSG_TRUE`) if the input is not empty. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`area_center_points_xld`, `gen_contours_skeleton_xld`, `smooth_contours_xld`, `gen_polygons_xld`

Alternatives

`moments_points_xld`, `moments_any_xld`

See also

`moments_points_xld`, `area_center_points_xld`, `moments_region_2nd`, `area_center`

Module

Foundation

moments_any_xld (XLD : : Mode, PointOrder, Area, CenterRow, CenterCol, P, Q : M)

Arbitrary geometric moments of contours or polygons.

`moments_any_xld` calculates arbitrary moments `M` of the regions enclosed by the contours or polygons `XLD`. The moments are computed by applying Green's theorem using only the points on the contour or polygon, i.e., no region is generated explicitly for the purpose of calculating the features. It is assumed that the contours or polygons are closed. If this is not the case `moments_any_xld` will artificially close the contours or polygons.

It should be noted that `moments_any_xld` only returns useful results if the contour or polygon encloses a region in the plane. In particular, the contour or polygon must not intersect itself. This is particularly important if open contours or polygons are passed because they are closed automatically, which can produce a self-intersection. To test whether the contours or polygons intersect themselves, `test_self_intersection_xld` can be used. If the contour or polygon intersects itself, useful values for the moments can be calculated with `moments_any_points_xld`.

The computed moments are normalized depending on the desired mode `Mode`:

'unnormalized': No normalization. Let R be the enclosed image region. Then the computed moment $M_{p,q}$ is equivalent to:

$$M_{p,q} = \int_R r^p c^q \, dr \, dc$$

'unnormalized_central': Shift the region by its centroid $[\bar{r}, \bar{c}] = $ [`CenterRow`, `CenterCol`]:

$$M_{p,q} = \int_R (r - \bar{r})^p (c - \bar{c})^q \, dr \, dc$$

'normalized': Normalization by the area $A = $ `Area` of the enclosed image region:

$$M_{p,q} = \frac{1}{A} \int_R r^p c^q \, dr \, dc$$

'central': Normalization by the area $A = $ `Area` of the enclosed image region and a shift of the region by it's centroid $[\bar{r}, \bar{c}] = $ [`CenterRow`, `CenterCol`]:

$$M_{p,q} = \frac{1}{A} \int_R (r - \bar{r})^p (c - \bar{c})^q \, dr \, dc$$

For the normalization of the moments three specific moments are used: The area `Area` of the enclosed image region and the coordinates `CenterRow`,`CenterCol` of it's centroid (this values can be calculated with `area_center_xld`). In addition to that `moments_any_xld` expects information about the point order of the input contours/polygons in `PointOrder`, see `area_center_xld` again.

If more than one contour or polygon is passed, `M` contains all desired moments of the first contour/polygon followed by all the moments of the second contour/polygon and so forth.

Parameters

- ▷ **`XLD`** (input_object) ..... xld(-array) ⇝ *object*
  Contours or polygons to be examined.
- ▷ **`Mode`** (input_control) ..... string ⇝ *string*
  Computation mode.
  Default: 'unnormalized'
  Suggested values: `Mode` ∈ {'unnormalized', 'unnormalized_central', 'normalized', 'central'}
- ▷ **`PointOrder`** (input_control) ..... string(-array) ⇝ *string*
  Point order along the boundary.
  Default: 'positive'
  Suggested values: `PointOrder` ∈ {'positive', 'negative'}
- ▷ **`Area`** (input_control) ..... real(-array) ⇝ *real*
  Area enclosed by the contour or polygon.
- ▷ **`CenterRow`** (input_control) ..... point.y(-array) ⇝ *real*
  Row coordinate of the centroid.
- ▷ **`CenterCol`** (input_control) ..... point.x(-array) ⇝ *real*
  Column coordinate of the centroid.
- ▷ **`P`** (input_control) ..... point.x(-array) ⇝ *integer*
  First index of the desired moments $M_{p,q}$.
  Default: 1
  Suggested values: `P` ∈ {0, 1, 2, 3, 4}
- ▷ **`Q`** (input_control) ..... point.x(-array) ⇝ *integer*
  Second index of the desired moments $M_{p,q}$.
  Default: 1
  Suggested values: `Q` ∈ {0, 1, 2, 3, 4}
- ▷ **`M`** (output_control) ..... real(-array) ⇝ *real*
  The computed moments.

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

`moments_any_xld` returns `2` (`H_MSG_TRUE`) if the input is not empty. If the input is empty the behavior can be set via `set_system(::'no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`area_center_xld`, `gen_contours_skeleton_xld`, `smooth_contours_xld`, `gen_polygons_xld`

Alternatives

`moments_xld`, `moments_any_points_xld`

See also

`moments_xld`, `area_center_xld`, `moments_region_2nd`, `area_center`

Module

Foundation

moments_points_xld (XLD : : : M11, M20, M02)

Geometric moments M_{20}, M_{02}, and M_{11} of contours or polygons treated as point clouds.

moments_points_xld calculates the moments (M20, M02, and M11) of a point cloud given by a contour or a polygon XLD (i.e., the order of the points in the contour or polygon is not taken into account). Therefore, the moments are given by

$$M_{p,q} = \sum_{i=0}^{n-1} (r_i - \bar{r})^p (c_i - \bar{c})^q.$$

. Here, $(\bar{r}, \bar{c})$ is the centroid and n the number of points.

If the contour or polygon is closed (end point = start point), the end point of the contour or polygon is not taken into account to avoid that it receives twice the weight of the other points.

moments_points_xld should be used if the contour XLD intersects itself or if it is not possible to close the contour using a line from end to start point without self-intersection, because in this case moments_xld does not produce useful results. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used.

If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD.

Parameters

- ▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
  Contours or polygons to be examined.
- ▷ **M11** (output_control) ..... real(-array) ⇝ *real*
  Mixed second order moment.
- ▷ **M20** (output_control) ..... real(-array) ⇝ *real*
  Second order moment along the row axis.
- ▷ **M02** (output_control) ..... real(-array) ⇝ *real*
  Second order moment along the column axis.

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

moments_points_xld returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, smooth_contours_xld, gen_polygons_xld

Alternatives

moments_xld, moments_any_xld, moments_any_points_xld

See also

moments_any_points_xld, area_center_points_xld, moments_region_2nd, area_center

Module

Foundation

moments_xld (XLD : : : M11, M20, M02)

Geometric moments M_{20}, M_{02}, and M_{11} of contours or polygons.

moments_xld calculates the moments (M20, M02, and M11) of the region enclosed by the contours or polygons XLD. See moments_region_2nd for the definition of these features. The moments are computed by applying Green's theorem using only the points on the contour or polygon, i.e., no region is generated explicitly for the purpose of calculating the features. It is assumed that the contours or polygons are closed. If this is not the case moments_xld will artificially close the contours or polygons.

It should be noted that moments_xld only returns useful results if the contour or polygon encloses a region in the plane. In particular, the contour or polygon must not intersect itself. This is particularly important if open contours or polygons are passed because they are closed automatically, which can produce a self-intersection. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used. If the contour or polygon intersects itself, useful values for the moments can be calculated with moments_points_xld.

If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **M11** (output_control) ..... real(-array) ⇝ *real*
Mixed second order moment.

▷ **M20** (output_control) ..... real(-array) ⇝ *real*
Second order moment along the row axis.

▷ **M02** (output_control) ..... real(-array) ⇝ *real*
Second order moment along the column axis.

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

moments_xld returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, smooth_contours_xld, gen_polygons_xld

Alternatives

moments_any_xld, moments_points_xld, moments_any_points_xld

See also

moments_any_xld, area_center_xld, moments_region_2nd, area_center

Module

Foundation

orientation_points_xld (XLD : : : Phi)

Calculate the orientation of contours or polygons treated as point clouds.

The operator orientation_points_xld calculates the orientation Phi of the point clouds given by the contours or polygons XLD (the order of the points in the contour or polygon is not taken into account). The orientation Phi is calculated the same way as in elliptic_axis_points_xld. If the contour or polygon is closed (end point = start point), the end point of the contour or polygon is not taken into account to avoid that it receives twice the weight of the other points.

In addition, the contour point p_m with maximum distance to the center of gravity c is calculated. If the angle between the vector $\overrightarrow{p_m c}$ and the vector given by Phi is greater than π, the value of π is added to the angle. If XLD consists of only two points, Phi is given by the direction from the first point towards the second point.

orientation_points_xld should be used if the contour XLD intersects itself or if it is not possible to close the contour using a line from end to start point without self-intersection, because in this case orientation_xld does not produce useful results. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used.

If more than one contour or polygon is passed, the values of the orientations are stored in a tuple in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Phi** (output_control) ..... angle.rad(-array) ⇝ *real*
Orientation of the contours or polygons (radians).
Assertion: - pi < Phi && Phi <= pi

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld

Alternatives

orientation_xld, elliptic_axis_points_xld, smallest_rectangle2_xld

See also

moments_region_2nd

Module

Foundation

orientation_xld (XLD : : : Phi)

Calculate the orientation of contours or polygons.

The operator orientation_xld calculates the orientation (Phi) of each input contour or polygon (XLD). The operator is based on elliptic_axis_xld. In addition, the contour respectively polygon point with maximum distance to the center of gravity is calculated. If the column coordinate of this point is less than the column coordinate of the center of gravity the value of π is added to the angle. It is assumed that the contours or polygons are closed. If this is not the case orientation_xld will artificially close the contours or polygons.

It should be noted that orientation_xld only returns useful results if the contour or polygon encloses a region in the plane. In particular, the contour or polygon must not intersect itself. This is particularly important if open contours or polygons are passed because they are closed automatically, which can produce a self-intersection. To test whether the contours or polygons intersect themselves, test_self_intersection_xld can be used. If the contour or polygon intersects itself, useful values for the orientation can be calculated with orientation_points_xld.

If more than one contour or polygon is passed, the values of the orientations are stored in a tuple in the same order as the respective contours or polygons in XLD.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Phi** (output_control) ..... angle.rad(-array) ⇝ *real*
Orientation of the contours or polygons (radians).
Assertion: - pi < Phi && Phi <= pi

Result

The operator orientation_xld returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contours available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld, test_self_intersection_xld

Alternatives

elliptic_axis_xld, smallest_rectangle2_xld

See also

moments_region_2nd

Module

Foundation

query_contour_attribs_xld (Contour : : : Attribs)

Return the names of the defined attributes of an XLD contour.

query_contour_attribs_xld returns the names of the defined attributes of an XLD contour Contour in Attribs. Attributes are additional values defined for each contour point, e.g., the direction perpendicular to the contour (*'angle'*). Operators which define contour point attributes contain a description of the name and semantics of the defined values. The operator get_contour_attrib_xld can be used to access the values of a particular attribute.

For an overview of the different contour point attributes and the operators that add them to XLD contours see the operator reference of get_contour_attrib_xld.

Parameters

▷ **Contour** (input_object) ..... xld_cont ⇝ *object*
Input contour.

▷ **Attribs** (output_control) ..... string-array ⇝ *string*
List of the defined contour attributes.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

lines_gauss, lines_facet, edges_sub_pix

See also

get_contour_attrib_xld, get_contour_global_attrib_xld, query_contour_global_attribs_xld

Module

Foundation

query_contour_global_attribs_xld (Contour : : : Attribs)

Return the names of the defined global attributes of an XLD contour.

query_contour_global_attribs_xld returns the names of the globally defined attributes that are set for an XLD Contour in Attribs. Global attributes are additional values defined for each contour, e.g., the normal vector of the regression line of a contour (*'regr_norm_row'* and *'regr_norm_col'*). Operators which define global attributes contain a description of the name and semantics of the defined values. The operator get_contour_global_attrib_xld can be used to access the value of a particular attribute.

For an overview of the different global contour attributes and the operators that add them to XLD contours see the operator reference of get_contour_global_attrib_xld.

Parameters

▷ **Contour** (input_object) ..... xld_cont ⇝ *object*
Input contour.

▷ **Attribs** (output_control) ..... string-array ⇝ *string*
List of the defined global contour attributes.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

lines_gauss, lines_facet, edges_sub_pix

See also

get_contour_global_attrib_xld, get_contour_attrib_xld, query_contour_attribs_xld

Module

Foundation

rectangularity_xld (XLD : : : Rectangularity)

Shape factor for the rectangularity of contours or polygons.

The operator rectangularity_xld calculates the rectangularity of each input contour or polygon in XLD. The input contour or polygon must not intersect itself, otherwise the resulting parameter is not meaningful (Whether the input contour or polygon intersects itself or not can be determined with test_self_intersection_xld). If the input contour or polygon is not closed it will be closed automatically.

To determine the rectangularity, first a rectangle is computed that has the same first and second order moments as the input contour or polygon. The computation of the rectangularity measure is finally based on the area of the difference between the computed rectangle and the input contour or polygon normalized with respect to the area of the rectangle.

For rectangles rectangularity_xld returns the value 1. The more the input contour or polygon deviates from a perfect rectangle, the less the returned value for Rectangularity will be. If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD.

Attention

For input contours or polygons which orientation cannot be computed by using second order moments (as it is the case for squares, for example), the returned Rectangularity is underestimated by up to 10% depending on the orientation of the input contour or polygon.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Rectangularity** (output_control) ..... real(-array) ⇝ *real*
Rectangularity of the input contours or polygons.
Assertion: 0 <= Rectangularity && Rectangularity <= 1.0

Result

The operator `rectangularity_xld` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input contours or polygons available) is set via the operator `set_system ('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`gen_contours_skeleton_xld`, `edges_sub_pix`, `threshold_sub_pix`, `gen_contour_polygon_xld`, `test_self_intersection_xld`

Alternatives

`circularity_xld`, `convexity_xld`, `compactness_xld`, `eccentricity_xld`

See also

`area_center_xld`, `select_shape_xld`

References

P. L. Rosin: “Measuring rectangularity”; Machine Vision and Applications; vol. 11; pp. 191-196; Springer-Verlag, 1999.

Module

Foundation

select_contours_xld (Contours : SelectedContours : Feature, Min1, Max1, Min2, Max2 :)

Select XLD contours according to several features.

`select_contours_xld` selects XLD contours from the input contours `Contours` according to the following features (parameter `Feature`):

'contour_length': all contours whose length is smaller than `Min1` or larger than `Max1` are not returned (`Min2` and `Max2` have no influence here).

'maximum_extent': all contours whose maximum extent (as measured by their eight extremal points in row and column direction, according to Haralick and Shapiro: Computer and Robot Vision, Addison-Wesley 1992, chapter 3.2) is smaller than `Min1` or larger than `Max1` are not returned (`Min2` and `Max2` have no influence here).

'direction': only contours for which the direction of the regression line is between `Min1` and `Max1` (in radians, counter-clockwise) are returned. `Min1` and `Max1` are mapped to the range of [0,2*PI[. (`Min2` and `Max2` have no influence here).

'curvature': only contours for which the mean distance from the regression line lies between `Min1` and `Max1`, and for which the standard deviation of the distances is between `Min2` and `Max2` are returned.

'closed': only contours for which the distance between their start point and their end point is less or equal `Max1` pixels are returned. (`Min1`, `Min2` and `Max2` have no influence here.)

'open': only contours for which the distance between their start point and their end point is greater than `Min1` pixels are returned. (`Max1`, `Min2` and `Max2` have no influence here).

If Min1 = Max1 = 0 or Min2 = Max2 = 0 is used for the selection according to curvature, the respective feature has no influence on the selection.

Attention

Before contour can be filtered by select_contours_xld according to *'direction'* or *'curvature'*, the parameters of the regression lines to the contours must be calculated with regress_contours_xld. If this has not been done, select_contours_xld calls regress_contours_xld internally with the parameters Mode = *'no'* and Iterations = *1*. If a different mode should be used, regress_contours_xld must be called explicitly.

Parameters

- ▷ **Contours** (input_object) ..... xld_cont-array ⇝ *object*
  Input XLD contours.
- ▷ **SelectedContours** (output_object) ..... xld_cont-array ⇝ *object*
  Output XLD contours.
- ▷ **Feature** (input_control) ..... string ⇝ *string*
  Feature to select contours with.
  Default: 'contour_length'
  List of values: Feature ∈ {'contour_length', 'maximum_extent', 'direction', 'curvature', 'closed', 'open'}
- ▷ **Min1** (input_control) ..... real ⇝ *real*
  Lower threshold.
  Default: 0.5
- ▷ **Max1** (input_control) ..... real ⇝ *real*
  Upper threshold.
  Default: 200.0
- ▷ **Min2** (input_control) ..... real ⇝ *real*
  Lower threshold.
  Default: -0.5
- ▷ **Max2** (input_control) ..... real ⇝ *real*
  Upper threshold.
  Default: 0.5

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

regress_contours_xld

See also

get_contour_xld, get_contour_attrib_xld, gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix, get_regress_params_xld, get_contour_global_attrib_xld, query_contour_global_attribs_xld

References

R. Haralick, L. Shapiro: "Computer and Robot Vision" Vol. 1; Kapitel 3.2, Addison-Wesley 1992

Module

Foundation

select_shape_xld (XLD : SelectedXLD : Features, Operation, Min, Max :)

Select contours or polygons using shape features.

The operator select_shape_xld selects contours or polygons according to their shape. For each input contour or polygon in XLD the indicated features (Features) are calculated. If each (Operation = *'and'*) or at least one (Operation = *'or'*) of the calculated features is within the limits (Min,Max) the contour respectively polygon is

copied into the output. The parameters Min and Max can be set to *'min'* or *'max'* in order to leave bottom and top limits, respectively, open.

Condition:

$$\texttt{Min}_i \leq \texttt{Features}_i(Object) \leq \texttt{Max}_i$$

Possible values for Features:

(all features can be used with polygons as well)

'area': Area of the contour (see area_center_xld)

'area_points': Area of the contour points (see area_center_points_xld)

'row': Row index of the center of gravity (see area_center_xld)

'column': Column index of the center of gravity (see area_center_xld)

'row_points': Row index of the center of gravity of the contour points (see area_center_points_xld)

'column_points': Column index of the center of gravity of the contour points (see area_center_points_xld)

'width': Width of the contour (see height_width_ratio_xld)

'height': Height of the contour (see height_width_ratio_xld)

'ratio': Ratio of the height and the width of the contour (see height_width_ratio_xld)

'row1': Row coordinate of upper left corner (see smallest_rectangle1_xld)

'column1': Column coordinate of upper left corner (see smallest_rectangle1_xld)

'row2': Row coordinate of lower right corner (see smallest_rectangle1_xld)

'column2': Column coordinate of lower right corner (see smallest_rectangle1_xld)

'circularity': Circularity (see circularity_xld)

'compactness': Compactness (see compactness_xld)

'rectangularity': Rectangularity (see rectangularity_xld)

'contlength': Total length of contour (see length_xld)

'convexity': Convexity (see convexity_xld)

'ra': Major radius of the equivalent ellipse (see elliptic_axis_xld)

'rb': Minor radius of the equivalent ellipse (see elliptic_axis_xld)

'phi': Orientation of the equivalent ellipse (see elliptic_axis_xld)

'ra_points': Major radius of the equivalent ellipse of the contour points (see elliptic_axis_points_xld)

'rb_points': Minor radius of the equivalent ellipse of the contour points (see elliptic_axis_points_xld)

'phi_points': Orientation of the equivalent ellipse of the contour points (see elliptic_axis_points_xld)

'anisometry:' Anisometry (see eccentricity_xld)

'anisometry_points:' Anisometry of the contour points (see eccentricity_points_xld)

'bulkiness:' Bulkiness (see eccentricity_xld)

'struct_factor:' Structure factor (see eccentricity_xld)

'outer_radius': Radius of smallest enclosing circle (see smallest_circle_xld)

'max_diameter': Maximum diameter of the contour (see diameter_xld)

'orientation': Orientation of the contour (see orientation_xld)

'orientation_points': Orientation of the contour points (see orientation_points_xld)

'rect2_phi': Orientation of the smallest surrounding rectangle (see smallest_rectangle2_xld)

'rect2_len1': Half the length of the smallest surrounding rectangle (see smallest_rectangle2_xld)

'rect2_len2': Half the width of the smallest surrounding rectangle (see smallest_rectangle2_xld)

'moments_m11', 'moments_m20', 'moments_m02': Geometric moments of the contour (see moments_region_2nd)

'moments_m11_points', 'moments_m20_points', 'moments_m02_points': Geometric moments of the contour points (see `moments_points_xld`)

If only one feature (`Features`) is used the value of `Operation` is meaningless. Several features are processed in the sequence in which they are entered.The use of some features requires that the input contour respectively polygon must not intersect itself, otherwise the results are not meaningful (Whether the input contour or polygon intersects itself or not can be determined with `test_self_intersection_xld`).

Parameters

▷ **`XLD`** (input_object) . . . . . xld-array ⇝ *object*
Contours or polygons to be examined.

▷ **`SelectedXLD`** (output_object) . . . . . xld-array ⇝ *object*
Contours or polygons fulfilling the condition(s).

▷ **`Features`** (input_control) . . . . . string(-array) ⇝ *string*
Shape features to be checked.
Default: 'area'
List of values: `Features` ∈ {'area', 'area_points', 'row', 'row_points', 'column', 'column_points', 'width', 'height', 'ratio', 'row1', 'column1', 'row2', 'column2', 'circularity', 'compactness', 'contlength', 'convexity', 'rectangularity', 'ra', 'ra_points', 'rb', 'rb_points', 'phi', 'phi_points', 'anisometry', 'anisometry_points', 'bulkiness', 'struct_factor', 'outer_radius', 'max_diameter', 'orientation', 'orientation_points', 'rect2_phi', 'rect2_len1', 'rect2_len2', 'moments_m11', 'moments_m11_points', 'moments_m20', 'moments_m20_points', 'moments_m02', 'moments_m02_points'}

▷ **`Operation`** (input_control) . . . . . string ⇝ *string*
Operation type between the individual features.
Default: 'and'
List of values: `Operation` ∈ {'and', 'or'}

▷ **`Min`** (input_control) . . . . . real(-array) ⇝ *real* / integer / string
Lower limits of the features or 'min'.
Default: 150.0
Minimum increment: 0.001
Recommended increment: 1.0

▷ **`Max`** (input_control) . . . . . real(-array) ⇝ *real* / integer / string
Upper limits of the features or 'max'.
Default: 99999.0
Minimum increment: 0.001
Recommended increment: 1.0
Restriction: Max >= Min

Result

The operator `select_shape_xld` returns the value `2` (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input objects available) is set via the operator `set_system ('no_object_result',<Result>)`. If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_contours_skeleton_xld`, `edges_sub_pix`, `threshold_sub_pix`, `gen_contour_polygon_xld`, `test_self_intersection_xld`

Possible Successors

`shape_trans_xld`, `count_obj`

See also

`area_center_xld`, `area_center_points_xld`, `circularity_xld`, `compactness_xld`, `convexity_xld`, `elliptic_axis_xld`, `elliptic_axis_points_xld`, `eccentricity_xld`, `eccentricity_points_xld`, `rectangularity_xld`, `smallest_circle_xld`,

smallest_rectangle1_xld, smallest_rectangle2_xld, diameter_xld, orientation_xld, orientation_points_xld, moments_xld, moments_points_xld, select_obj

Module

Foundation

select_xld_point (XLDs : DestXLDs : Row, Column :)

Choose all contours or polygons containing a given point.

The operator select_xld_point selects all contours or polygons from XLDs containing the test point (Row,Column). If the contours or polygons overlap, more than one contour or polygon might contain the point. In this case, all these contours or polygons are returned. If no contour or polygon contains the indicated point, an empty object is returned. It is assumed that the contours or polygons are closed. If this is not the case, select_xld_point will artificially close the contours or polygons. Nevertheless, it is strongly recommended to pass only meaningful and closed contours or polygons in XLDs by calling select_contours_xld with the parameter *'closed'* before calling select_xld_point.

Attention

If the test point is on the border of a contour or polygon, select_xld_point will deliver unpredictable results, i.e., the contour or polygon may be or may not be added to DestXLDs, depending on arbitrary factors such as how the polygon is oriented with respect to the coordinate system. Note further that for points in the near proximity of the contour or polygon sides, a reliable classification is practically impossible because of numerical inaccuracies.

Parameters

- ▷ **XLDs** (input_object) ..... xld(-array) ⇝ *object*
  Contours or polygons to be examined.
- ▷ **DestXLDs** (output_object) ..... xld(-array) ⇝ *object*
  All contours or polygons containing the test point.
- ▷ **Row** (input_control) ..... point.y ⇝ *real* / integer
  Line coordinate of the test point.
  Default: 100.0
- ▷ **Column** (input_control) ..... point.x ⇝ *real* / integer
  Column coordinate of the test point.
  Default: 100.0

Result

The operator select_xld_point returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input contours or polygons available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

select_contours_xld, close_contours_xld, threshold_sub_pix

Alternatives

test_xld_point

Module

Foundation

smallest_circle_xld (XLD : : : Row, Column, Radius)

Smallest enclosing circle of contours or polygons.

The operator smallest_circle_xld determines the smallest enclosing circle of contours or polygons, i.e., the circle with the smallest area of all circles containing the contour. For this circle the center (Row,Column) and the radius (Radius) are calculated.

If several contours or polygons are passed in XLD corresponding tuples are returned as output parameter. In case of empty contours all parameters have the value 0.0 if no other behavior was set (see set_system).

Parameters

- ▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
  Contours or polygons to be examined.
- ▷ **Row** (output_control) ..... circle.center.y(-array) ⇝ *real*
  Row coordinate of the center of the enclosing circle.
- ▷ **Column** (output_control) ..... circle.center.x(-array) ⇝ *real*
  Column coordinate of the center of the enclosing circle.
- ▷ **Radius** (output_control) ..... circle.radius(-array) ⇝ *real*
  Radius of the enclosing circle.
  Assertion: Radius >= 0

Complexity

If N is the number of contour points then the runtime complexity is $O(N * ln(N))$.

Result

The operator smallest_circle_xld returns the value 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behavior can be set via set_system(::'no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld

Possible Successors

gen_ellipse_contour_xld

Alternatives

smallest_rectangle1_xld, smallest_rectangle2_xld

See also

elliptic_axis_xld, smallest_rectangle1_xld, smallest_rectangle2_xld, shape_trans_xld

Module

Foundation

smallest_rectangle1_xld (XLD : : : Row1, Column1, Row2, Column2)

Enclosing rectangle parallel to the coordinate axes of contours or polygons.

The operator smallest_rectangle1_xld calculates the enclosing rectangle (parallel to the coordinate axes) for each input contour or polygon. The enclosing rectangle is described by the coordinates of the corner pixels (Row1,Column1,Row2,Column2)

If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in XLD. In case of an empty contour all parameters have the value 0 if no other behavior was set (see set_system).

Attention

In case of empty contours the result of Row1,Column1, Row2 and Column2 (all are 0) can lead to confusion.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Row1** (output_control) ..... rectangle.origin.y(-array) ⇝ *real*
Row coordinate of upper left corner point of the enclosing rectangle.

▷ **Column1** (output_control) ..... rectangle.origin.x(-array) ⇝ *real*
Column coordinate of upper left corner point of the enclosing rectangle.

▷ **Row2** (output_control) ..... rectangle.corner.y(-array) ⇝ *real*
Row coordinate of lower right corner point of the enclosing rectangle.

▷ **Column2** (output_control) ..... rectangle.corner.x(-array) ⇝ *real*
Column coordinate of lower right corner point of the enclosing rectangle.

Complexity

If N is the number of contour points, the runtime complexity is $O(N)$.

Result

`smallest_rectangle1_xld` returns `2` (`H_MSG_TRUE`) if the input is not empty. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`gen_contours_skeleton_xld`, `edges_sub_pix`, `threshold_sub_pix`, `gen_contour_polygon_xld`

Possible Successors

`gen_polygons_xld`

Alternatives

`height_width_ratio_xld`, `smallest_rectangle2_xld`, `shape_trans_xld`

See also

`shape_trans_xld`, `smallest_rectangle2_xld`, `smallest_circle_xld`, `elliptic_axis_xld`, `area_center_xld`

Module

Foundation

smallest_rectangle2_xld (XLD : : : Row, Column, Phi, Length1, Length2)

Smallest enclosing rectangle with arbitrary orientation of contours or polygons.

The operator `smallest_rectangle2_xld` determines the smallest enclosing rectangle of each input contour or polygon, i.e., the rectangle with the smallest area of all rectangles containing the contour. For this rectangle the center, the inclination, and the two radii are calculated.

If more than one contour or polygon is passed, the results are stored in tuples in the same order as the respective contours or polygons in `XLD`. In case of an empty contour all parameters have the value 0.0 if no other behavior was set (see `set_system`).

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be examined.

▷ **Row** (output_control) ..... rectangle2.center.y(-array) ⇝ *real*
Row coordinate of the center point of the enclosing rectangle.

▷ **Column** (output_control) . . . . . rectangle2.center.x(-array) ⇝ *real*
Column coordinate of the center point of the enclosing rectangle.

▷ **Phi** (output_control) . . . . . rectangle2.angle.rad(-array) ⇝ *real*
Orientation of the enclosing rectangle (arc measure)
Assertion: - pi / 2 < Phi && Phi <= pi / 2

▷ **Length1** (output_control) . . . . . rectangle2.hwidth(-array) ⇝ *real*
First radius (half length) of the enclosing rectangle.
Assertion: Length1 >= 0.0

▷ **Length2** (output_control) . . . . . rectangle2.hheight(-array) ⇝ *real*
Second radius (half width) of the enclosing rectangle.
Assertion: Length2 >= 0.0 && Length2 <= Length1

Complexity

If N is the number of contour points and C is the number of points in the convex hull, the runtime complexity is $O(N * ln(N) + C^2)$.

Result

`smallest_rectangle2_xld` returns `2` (`H_MSG_TRUE`) if the input is not empty. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`gen_contours_skeleton_xld`, `edges_sub_pix`, `threshold_sub_pix`, `gen_contour_polygon_xld`

Possible Successors

`gen_polygons_xld`

Alternatives

`smallest_rectangle1`, `shape_trans_xld`

See also

`smallest_rectangle1`, `smallest_circle`, `elliptic_axis_xld`

Module

Foundation

test_closed_xld (XLD : : : IsClosed)

Test whether contours or polygons are closed.

`test_closed_xld` tests each contour or polygon in `XLD`, whether it is closed and returns the result in `IsClosed`. If the contour contour or polygon is closed, the respective return value is 1, otherwise 0. A contour or polygon is closed, if its last point is equal to its first point.

Parameters

▷ **XLD** (input_object) . . . . . xld(-array) ⇝ *object*
Contours or polygons to be tested.

▷ **IsClosed** (output_control) . . . . . integer(-array) ⇝ *integer*
Tuple with Boolean numbers.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).

- Processed without parallelization.

Possible Predecessors

gen_contour_region_xld, edges_sub_pix

Module

Foundation

test_self_intersection_xld (XLD : : CloseXLD : DoesIntersect)

Test XLD contours or polygons for self intersection.

test_self_intersection_xld tests whether the input contours or polygons in XLD intersect or touch themselves. The output array DoesIntersect contains a *1* if the corresponding input contour or polygon intersects or touches itself and *0* otherwise.

If the input parameter CloseXLD is set to *'true'*, open contours or polygons are closed before the check is performed by simply connecting the first and the last contour or polygon point with a line segment.

Attention

A contour or polygon touches itself if a contour or polygon point lies exactly on a contour or polygon segment. However, whether a point lies exactly on a contour or polygon segment or not can practically not be decided because of numerical reasons.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Input contours or polygons.

▷ **CloseXLD** (input_control) ..... string ⇝ *string*
Should the input contours or polygons be closed first?
Default: 'true'
List of values: CloseXLD ∈ {'true', 'false'}

▷ **DoesIntersect** (output_control) ..... number(-array) ⇝ *integer*
1 for contours or polygons with self intersection and 0 otherwise.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Module

Foundation

test_xld_point (XLD : : Row, Column : IsInside)

Test whether one or more contours or polygons enclose the given point(s).

test_xld_point tests whether a test point (Row, Column) is enclosed by a contour or polygon XLD and returns the result in IsInside. If the input point is enclosed by the contour or polygon, the return value is *1*, otherwise *0*. In the case of a single test point and multiple contours or polygons, the (i-1)-th element of the resulting tuple IsInside indicates, whether the point is enclosed by the i-th contour or polygon. In the case of multiple test points and a single contour or polygon, the i-th element of the resulting tuple IsInside indicates, whether the i-th point is enclosed by the contour or polygon. In the case of multiple test points and multiple contours or polygons, the (i-1)-th element of the resulting tuple IsInside indicates, whether the (i-1)-th point is enclosed by the i-th contour or polygon (the subscription differs, because the indexing of the results tuple starts with 0 in contrast to the indexing of object tuples). In this case the number of input points must be equal to the number of contours or polygons. It is assumed that the contours or polygons are closed. If this is not the case test_xld_point will artificially close the contour. Nevertheless it is strongly recommended to pass only meaningful and closed contours or polygons in XLD.

Attention

If a test point is on the border of the contour or polygon, `test_xld_point` will deliver unpredictable results, i.e. the result may be *0* or *1* depending on arbitrary factors such as how the polygon is oriented with respect to the coordinate system. Note further that for points in the near proximity of the contour or polygon sides, a reliable classification is practically impossible because of numerical inaccuracies.

Parameters

- ▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
  Contours or polygons to be tested.
- ▷ **Row** (input_control) ..... point.y(-array) ⇝ *real*
  Row coordinates of the points to be tested.
- ▷ **Column** (input_control) ..... point.x(-array) ⇝ *real*
  Column coordinates of the points to be tested.
  Number of elements: Column == Row
- ▷ **IsInside** (output_control) ..... integer(-array) ⇝ *integer*
  Tuple with Boolean numbers.

Result

`test_xld_point` returns `2` (`H_MSG_TRUE`) if the input is not empty. If the input is empty the behavior can be set via `set_system('no_object_result',<Result>:)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`select_contours_xld`, `close_contours_xld`, `threshold_sub_pix`

Module

Foundation

29.4 Geometric Transformations

affine_trans_contour_xld (
Contours : ContoursAffineTrans : HomMat2D :)

Apply an arbitrary affine 2D transformation to XLD contours.

`affine_trans_contour_xld` applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the XLD contours given in `Contours` and returns the transformed contours in `ContoursAffineTrans`. The affine transformation is described by the homogeneous transformation matrix given in `HomMat2D`, which can be created using the operators `hom_mat2d_identity`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_translate`, etc., or be the result of operators like `vector_angle_to_rigid`.

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to x and the *column* coordinate corresponds to y of the coordinate system in which the transformation matrix was defined. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

Attention

`affine_trans_contour_xld` does not use the HALCON standard coordinate system (with the origin in the center of the upper left pixel), but instead uses the same coordinate system as in `affine_trans_pixel`, i.e., the origin lies in the upper left corner of the upper left pixel. Therefore, applying `affine_trans_contour_xld` corresponds to a chain of transformations (see `affine_trans_pixel`), which is applied to each point of the contour (input and output pixels as homogeneous vectors). As an effect, you might get unexpected results when creating affine transformations based on coordinates that are derived from the contour, e.g., by operators like

area_center_xld. For example, if you use this operator to calculate the center of gravity of a rotationally symmetric XLD contour and then rotate the contour around this point using hom_mat2d_rotate, the resulting contour will not lie on the original one. In such a case, you can compensate this effect by applying the following translations to HomMat2D before using it in affine_trans_contour_xld:

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
affine_trans_contour_xld(Contours, ContoursAffineTrans,
HomMat2DAdapted)
```

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

- ▷ **Contours** (input_object) ..... xld_cont(-array) ⇝ *object*
  Input XLD contours.
- ▷ **ContoursAffineTrans** (output_object) ..... xld_cont(-array) ⇝ *object*
  Transformed XLD contours.
- ▷ **HomMat2D** (input_control) ..... hom_mat2d ⇝ *real*
  Input transformation matrix.

Result

If the matrix HomMat2D represents an affine transformation (i.e., not a projective transformation), affine_trans_contour_xld returns 2 (H_MSG_TRUE). If the input is empty the behavior can be set via set_system('no_object_result',<Result>:). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

hom_mat2d_identity, hom_mat2d_translate, hom_mat2d_rotate, hom_mat2d_scale, hom_mat2d_reflect

See also

affine_trans_image, affine_trans_region

Module

Foundation

affine_trans_polygon_xld (Polygons : PolygonsAffineTrans : HomMat2D :)

Apply an arbitrary affine transformation to XLD polygons.

affine_trans_polygon_xld applies an arbitrary affine transformation, i.e., scaling, rotation, translation, and slant (skewing), to the XLD polygons given in Polygons and returns the transformed polygons in PolygonsAffineTrans. The affine transformation is described by the homogeneous transformation matrix given in HomMat2D. This matrix can be created using the operators hom_mat2d_identity, hom_mat2d_scale, hom_mat2d_rotate, hom_mat2d_translate, etc., or be the result of operators like vector_angle_to_rigid.

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to x and the *column* coordinate corresponds to y of the coordinate system in which the transformation matrix was defined. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

Attention

The XLD contours that are possibly referenced by `Polygons` are neither transformed nor stored with the output polygons, since this is generally impossible without creating inconsistencies for the attributes of the XLD contours. Hence, operators that access the contours associated with a polygon, e.g., `split_contours_xld` will not work correctly.

`affine_trans_polygon_xld` does not use the HALCON standard coordinate system (with the origin in the center of the upper left pixel), but instead uses the same coordinate system as in `affine_trans_pixel`, i.e., the origin lies in the upper left corner of the upper left pixel. Therefore, applying `affine_trans_polygon_xld` corresponds to a chain of transformations (see `affine_trans_pixel`), which is applied to each point of the polygon (input and output pixels as homogeneous vectors). As an effect, you might get unexpected results when creating affine transformations based on coordinates that are derived from the polygon, e.g., by operators like `area_center_xld`. For example, if you use this operator to calculate the center of gravity of a rotationally symmetric XLD polygon and then rotate the polygon around this point using `hom_mat2d_rotate`, the resulting polygon will not lie on the original one. In such a case, you can compensate this effect by applying the following translations to `HomMat2D` before using it in `affine_trans_polygon_xld`:

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
affine_trans_polygon_xld(Polygons, PolygonsAffineTrans,
HomMat2DAdapted)
```

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

- ▷ **`Polygons`** (input_object) ..... xld_poly(-array) ⇝ *object*
  Input XLD polygons.
- ▷ **`PolygonsAffineTrans`** (output_object) ..... xld_poly(-array) ⇝ *object*
  Transformed XLD polygons.
- ▷ **`HomMat2D`** (input_control) ..... hom_mat2d ⇝ *real*
  Input transformation matrix.

Result

If the matrix `HomMat2D` represents an affine transformation (i.e., not a projective transformation), `affine_trans_polygon_xld` returns `2` (`H_MSG_TRUE`). If the input is empty the behavior can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

`hom_mat2d_identity`, `hom_mat2d_translate`, `hom_mat2d_rotate`, `hom_mat2d_scale`, `hom_mat2d_reflect`

See also

`affine_trans_image`, `affine_trans_region`, `affine_trans_contour_xld`

Module

Foundation

`gen_parallel_contour_xld` (Contours : ParallelContours : Mode, Distance :)

Compute the parallel contour of an XLD contour.

gen_parallel_contour_xld computes for each of the input contours Contours a parallel contour with distance Distance. The resulting contours are returned in ParallelContours. To calculate the parallel contour, the normal vector of the input contour is needed in every contour point. The parameter Mode determines how these normal vectors are computed. If Mode = *'gradient'*, it is assumed that the input contours are edges, and the normal information is obtained from the gradient direction of the edge (see edges_sub_pix). For this, the attribute *'edge_direction'* must exist for the input contour (see get_contour_attrib_xld). If Mode = *'contour_normal'*, a possibly existing normal information is used to determine the normals. For this, the contour attribute *'angle'* must exist (see lines_gauss or edges_sub_pix). Finally, if Mode = *'regression_normal'*, the normal vectors are determined from a local line fit to each contour point. Here, the normal vectors are oriented such that they point to the right side of the contour when the contour is traversed from start to end. In contrast to the first two modes, this mode can be used for all XLD contours, no matter how they were generated.

Parameters

▷ **Contours** (input_object) . . . . . xld_cont-array ⇝ *object*
Contours to be transformed.

▷ **ParallelContours** (output_object) . . . . . xld_cont-array ⇝ *object*
Parallel contours.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Mode, with which the direction information is computed.
Default: 'regression_normal'
Suggested values: Mode ∈ {'gradient', 'contour_normal', 'regression_normal'}

▷ **Distance** (input_control) . . . . . number ⇝ *real* / integer
Distance of the parallel contour.
Default: 1
Suggested values: Distance ∈ {0.2, 0.4, 0.6, 0.8, 1, 2, 3, 4, 5, 7, 10, 15, 20, 30, 40, 50}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix, threshold_sub_pix

Possible Successors

gen_polygons_xld

See also

get_contour_xld

Module

Foundation

polar_trans_contour_xld (Contour : PolarTransContour : Row, Column, AngleStart, AngleEnd, RadiusStart, RadiusEnd, Width, Height :)

Transform a contour in an annular arc to polar coordinates.

polar_trans_contour_xld transforms the Contour in the annular arc specified by the center point (Row, Column), the radii RadiusStart and RadiusEnd and the angles AngleStart and AngleEnd to its polar coordinate version PolarTransContour in the virtual image of the dimensions Width × Height. The output contour is cropped at the borders of this virtual image.

The coordinate (0,0) in the output contour always corresponds to the contour point specified by RadiusStart and AngleStart. Analogously, the coordinate (Height − 1,Width − 1) in the output contour always corresponds to the point in the contour that is specified by RadiusEnd and AngleEnd, even if the contour does not contain these points. In the usual mode (AngleStart < AngleEnd and RadiusStart < RadiusEnd),

the polar transformation is performed in the mathematically positive orientation (counterclockwise). Furthermore, contour points with smaller radii lie in the upper part of the output contour. By suitably exchanging the values of these parameters (e.g., AngleStart > AngleEnd or RadiusStart > RadiusEnd), any desired orientation of the output contour PolarTransContour can be achieved.

The angles can be chosen from all real numbers. Center point and radii can be real as well.

Note that PolarTransContour can consist of more than one contour because Contour may be split at AngleStart respectively AngleEnd or the size of the angle interval [AngleStart, AngleEnd] may be greater than one $2pi$ period.

If more than one contour is passed in Contour, their polar transformations are computed individually and stored as a tuple in PolarTransContour. However, since one contour may be transformed into several contours, there is no relation between the indices of the contours in the input tuple Contour and the indices in the output tuple PolarTransContour.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Attention

Only the contour points are transformed. As the polar transformation is not affine, polar_trans_contour_xld only produces reliable results if the spacing of the contour points is small. Existing attributes are not transformed.

Parameters

▷ **Contour** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contour.

▷ **PolarTransContour** (output_object) ..... xld_cont(-array) ⇝ *object*
Output contour.

▷ **Row** (input_control) ..... number ⇝ *real* / integer
Row coordinate of the center of the arc.
Default: 256
Suggested values: Row ∈ {0, 16, 32, 64, 128, 240, 256, 480, 512}

▷ **Column** (input_control) ..... number ⇝ *real* / integer
Column coordinate of the center of the arc.
Default: 256
Suggested values: Column ∈ {0, 16, 32, 64, 128, 256, 320, 512, 640}

▷ **AngleStart** (input_control) ..... angle.rad ⇝ *real*
Angle of the ray to be mapped to the column coordinate *0* of PolarTransContour.
Default: 0.0
Suggested values: AngleStart ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853, 12.566370616}
Value range: -6.2831853 ≤ AngleStart ≤ 6.2831853

▷ **AngleEnd** (input_control) ..... angle.rad ⇝ *real*
Angle of the ray to be mapped to the column coordinate Width − 1 of PolarTransContour to.
Default: 6.2831853
Suggested values: AngleEnd ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853, 12.566370616}
Value range: -6.2831853 ≤ AngleEnd ≤ 6.2831853

▷ **RadiusStart** (input_control) ..... number ⇝ *real* / integer
Radius of the circle to be mapped to the row coordinate *0* of PolarTransContour.
Default: 0
Suggested values: RadiusStart ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ RadiusStart ≤ 32767

▷ **RadiusEnd** (input_control) ..... number ⇝ *real* / integer
Radius of the circle to be mapped to the row coordinate Height − 1 of PolarTransContour.
Default: 100
Suggested values: RadiusEnd ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ RadiusEnd ≤ 32767

▷ **Width** (input_control) . . . . . extent.x ⇝ *integer*
Width of the virtual output image.
Default: 512
Suggested values: Width ∈ {256, 320, 512, 640, 800, 1024}
Value range: 0 ≤ Width ≤ 32767

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the virtual output image.
Default: 512
Suggested values: Height ∈ {240, 256, 480, 512, 600, 1024}
Value range: 0 ≤ Height ≤ 32767

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

See also

polar_trans_image_ext, polar_trans_image_inv, polar_trans_region, polar_trans_region_inv, polar_trans_contour_xld_inv

Module

Foundation

polar_trans_contour_xld_inv (PolarContour : XYTransContour : Row, Column, AngleStart, AngleEnd, RadiusStart, RadiusEnd, WidthIn, HeightIn, Width, Height :)

Transform a contour in polar coordinates back to Cartesian coordinates

polar_trans_contour_xld_inv transforms the polar coordinate representation of a contour, stored in PolarContour, back onto an annular arc in Cartesian coordinates, described by the radii RadiusStart and RadiusEnd and the angles AngleStart and AngleEnd with the center point located at (Row, Column). All of these values can be chosen as real numbers. In addition, the dimensions of the virtual image containing the contour PolarContour must be given in WidthIn and HeightIn. WidthIn − 1 is the column coordinate corresponding to AngleEnd and HeightIn − 1 is the row coordinate corresponding to RadiusEnd. AngleStart and RadiusStart correspond to column and row coordinate *0*. Furthermore, the dimensions Width and Height of the virtual output image containing the transformed contour XYTransContour are required. The output contour is cropped at the borders of this virtual output image.

polar_trans_contour_xld_inv is the inverse function of polar_trans_contour_xld.

The call sequence:

```
polar_trans_contour_xld(Contour, PolarContour, Row, Column,
rad(360)
polar_trans_contour_xld_inv(PolarContour, XYTransContour,
Row, Column, rad(360), 0, 0, Radius, Width, Height, WidthOut,
HeightOut)
```

returns the contour Contour, restricted to the circle around (Row, Column) with radius Radius, as its output contour XYTransContour.

Note that XYTransContour can consist of more than one contour because PolarContour may be cropped at the borders of the virtual output image given by Width and Height.

If more than one contour is passed in PolarContour, their transformations are computed individually and stored as a tuple in XYTransContour. However, since one contour may be transformed into several contours, there is no relation between the indices of the contours in the input tuple PolarContour and the indices in the output tuple XYTransContour.

Further Information

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Attention

Only the contour points are transformed. As the polar transformation is not affine, `polar_trans_contour_xld_inv` only produces reliable results if the spacing of the contour points is small. Existing attributes are not transformed.

Parameters

▷ **`PolarContour`** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contour.

▷ **`XYTransContour`** (output_object) ..... xld_cont(-array) ⇝ *object*
Output contour.

▷ **`Row`** (input_control) ..... number ⇝ *real* / integer
Row coordinate of the center of the arc.
Default: 256
Suggested values: `Row` ∈ {0, 16, 32, 64, 128, 240, 256, 480, 512}

▷ **`Column`** (input_control) ..... number ⇝ *real* / integer
Column coordinate of the center of the arc.
Default: 256
Suggested values: `Column` ∈ {0, 16, 32, 64, 128, 256, 320, 512, 640}

▷ **`AngleStart`** (input_control) ..... angle.rad ⇝ *real*
Angle of the ray to map the column coordinate *0* of `PolarContour` to.
Default: 0.0
Suggested values: `AngleStart` ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853}
Value range: -6.2831853 ≤ `AngleStart` ≤ 6.2831853

▷ **`AngleEnd`** (input_control) ..... angle.rad ⇝ *real*
Angle of the ray to map the column coordinate `WidthIn` − 1 of `PolarContour` to.
Default: 6.2831853
Suggested values: `AngleEnd` ∈ {0.0, 0.78539816, 1.57079632, 3.141592654, 6.2831853}
Value range: -6.2831853 ≤ `AngleEnd` ≤ 6.2831853

▷ **`RadiusStart`** (input_control) ..... number ⇝ *real* / integer
Radius of the circle to map the row coordinate *0* of `PolarContour` to.
Default: 0
Suggested values: `RadiusStart` ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ `RadiusStart` ≤ 32767

▷ **`RadiusEnd`** (input_control) ..... number ⇝ *real* / integer
Radius of the circle to map the row coordinate `HeightIn` − 1 of `PolarContour` to.
Default: 100
Suggested values: `RadiusEnd` ∈ {0, 16, 32, 64, 100, 128, 256, 512}
Value range: 0 ≤ `RadiusEnd` ≤ 32767

▷ **`WidthIn`** (input_control) ..... extent.x ⇝ *integer*
Width of the virtual input image.
Default: 512
Suggested values: `WidthIn` ∈ {256, 320, 512, 640, 800, 1024}
Value range: 0 ≤ `WidthIn` ≤ 32767

▷ **`HeightIn`** (input_control) ..... extent.y ⇝ *integer*
Height of the virtual input image.
Default: 512
Suggested values: `HeightIn` ∈ {240, 256, 480, 512, 600, 1024}
Value range: 0 ≤ `HeightIn` ≤ 32767

▷ **`Width`** (input_control) ..... extent.x ⇝ *integer*
Width of the virtual output image.
Default: 512
Suggested values: `Width` ∈ {256, 320, 512, 640, 800, 1024}
Value range: 0 ≤ `Width` ≤ 32767

▷ **Height** (input_control) . . . . . extent.y ⇝ *integer*
Height of the virtual output image.
Default: 512
Suggested values: Height ∈ {240, 256, 480, 512, 600, 1024}
Value range: 0 ≤ Height ≤ 32767

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

See also

polar_trans_image_ext, polar_trans_image_inv, polar_trans_region, polar_trans_region_inv, polar_trans_contour_xld

Module

Foundation

projective_trans_contour_xld (Contours : ContoursProjTrans : HomMat2D :)

Apply a projective transformation to an XLD contour.

projective_trans_contour_xld applies the projective transformation specified by the homogeneous matrix HomMat2D on the contours in Contours and returns the transformed contours in ContoursProjTrans.

For creation and interpretation details of this matrix see also projective_trans_image.

Attention

The used coordinate system is the same as in affine_trans_pixel. This means that in fact not HomMat2D is applied but a modified version. Therefore, applying projective_trans_contour_xld corresponds to the following chain of transformations, which is applied to each point (Row_i, Col_i) of the contour (input and output pixels as homogeneous vectors):

$$\begin{pmatrix} RowTrans_i \\ ColTrans_i \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & -0.5 \\ 0 & 1 & -0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D} \cdot \begin{bmatrix} 1 & 0 & +0.5 \\ 0 & 1 & +0.5 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} Row_i \\ Col_i \\ 1 \end{pmatrix}$$

As an effect, you might get unexpected results when creating projective transformations based on coordinates that are derived from the contour, e.g., by operators like area_center_xld. For example, if you use this operator to calculate the center of gravity of a rotationally symmetric XLD contour and then rotate the contour around this point using hom_mat2d_rotate, the resulting contour will not lie on the original one. In such a case, you can compensate this effect by applying the following translations to HomMat2D before using it in projective_trans_contour_xld:

```
hom_mat2d_translate(HomMat2D, 0.5, 0.5, HomMat2DTmp)
hom_mat2d_translate_local(HomMat2DTmp, -0.5, -0.5,
HomMat2DAdapted)
projective_trans_contour_xld(Contours, ContoursAffineTrans,
HomMat2DAdapted)
```

For an explanation of the different 2D coordinate systems used in HALCON, see the introduction of chapter Transformations / 2D Transformations.

Parameters

▷ **Contours** (input_object) . . . . . xld_cont(-array) ⇝ *object*
Input contours.

▷ **ContoursProjTrans** (output_object) . . . . . xld_cont(-array) ⇝ *object*
Output contours.

▷ **HomMat2D** (input_control) . . . . . hom_mat2d ⇝ *real*
Homogeneous projective transformation matrix.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

vector_to_proj_hom_mat2d, hom_vector_to_proj_hom_mat2d, proj_match_points_ransac, proj_match_points_ransac_guided, hom_mat3d_project

See also

projective_trans_image, projective_trans_image_size, projective_trans_region, projective_trans_point_2d, projective_trans_pixel

Module

Foundation

29.5 Sets

```
difference_closed_contours_xld ( Contours,
    Sub :  ContoursDifference :  :  )
```

Compute the difference of closed contours.

The operator difference_closed_contours_xld calculates the difference of the regions enclosed by the contours Contours and Sub. The boundaries of this difference are returned in ContoursDifference.

If the individual input contours are not closed, they are closed automatically by connecting their start and end points.

Internally, first, the regions enclosed by each set of contours (Contours and Sub, respectively) are determined as described below. Then, the difference of these two regions is calculated.

In the following, the two sets of input contours are referred to as *boundary sets*. A boundary set consists of an arbitrary number of boundaries (the individual contours of Contours and Sub, respectively). Each boundary may be convex or concave and self-intersecting. Internal holes may be formed by the nesting of boundaries.

The region enclosed by such a boundary set is defined by the so-called Even-Odd-Rule. Thus, it consists of all points with the following property: The line constructed by connecting the point with another reference point that lies outside of the region has an odd number of intersections with boundaries.

More descriptive, this means that a boundary defines a hole in the region enclosed by another boundary if it lies completely inside that region. Similarly, the overlapping area of two separate boundaries of one boundary set is taken as a "hole", i.e., it does not belong to the region enclosed by the two boundaries. Note that the region enclosed by a boundary does not depend on the orientation of the boundary.

A self-intersecting boundary may be split at the intersection point or it is reordered such that it touches itself at the intersection point. The resulting boundaries are treated as separate boundaries.

The resulting boundaries ContoursDifference are automatically classified into boundaries that enclose regions and boundaries that enclose holes. This information is stored in the global contour attribute *'is_hole'*. For further information about global contour attributes see get_contour_global_attrib_xld.

Note that subsequent points whose row and column coordinates differ by less than 1e-06 pixels are considered as one point.

Parameters

▷ **Contours** (input_object) . . . . . xld_cont(-array) ⇝ *object*
Contours enclosing the region from which the second region is subtracted.

▷ **Sub** (input_object) . . . . . xld_cont(-array) ⇝ *object*
Contours enclosing the region that is subtracted from the first region.

▷ **ContoursDifference** (output_object) . . . . . xld_cont(-array) ⇝ *object*
Contours enclosing the difference.

Result

`difference_closed_contours_xld` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_contour_region_xld`, `zero_crossing_sub_pix`, `threshold_sub_pix`

Alternatives

`difference`

See also

`intersection_closed_contours_xld`, `symm_difference_closed_contours_xld`, `union2_closed_contours_xld`, `difference_closed_polygons_xld`

Module

Foundation

difference_closed_polygons_xld (Polygons, Sub : PolygonsDifference : :)

Compute the difference of closed polygons.

The operator `difference_closed_polygons_xld` calculates the difference of the regions enclosed by the polygons `Polygons` and `Sub`. The boundaries of this difference are returned in `PolygonsDifference`.

If the individual input polygons are not closed, they are closed automatically by connecting their start and end points.

Internally, first, the regions enclosed by each set of polygons (`Polygons` and `Sub`, respectively) are determined as described below. Then, the difference of these two regions is calculated.

In the following, the two sets of input polygons are referred to as *boundary sets*. A boundary set consists of an arbitrary number of boundaries (the individual polygons of `Polygons` and `Sub`, respectively). Each boundary may be convex or concave and self-intersecting. Internal holes may be formed by the nesting of boundaries.

The region enclosed by such a boundary set is defined by the so-called Even-Odd-Rule. Thus, it consists of all points with the following property: The line constructed by connecting the point with another reference point that lies outside of the region has an odd number of intersections with boundaries.

More descriptive, this means that a boundary defines a hole in the region enclosed by another boundary if it lies completely inside that region. Similarly, the overlapping area of two separate boundaries of one boundary set is taken as a "hole", i.e., it does not belong to the region enclosed by the two boundaries. Note that the region enclosed by a boundary does not depend on the orientation of the boundary.

A self-intersecting boundary may be split at the intersection point or it is reordered such that it touches itself at the intersection point. The resulting boundaries are treated as separate boundaries.

Attention

The resulting polygons `PolygonsDifference` contain no references to the XLD contours that are possibly referenced by `Polygons` and `Sub`. Hence, operators that access the contours associated with a polygon, e.g., `split_contours_xld` will not work correctly.

Parameters

▷ **Polygons** (input_object) . . . . . xld_poly(-array) ⇝ *object*
Polygons enclosing the region from which the second region is subtracted.

▷ **Sub** (input_object) . . . . . xld_poly(-array) ⇝ *object*
Polygons enclosing the region that is subtracted from the first region.

▷ **PolygonsDifference** (output_object) . . . . . xld_poly(-array) ⇝ *object*
Polygons enclosing the difference.

Result

`difference_closed_polygons_xld` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_polygons_xld`

Alternatives

`difference`

See also

`intersection_closed_polygons_xld`, `symm_difference_closed_polygons_xld`, `union2_closed_polygons_xld`, `difference_closed_contours_xld`

Module

Foundation

intersection_closed_contours_xld (Contours1, Contours2 : : ContoursIntersection : :)

Intersect closed contours.

The operator `intersection_closed_contours_xld` calculates the intersection of the regions enclosed by the contours `Contours1` and `Contours2`. The boundaries of this intersection are returned in `ContoursIntersection`.

If the individual input contours are not closed, they are closed automatically by connecting their start and end points.

Internally, first, the regions enclosed by each set of contours (`Contours1` and `Contours2`, respectively) are determined as described below. Then, the intersection of these two regions is calculated.

In the following, the two sets of input contours are referred to as *boundary sets*. A boundary set consists of an arbitrary number of boundaries (the individual contours of `Contours1` and `Contours2`, respectively). Each boundary may be convex or concave and self-intersecting. Internal holes may be formed by the nesting of boundaries.

The region enclosed by such a boundary set is defined by the so-called Even-Odd-Rule. Thus, it consists of all points with the following property: The line constructed by connecting the point with another reference point that lies outside of the region has an odd number of intersections with boundaries.

More descriptive, this means that a boundary defines a hole in the region enclosed by another boundary if it lies completely inside that region. Similarly, the overlapping area of two separate boundaries of one boundary set is taken as a "hole", i.e., it does not belong to the region enclosed by the two boundaries. Note that the region enclosed by a boundary does not depend on the orientation of the boundary.

A self-intersecting boundary may be split at the intersection point or it is reordered such that it touches itself at the intersection point. The resulting boundaries are treated as separate boundaries.

The resulting boundaries `ContoursIntersection` are automatically classified into boundaries that enclose regions and boundaries that enclose holes. This information is stored in the global contour attribute *'is_hole'*. For further information about global contour attributes see `get_contour_global_attrib_xld`.

Note that subsequent points whose row and column coordinates differ by less than 1e-06 pixels are considered as one point.

Parameters

▷ **Contours1** (input_object) ..... xld_cont(-array) ⇝ *object*
Contours enclosing the first region to be intersected.

▷ **Contours2** (input_object) ..... xld_cont(-array) ⇝ *object*
Contours enclosing the second region to be intersected.

▷ **ContoursIntersection** (output_object) ..... xld_cont(-array) ⇝ *object*
Contours enclosing the intersection.

Result

`intersection_closed_contours_xld` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_contour_region_xld`, `zero_crossing_sub_pix`, `threshold_sub_pix`

Alternatives

`intersection`

See also

`difference_closed_contours_xld`, `symm_difference_closed_contours_xld`, `union2_closed_contours_xld`, `intersection_closed_polygons_xld`

Module

Foundation

intersection_closed_polygons_xld (Polygons1, Polygons2 : PolygonsIntersection : :)

Intersect closed polygons.

The operator `intersection_closed_polygons_xld` calculates the intersection of the regions enclosed by the polygons `Polygons1` and `Polygons2`. The boundaries of this intersection are returned in `PolygonsIntersection`.

If the individual input polygons are not closed, they are closed automatically by connecting their start and end points.

Internally, first, the regions enclosed by each set of polygons (`Polygons1` and `Polygons2`, respectively) are determined as described below. Then, the intersection of these two regions is calculated.

In the following, the two sets of input polygons are referred to as *boundary sets*. A boundary set consists of an arbitrary number of boundaries (the individual polygons of `Polygons1` and `Polygons2`, respectively). Each boundary may be convex or concave and self-intersecting. Internal holes may be formed by the nesting of boundaries.

The region enclosed by such a boundary set is defined by the so-called Even-Odd-Rule. Thus, it consists of all points with the following property: The line constructed by connecting the point with another reference point that lies outside of the region has an odd number of intersections with boundaries.

More descriptive, this means that a boundary defines a hole in the region enclosed by another boundary if it lies completely inside that region. Similarly, the overlapping area of two separate boundaries of one boundary set is taken as a "hole", i.e., it does not belong to the region enclosed by the two boundaries. Note that the region enclosed by a boundary does not depend on the orientation of the boundary.

A self-intersecting boundary may be split at the intersection point or it is reordered such that it touches itself at the intersection point. The resulting boundaries are treated as separate boundaries.

Attention

The resulting polygons PolygonsIntersection contain no references to the XLD contours that are possibly referenced by Polygons1 and Polygons2. Hence, operators that access the contours associated with a polygon, e.g., split_contours_xld will not work correctly.

Parameters

▷ **Polygons1** (input_object) ..... xld_poly(-array) ⇝ *object*
Polygons enclosing the first region to be intersected.

▷ **Polygons2** (input_object) ..... xld_poly(-array) ⇝ *object*
Polygons enclosing the second region to be intersected.

▷ **PolygonsIntersection** (output_object) ..... xld_poly(-array) ⇝ *object*
Polygons enclosing the intersection.

Result

intersection_closed_polygons_xld returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_polygons_xld

Alternatives

intersection

See also

difference_closed_polygons_xld, symm_difference_closed_polygons_xld, union2_closed_polygons_xld, intersection_closed_contours_xld

Module

Foundation

intersection_region_contour_xld (Region, Contour : ContourIntersection : Mode :)

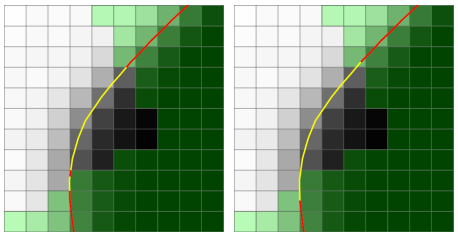
Intersect a contour with a region.

The operator intersection_region_contour_xld intersects the contour Contour with the region Region and returns the parts of the contour that are within the region in ContourIntersection.

The type of intersection can be specified with Mode. The following values are possible:

'lines' **(Default):** In *'lines'* mode, each line segment of the input contour is intersected with the region with subpixel accuracy. This mode is usually slower than *'points'* but more accurate. Line segments that are only partially within the region are split into multiple segments if necessary.

*'points'***:** In *'points'* mode, entire line segments from Contour are tested against the region using their start and end point. If both the start and end point of a line segment are within the region, that line segment is included in the output. This mode is faster than *'lines'* but less accurate, especially if the contour is coarsely sampled. For example, line segments might leave the region between their start and end point, but are still included in the output. Further, a line segment might be contained in the region for the most part but is not included in the output if the start or end point are not in the region.



(1) (2)

Result for *'lines'* (1) and *'points'* mode (2) when intersecting a contour with a region (green). The red sections of the contour will be returned as the result.

Parameters

▷ **Region** (input_object) . . . . . region ⇝ *object*
Input region.

▷ **Contour** (input_object) . . . . . xld_cont-array ⇝ *object*
Input contour.

▷ **ContourIntersection** (output_object) . . . . . xld_cont-array ⇝ *object*
Selected part of the input contour.

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Intersection mode.
Default: 'lines'
List of values: Mode ∈ {'points', 'lines'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

threshold, gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix, gen_contour_region_xld, zero_crossing_sub_pix, threshold_sub_pix

Alternatives

intersection_closed_contours_xld, intersection_closed_polygons_xld

See also

intersection_closed_contours_xld, intersection_closed_polygons_xld, gen_region_contour_xld

Module

Foundation

symm_difference_closed_contours_xld (Contours1, Contours2 : ContoursDifference : :)

Compute the symmetric difference of closed contours.

The operator symm_difference_closed_contours_xld calculates the symmetric difference of the regions enclosed by the contours Contours1 and Contours2. The boundaries of this symmetric difference are returned in ContoursDifference.

If the individual input contours are not closed, they are closed automatically by connecting their start and end points.

Internally, first, the regions enclosed by each set of contours (Contours1 and Contours2, respectively) are determined as described below. Then, the symmetric difference of these two regions is calculated.

In the following, the two sets of input contours are referred to as *boundary sets*. A boundary set consists of an arbitrary number of boundaries (the individual contours of Contours1 and Contours2, respectively). Each

boundary may be convex or concave and self-intersecting. Internal holes may be formed by the nesting of boundaries.

The region enclosed by such a boundary set is defined by the so-called Even-Odd-Rule. Thus, it consists of all points with the following property: The line constructed by connecting the point with another reference point that lies outside of the region has an odd number of intersections with boundaries.

More descriptive, this means that a boundary defines a hole in the region enclosed by another boundary if it lies completely inside that region. Similarly, the overlapping area of two separate boundaries of one boundary set is taken as a "hole", i.e., it does not belong to the region enclosed by the two boundaries. Note that the region enclosed by a boundary does not depend on the orientation of the boundary.

A self-intersecting boundary may be split at the intersection point or it is reordered such that it touches itself at the intersection point. The resulting boundaries are treated as separate boundaries.

The resulting boundaries `ContoursDifference` are automatically classified into boundaries that enclose regions and boundaries that enclose holes. This information is stored in the global contour attribute *'is_hole'*. For further information about global contour attributes see `get_contour_global_attrib_xld`.

Note that subsequent points whose row and column coordinates differ by less than 1e-06 pixels are considered as one point.

Parameters

▷ **`Contours1`** (input_object) ..... xld_cont(-array) ⇝ *object*
Contours enclosing the first region.

▷ **`Contours2`** (input_object) ..... xld_cont(-array) ⇝ *object*
Contours enclosing the second region.

▷ **`ContoursDifference`** (output_object) ..... xld_cont(-array) ⇝ *object*
Contours enclosing the symmetric difference.

Result

`symm_difference_closed_contours_xld` returns `2` (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_contour_region_xld`, `zero_crossing_sub_pix`, `threshold_sub_pix`

Alternatives

`symm_difference`

See also

`intersection_closed_contours_xld`, `difference_closed_contours_xld`, `union2_closed_contours_xld`, `symm_difference_closed_polygons_xld`

Module

Foundation

`symm_difference_closed_polygons_xld` (`Polygons1`, `Polygons2` : `PolygonsDifference` : :)

Compute the symmetric difference of closed polygons.

The operator `symm_difference_closed_polygons_xld` calculates the symmetric difference of the regions enclosed by the polygons `Polygons1` and `Polygons2`. The boundaries of this symmetric difference are returned in `PolygonsDifference`.

If the individual input polygons are not closed, they are closed automatically by connecting their start and end points.

Internally, first, the regions enclosed by each set of polygons (Polygons1 and Polygons2, respectively) are determined as described below. Then, the symmetric difference of these two regions is calculated.

In the following, the two sets of input polygons are referred to as *boundary sets*. A boundary set consists of an arbitrary number of boundaries (the individual polygons of Polygons1 and Polygons2, respectively). Each boundary may be convex or concave and self-intersecting. Internal holes may be formed by the nesting of boundaries.

The region enclosed by such a boundary set is defined by the so-called Even-Odd-Rule. Thus, it consists of all points with the following property: The line constructed by connecting the point with another reference point that lies outside of the region has an odd number of intersections with boundaries.

More descriptive, this means that a boundary defines a hole in the region enclosed by another boundary if it lies completely inside that region. Similarly, the overlapping area of two separate boundaries of one boundary set is taken as a "hole", i.e., it does not belong to the region enclosed by the two boundaries. Note that the region enclosed by a boundary does not depend on the orientation of the boundary.

A self-intersecting boundary may be split at the intersection point or it is reordered such that it touches itself at the intersection point. The resulting boundaries are treated as separate boundaries.

Attention

The resulting polygons PolygonsDifference contain no references to the XLD contours that are possibly referenced by Polygons1 and Polygons2. Hence, operators that access the contours associated with a polygon, e.g., split_contours_xld will not work correctly.

Parameters

- ▷ **Polygons1** (input_object) ..... xld_poly(-array) ⇝ *object*
  Polygons enclosing the first region.
- ▷ **Polygons2** (input_object) ..... xld_poly(-array) ⇝ *object*
  Polygons enclosing the second region.
- ▷ **PolygonsDifference** (output_object) ..... xld_poly(-array) ⇝ *object*
  Polygons enclosing the symmetric difference.

Result

symm_difference_closed_polygons_xld returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_polygons_xld

Alternatives

symm_difference

See also

intersection_closed_polygons_xld, difference_closed_polygons_xld, union2_closed_polygons_xld, symm_difference_closed_contours_xld

Module

Foundation

union2_closed_contours_xld (Contours1, Contours2 : ContoursUnion : :)

Compute the union of closed contours.

The operator union2_closed_contours_xld calculates the union of the regions enclosed by the contours Contours1 and Contours2. The boundaries of this union are returned in ContoursUnion.

If the individual input contours are not closed, they are closed automatically by connecting their start and end points.

Internally, first, the regions enclosed by each set of contours (Contours1 and Contours2, respectively) are determined as described below. Then, the union of these two regions is calculated.

In the following, the two sets of input contours are referred to as *boundary sets*. A boundary set consists of an arbitrary number of boundaries (the individual contours of Contours1 and Contours2, respectively). Each boundary may be convex or concave and self-intersecting. Internal holes may be formed by the nesting of boundaries.

The region enclosed by such a boundary set is defined by the so-called Even-Odd-Rule. Thus, it consists of all points with the following property: The line constructed by connecting the point with another reference point that lies outside of the region has an odd number of intersections with boundaries.

More descriptive, this means that a boundary defines a hole in the region enclosed by another boundary if it lies completely inside that region. Similarly, the overlapping area of two separate boundaries of one boundary set is taken as a "hole", i.e., it does not belong to the region enclosed by the two boundaries. Note that the region enclosed by a boundary does not depend on the orientation of the boundary.

A self-intersecting boundary may be split at the intersection point or it is reordered such that it touches itself at the intersection point. The resulting boundaries are treated as separate boundaries.

The resulting boundaries ContoursUnion are automatically classified into boundaries that enclose regions and boundaries that enclose holes. This information is stored in the global contour attribute *'is_hole'*. For further information about global contour attributes see get_contour_global_attrib_xld.

Note that subsequent points whose row and column coordinates differ by less than 1e-06 pixels are considered as one point.

Parameters

▷ **Contours1** (input_object) ..... xld_cont(-array) ⇝ *object*
Contours enclosing the first region.

▷ **Contours2** (input_object) ..... xld_cont(-array) ⇝ *object*
Contours enclosing the second region.

▷ **ContoursUnion** (output_object) ..... xld_cont(-array) ⇝ *object*
Contours enclosing the union.

Result

union2_closed_contours_xld returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contour_region_xld, zero_crossing_sub_pix, threshold_sub_pix

Alternatives

union2

See also

difference_closed_contours_xld, symm_difference_closed_contours_xld, intersection_closed_contours_xld, union2_closed_polygons_xld

Module

Foundation

union2_closed_polygons_xld (Polygons1, Polygons2 : PolygonsUnion : :)

Compute the union of closed polygons.

The operator union2_closed_polygons_xld calculates the union of the regions enclosed by the polygons Polygons1 and Polygons2. The boundaries of this union are returned in PolygonsUnion.

If the individual input polygons are not closed, they are closed automatically by connecting their start and end points.

Internally, first, the regions enclosed by each set of polygons (Polygons1 and Polygons2, respectively) are determined as described below. Then, the union of these two regions is calculated.

In the following, the two sets of input polygons are referred to as *boundary sets*. A boundary set consists of an arbitrary number of boundaries (the individual polygons of Polygons1 and Polygons2, respectively). Each boundary may be convex or concave and self-intersecting. Internal holes may be formed by the nesting of boundaries.

The region enclosed by such a boundary set is defined by the so-called Even-Odd-Rule. Thus, it consists of all points with the following property: The line constructed by connecting the point with another reference point that lies outside of the region has an odd number of intersections with boundaries.

More descriptive, this means that a boundary defines a hole in the region enclosed by another boundary if it lies completely inside that region. Similarly, the overlapping area of two separate boundaries of one boundary set is taken as a "hole", i.e., it does not belong to the region enclosed by the two boundaries. Note that the region enclosed by a boundary does not depend on the orientation of the boundary.

A self-intersecting boundary may be split at the intersection point or it is reordered such that it touches itself at the intersection point. The resulting boundaries are treated as separate boundaries.

Attention

The resulting polygons PolygonsUnion contain no references to the XLD contours that are possibly referenced by Polygons1 and Polygons2. Hence, operators that access the contours associated with a polygon, e.g., split_contours_xld will not work correctly.

Parameters

- ▷ **Polygons1** (input_object) ..... xld_poly(-array) ⇝ *object*
  Polygons enclosing the first region.
- ▷ **Polygons2** (input_object) ..... xld_poly(-array) ⇝ *object*
  Polygons enclosing the second region.
- ▷ **PolygonsUnion** (output_object) ..... xld_poly(-array) ⇝ *object*
  Polygons enclosing the union.

Result

union2_closed_polygons_xld returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_polygons_xld

Alternatives

union2

See also

difference_closed_polygons_xld, symm_difference_closed_polygons_xld, intersection_closed_polygons_xld, union2_closed_contours_xld

Module

Foundation

29.6 Transformations

add_noise_white_contour_xld (
Contours : NoisyContours : NumRegrPoints, Amp :)

Add noise to XLD contours.

add_noise_white_contour_xld adds noise to the input XLD contours Contours and returns the noisy contours in NoisyContours. For each point along the original contours the local regression line (i.e., a least-squares approximating line) based on NumRegrPoints neighboring points is determined. Then the point is shifted perpendicular to this line. The length of the shifts corresponds to white noise, equally distributed in the interval [-Amp,Amp].

The random noise is generated using the C function "drand48()". See the parameter *'seed_rand'* of set_system for information on the used random seed.

Parameters

- ▷ **Contours** (input_object) ..... xld_cont(-array) ⇝ *object*
  Original contours.
- ▷ **NoisyContours** (output_object) ..... xld_cont(-array) ⇝ *object*
  Noisy contours.
- ▷ **NumRegrPoints** (input_control) ..... integer ⇝ *integer*
  Number of points used to calculate the regression line.
  Default: 5
  Suggested values: NumRegrPoints ∈ {3, 5, 7, 9}
  Restriction: NumRegrPoints >= 3 && odd(NumRegrPoints)
- ▷ **Amp** (input_control) ..... real ⇝ *real*
  Maximum amplitude of the added noise (equally distributed in [-Amp,Amp]).
  Default: 1.0
  Suggested values: Amp ∈ {0.25, 0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 10.0}
  Restriction: Amp > 0

Example

```
draw_ellipse(WindowHandle,Row,Column,Phi,Radius1,Radius2)
gen_ellipse_contour_xld(Ellipse,Row,Column,Phi,Radius1,Radius2,0,6.28319, \
                        'positive', 1.5)
add_noise_white_contour_xld(Ellipse,NoisyEllipse,5,0.5)
fit_ellipse_contour_xld (NoisyEllipse, 'fitzgibbon', -1, 2, 0, 200, 3, 2.0, \
                         ERow, EColumn, EPhi, ERadius1, ERadius2, \
                         EStartPhi, EEndPhi, EPointOrder)
```

Result

add_noise_white_contour_xld returns 2 (H_MSG_TRUE) if all parameter values are correct. If the input is empty the behavior can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix, gen_ellipse_contour_xld

Possible Successors

smooth_contours_xld

See also

smooth_contours_xld, add_noise_white

Module

Foundation

clip_contours_xld (Contours : ClippedContours : Row1, Column1, Row2, Column2 :)

Clip an XLD contour.

clip_contours_xld clips all XLD contours given in Contours, i.e., only contour points contained in the rectangle given by Row1, Column1, Row2, and Column2 are returned on output. Note, that the rectangle behaves like a region, i.e., the rectangle encloses the pixels completely. If necessary, contours are split, and several new contours are produced. The resulting contours are returned in ClippedContours.

Parameters

- ▷ **Contours** (input_object) . . . . . xld_cont(-array) ⇝ *object*
  Contours to be clipped.
- ▷ **ClippedContours** (output_object) . . . . . xld_cont(-array) ⇝ *object*
  Clipped contours.
- ▷ **Row1** (input_control) . . . . . rectangle.origin.y ⇝ *integer*
  Row coordinate of the upper left corner of the clip rectangle.
  Default: 0
  Suggested values: Row1 ∈ {0, 500, 1000, 1500, 2000}
- ▷ **Column1** (input_control) . . . . . rectangle.origin.x ⇝ *integer*
  Column coordinate of the upper left corner of the clip rectangle.
  Default: 0
  Suggested values: Column1 ∈ {0, 500, 1000, 1500, 2000}
- ▷ **Row2** (input_control) . . . . . rectangle.corner.y ⇝ *integer*
  Row coordinate of the lower right corner of the clip rectangle.
  Default: 512
  Suggested values: Row2 ∈ {512, 1024, 1536, 2048}
- ▷ **Column2** (input_control) . . . . . rectangle.corner.x ⇝ *integer*
  Column coordinate of the lower right corner of the clip rectangle.
  Default: 512
  Suggested values: Column2 ∈ {512, 1024, 1536, 2048}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix

Possible Successors

gen_polygons_xld

Alternatives

crop_contours_xld

See also

clip_region, crop_part, crop_contours_xld

Module

Foundation

clip_end_points_contours_xld (Contours : ClippedContours : Mode, Length :)

Clip the end points of an XLD contour.

The operator clip_end_points_contours_xld clips the end points of an XLD contour. The parameter Mode determines the measure used to determine which part of the contour is clipped.

If Mode = *'length'*, the Euclidean length of the part to be clipped is passed in Length. If Mode = *'num_points'*, the number of points to be clipped is passed in Length.

The clipped contour is returned in ClippedContours. If all points of the input contour are clipped, no contour is returned. Therefore, the number of returned contours is less or equal to the number of input contours.

Parameters

▷ **Contours** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contour

▷ **ClippedContours** (output_object) ..... xld_cont(-array) ⇝ *object*
Clipped contour

▷ **Mode** (input_control) ..... string ⇝ *string*
Clipping mode.
Default: 'num_points'
List of values: Mode ∈ {'num_points', 'length'}

▷ **Length** (input_control) ..... number ⇝ *real* / integer
Clipping length in unit pixels (Mode = *'length'*) or number (Mode = *'num_points'*)
Default: 3
Suggested values: Length ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Result

clip_end_points_contours_xld returns 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contour available) is set via the operator set_system ('no_object_result',<Result>). If necessary an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

segment_contours_xld

See also

clip_contours_xld, crop_contours_xld

Module

Foundation

close_contours_xld (Contours : ClosedContours : :)

Close an XLD contour.

close_contours_xld closes all XLD contours given in Contours. The resulting contours are returned in ClosedContours. To obtain meaningful results, it is strongly recommended to select contours with select_contours_xld that have end points that are not further apart than a certain maximum distance before passing them to close_contours_xld.

Parameters

▷ **Contours** (input_object) ..... xld_cont-array ⇝ *object*
Contours to be closed.

▷ **ClosedContours** (output_object) ..... xld_cont(-array) ⇝ *object*
Closed contours.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).

- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`select_contours_xld`

Possible Successors

`area_center_xld`, `circularity_xld`, `compactness_xld`, `convexity_xld`, `eccentricity_xld`, `elliptic_axis_xld`, `moments_xld`, `moments_any_xld`, `orientation_xld`, `select_shape_xld`, `test_xld_point`

Module

Foundation

combine_roads_xld (EdgePolygons, ModParallels, ExtParallels, CenterLines : RoadSides : MaxAngleParallel, MaxAngleColinear, MaxDistanceParallel, MaxDistanceColinear :)

Combine road hypotheses from two resolution levels.

`combine_roads_xld` combines road hypotheses obtained from two different resolution levels. The algorithm selects only those hypotheses which mutually support each other in both resolution levels. The parameters `EdgePolygons`, `ModParallels` and `ExtParallels` correspond to the road hypotheses from the highest resolution level. The parameter `CenterLines` is the result of road extraction in a lower resolution level. Those of the polygons `EdgePolygons` are returned for which evidence of being roadsides is found in both resolution levels. The parameters `MaxAngleParallel` and `MaxAngleColinear` determine the angle, that may be formed by two parallel collinear line segments, respectively. The parameters `MaxDistanceParallel` and `MaxDistanceColinear` determine the maximum allowed distance of two parallel or collinear line segments, respectively. The combination is done using a number of rules.

Parameters

▷ **EdgePolygons** (input_object) ..... xld_poly-array ⇝ *object*
XLD polygons to be examined.

▷ **ModParallels** (input_object) ..... xld_mod_para-array ⇝ *object*
Modified parallels obtained from `EdgePolygons`.

▷ **ExtParallels** (input_object) ..... xld_ext_para-array ⇝ *object*
Extended parallels obtained from `EdgePolygons`.

▷ **CenterLines** (input_object) ..... xld_poly-array ⇝ *object*
Road-center-line polygons to be examined.

▷ **RoadSides** (output_object) ..... xld_poly-array ⇝ *object*
Roadsides found.

▷ **MaxAngleParallel** (input_control) ..... angle.rad ⇝ *real* / integer
Maximum angle between two parallel line segments.
Default: 0.523598775598
Suggested values: `MaxAngleParallel` ∈ {0.349065850399, 0.523598775598, 0.6981317008}
Restriction: 0 <= MaxAngleParallel <= pi / 2

▷ **MaxAngleColinear** (input_control) ..... angle.rad ⇝ *real* / integer
Maximum angle between two collinear line segments.
Default: 0.261799387799
Suggested values: `MaxAngleColinear` ∈ {0.174532925199, 0.261799387799, 0.349065850399}
Restriction: 0 <= MaxAngleColinear <= pi / 2

▷ **MaxDistanceParallel** (input_control) ..... real ⇝ *real* / integer
Maximum distance between two parallel line segments.
Default: 40
Suggested values: `MaxDistanceParallel` ∈ {20, 30, 40, 50, 60}
Restriction: MaxDistanceParallel > 0

▷ **MaxDistanceColinear** (input_control) ..... real ⇝ *real* / integer
Maximum distance between two collinear line segments.
Default: 40
Suggested values: MaxDistanceColinear ∈ {20, 30, 40, 50, 60}
Restriction: MaxDistanceColinear > 0

Execution Information

- Multithreading type: mutually exclusive (runs in parallel with other non-exclusive operators, but not with itself).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

mod_parallels_xld, gen_polygons_xld, affine_trans_contour_xld

Possible Successors

get_polygon_xld, get_lines_xld

See also

lines_gauss, lines_facet, get_channel_info, edges_sub_pix

References

C. Steger, C. Glock, W. Eckstein, H. Mayer, B. Radig; "Model-based Road Extraction from Images"; in "Automatic Extraction of Man-Made Objects from Aerial and Space Images"; A. Gruen, O. Kuebler, P. Agouris (Editors); Birkhäuser Verlag (1995), pp. 275-284.

C. Heipke, C. Steger, R. Multhammer; "A Hierarchical Approach to Automatic Road Extraction from Aerial Imagery"; in "Integrating Photogrammetric Techniques with Scene Analysis and Machine Vision II"; D. M. McKeown, Jr., I. J. Dowman (Editors); Proc. SPIE 2486 (1995), pp. 222-231.

Module

Foundation

crop_contours_xld (Contours : CroppedContours : Row1, Col1, Row2, Col2, CloseContours :)

Crop an XLD contour.

crop_contours_xld crops all XLD contours given in Contours, i.e., only contour segments contained in the rectangle given by Row1, Col1, Row2 and Col2 are returned on output. While in clip_contours_xld all contour segments that intersect a boundary line are omitted, here they are cropped at the exact intersection point with the cropping rectangle. If necessary, contours are split and several new contours are produced. Closed contours can, however, be closed again using the parameter CloseContours. If it is set to *'true'*, a closed contour is transformed into one or several contours that are closed via segments on the cropping rectangle. If it is *'false'*, all contours end at the points where they leave the cropping rectangle. The resulting contours are returned in CroppedContours.

Parameters

▷ **Contours** (input_object) ..... xld_cont(-array) ⇝ *object*
Input contours.

▷ **CroppedContours** (output_object) ..... xld_cont(-array) ⇝ *object*
Output contours.

▷ **Row1** (input_control) ..... number ⇝ *real* / integer
Upper border of the cropping rectangle.
Default: 0
Suggested values: Row1 ∈ {0, 500, 1000, 1500, 2000}

▷ **Col1** (input_control) ..... number ⇝ *real* / integer
Left border of the cropping rectangle.
Default: 0
Suggested values: Col1 ∈ {0, 500, 1000, 1500, 2000}

▷ **Row2** (input_control) ..... number ⇝ *real* / integer
Lower border of the cropping rectangle.
Default: 512
Suggested values: Row2 ∈ {512, 1024, 1536, 2048}

▷ **Col2** (input_control) ..... number ⇝ *real* / integer
Right border of the cropping rectangle.
Default: 512
Suggested values: Col2 ∈ {512, 1024, 1536, 2048}

▷ **CloseContours** (input_control) ..... string ⇝ *string*
Should closed contours produce closed output contours?
Default: 'true'
List of values: CloseContours ∈ {'true', 'false'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Alternatives

clip_contours_xld

See also

clip_region, crop_part, clip_contours_xld

Module

Foundation

merge_cont_line_scan_xld (CurrConts, PrevConts : CurrMergedConts, PrevMergedConts : ImageHeight, Margin, MergeBorder, MaxImagesCont :)

Merge XLD contours from successive line scan images.

The operator merge_cont_line_scan_xld connects adjacent XLD contours, which were extracted from adjacent images with the height ImageHeight. This operator was especially designed to connect contours that were extracted from images grabbed by a line scan camera. CurrConts contains the contours from the current image and PrevConts the contours from the previous one.

With the help of the parameter MergeBorder two cases can be distinguished: If the top (first) line of the current image touches the bottom (last) line of the previous image, MergeBorder must be set to *'top'*, otherwise set MergeBorder to *'bottom'*. MergeBorder defines a margin to the border. Only those end points of the contours which are inside this margin are considered for the following merging process.

If the operator merge_cont_line_scan_xld is used recursively, the parameter MaxImagesCont determines the maximum number of images which are covered by a merged contour. All points of the merged contour from an older image are removed.

The operator merge_cont_line_scan_xld returns two contour arrays. PrevMergedConts contains all those contours from the previous input contours PrevConts, which could not be merged with a current contour. CurrMergedConts collects all current contours together with the merged parts from the previous images. Merged contours will exceed the original image, because the previous contours are moved upward (MergeBorder=*'top'*) or downward (MergeBorder=*'bottom'*) according to the image height.

Parameters

▷ **CurrConts** (input_object) ..... xld_cont(-array) ⇝ *object*
Current input contours.

▷ **PrevConts** (input_object) ..... xld_cont(-array) ⇝ *object*
Merged contours from the previous iteration.

▷ **CurrMergedConts** (output_object) ..... xld_cont(-array) ⇝ *object*
Current contours, merged with old ones where applicable.

▷ **PrevMergedConts** (output_object) . . . . . xld_cont(-array) ⇝ *object*
Contours from the previous iteration which could not be merged with the current ones.

▷ **ImageHeight** (input_control) . . . . . integer ⇝ *integer*
Height of the line scan images.
Default: 512
Suggested values: ImageHeight ∈ {240, 480, 512}

▷ **Margin** (input_control) . . . . . real ⇝ *real* / integer
Maximum distance of contours from the image border.
Default: 0.0
Suggested values: Margin ∈ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0}

▷ **MergeBorder** (input_control) . . . . . string ⇝ *string*
Image line of the current image, which touches the previous image.
Default: 'top'
List of values: MergeBorder ∈ {'top', 'bottom'}

▷ **MaxImagesCont** (input_control) . . . . . integer ⇝ *integer*
Maximum number of images covered by one contour.
Default: 3
Suggested values: MaxImagesCont ∈ {1, 2, 3, 4, 5}

Result

The operator merge_cont_line_scan_xld returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Module

Foundation

regress_contours_xld (Contours : RegressContours : Mode, Iterations :)

Calculate the parameters of a regression line to an XLD contour.

regress_contours_xld calculates the following parameters for the input XLD contours Contours, and stores them with the resulting contours as global attributes:

- the coordinates of the normal vector of the regression line, i.e., the least-squares approximating line, of all contour points; the normal vector always points from the origin to the line (attributes: *'regr_norm_row'*, *'regr_norm_col'*),
- the mean of the Euclidean distance of the contour points from the regression line (attribute: *'regr_mean_dist'*),
- the standard deviation of these distances to the regression line (attribute: *'regr_dev_dist'*),
- the distance of the regression line from the origin (attribute: *'regr_dist'*).

See get_contour_global_attrib_xld for further information about global contour attributes.

For Mode = *'no'*, the parameters of the regression line are calculated for all points of the contour. In addition, three different kinds of outlier treatment can be applied. Outliers are contour points which do not lie on the general contour direction in an "obvious" manner, and thus "distort" the resulting regression line.

Mode =

- *'drop'*: All contour points further away from the contour than the mean distance to the regression line are ignored for the calculation of the undistorted regression line.

- *'gauss'*: The distances of the contour points are weighted according to their probability of occurrence in a Gaussian distribution around the normal regression line.
- *'median'*: Here, also a normal distribution is assumed for the distances to the normal regression line, however with the outlier-independent standard deviation $\frac{M_d}{0.6745}$, with M_d: median of all distances. Again, the distances are weighted, and points further away than a certain distance are ignored for the undistorted regression line.

The calculation of the undistorted regression line can be iterated several times (`Iterations`).

Parameters

▷ **`Contours`** (input_object) ..... xld_cont-array ⇝ *object*
Input XLD contours.

▷ **`RegressContours`** (output_object) ..... xld_cont-array ⇝ *object*
Resulting XLD contours.

▷ **`Mode`** (input_control) ..... string ⇝ *string*
Type of outlier treatment.
Default: 'no'
List of values: `Mode` ∈ {'no', 'drop', 'gauss', 'median'}

▷ **`Iterations`** (input_control) ..... integer ⇝ *integer*
Number of iterations for the outlier treatment.
Default: 1
Suggested values: `Iterations` ∈ {1, 2, 3, 5, 10, 20}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`gen_contours_skeleton_xld`, `lines_gauss`, `lines_facet`, `edges_sub_pix`

Possible Successors

`get_regress_params_xld`

See also

`smooth_contours_xld`, `get_contour_global_attrib_xld`, `query_contour_global_attribs_xld`

References

H. Suesse, K. Voss: "Adaptive Ausgleichsrechnung und Ausreißerproblematik für die digitale Bildverarbeitung"; Proc. 15. DAGM Symposium, Springer Verlag, Lübeck 1993

R. Haralick, L. Shapiro: "Computer and Robot Vision" Vol. 2; Kapitel 14.9, Addison-Wesley 1992

Module

Foundation

`segment_contour_attrib_xld` (Contour : ContourPart : Attribute, Operation, Min, Max :)

Segment XLD contour parts whose local attributes fulfill given conditions.

`segment_contour_attrib_xld` segments the contour in `Contour` into contour parts `ContourPart`. The segmentation of the contour depends on the values of its attributes `Attribute` (see the possible values for `Attribute` below). The operator checks which attribute values lie within the given limits (`Min`, `Max`) and returns the corresponding contour segments with its attributes in `ContourPart`.

If `Operation` is set to *'and'*, each of the given attributes must lie within the limits. If `Operation` is set to *'or'*, at least one of the attributes has to be within the limits. The attributes are processed in the same order as they are given in `Attribute`. If only one attribute is used, the value of `Operation` is ignored. The parameters `Min` and `Max` can be set to *'min'* or *'max'* in order to leave bottom and top limit, respectively, open.

Note that not all attributes are defined in all contours per default. Which attributes are defined for a contour depends on how the contour was instantiated, and whether some operations have been previously performed on it. For example, the attributes *'width_right'* and *'width_left'* are only defined, if the contour was created with the operator `lines_gauss`. Likewise, the attribute *'distance'* is only present if the distances to some other contour was previously calculated with `distance_contours_xld` or `apply_distance_transform_xld`.

Condition: $Min_i \leq Attribute_i(Object) \leq Max_i$

Possible values for `Attribute`:

'edge_direction': Edge direction

'angle': Angle of the direction perpendicular to the contour

'response': Magnitude of the second derivative

'width_right': Line width to the right of the line

'width_left': Line width to the left of the line

'contrast': Contrast of the line point

'asymmetry': Asymmetry of the line point

'distance': Minimum pointwise distance to a reference contour

To find out which operators add the attributes above to contours and further details about those attributes, see `get_contour_attrib_xld`.

Parameters

▷ **`Contour`** (input_object) ..... xld_cont(-array) ⇝ *object*
Contour to be segmented.

▷ **`ContourPart`** (output_object) ..... xld_cont-array ⇝ *object*
Segmented contour parts.

▷ **`Attribute`** (input_control) ..... string(-array) ⇝ *string*
Contour attributes to be checked.
Default: 'distance'
List of values: `Attribute` ∈ {'distance', 'angle', 'edge_direction', 'width_right', 'width_left', 'response', 'contrast', 'asymmetry'}

▷ **`Operation`** (input_control) ..... string ⇝ *string*
Linkage type of the individual attributes.
Default: 'and'
List of values: `Operation` ∈ {'and', 'or'}

▷ **`Min`** (input_control) ..... real(-array) ⇝ *real* / integer / string
Lower limits of the attribute values.
Default: 150.0
Minimum increment: 0.001
Recommended increment: 1.0

▷ **`Max`** (input_control) ..... real(-array) ⇝ *real* / integer / string
Upper limits of the attribute values.
Default: 99999.0
Minimum increment: 0.001
Recommended increment: 1.0
Restriction: Max >= Min

Example

```
read_image (Image, 'fabrik')
edges_sub_pix (Image, Edges, 'canny', 1, 20, 40)
* select 'vertical' edges:
segment_contour_attrib_xld (Edges, ContourPart, 'edge_direction', \
                            'and', rad(90-20), rad(90+20))
```

Result

The operator `segment_contour_attrib_xld` returns the value `2` (`H_MSG_TRUE`) if the input is not

empty. The behavior in case of empty input (no input objects available) is set via the operator set_system ('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

lines_gauss, edges_sub_pix, apply_distance_transform_xld, distance_contours_xld

Possible Successors

fit_line_contour_xld, fit_ellipse_contour_xld, fit_circle_contour_xld, fit_rectangle2_contour_xld

See also

split_contours_xld, get_contour_global_attrib_xld, query_contour_attribs_xld, get_contour_attrib_xld

Module

Foundation

segment_contours_xld (Contours : ContoursSplit : Mode, SmoothCont, MaxLineDist1, MaxLineDist2 :)

Segment XLD contours into line segments and circular or elliptic arcs.

segment_contours_xld segments the input contours Contours into lines if Mode='*lines*', into lines and circular arcs if Mode='*lines_circles*', or into lines and elliptic arcs if Mode='*lines_ellipses*'. The segmented contours are returned in ContoursSplit. The information on whether an output contour represents a line or a circular or elliptic arc is done via the global contour attribute *'cont_approx'* (see get_contour_global_attrib_xld for further information).

segment_contours_xld first approximates the input contours by polygons. With this, the contours are oversegmented in curved areas. After this, the contours are approximated according to the modes Mode given below. If SmoothCont is set to a value > 0, the input contours are first smoothed (see smooth_contours_xld). This can be necessary to prevent very short segments in the polygonal approximation and to achieve a more robust fit with circular or elliptic arcs, because the smoothing suppresses outliers on the contours.

The calculation of the segmentation depends on the respective selected method in Mode:

'lines' The Ramer algorithm is used to perform a polygon approximation with a maximum distance of MaxLineDist1 (see gen_polygons_xld). If Mode='*lines*', the parameter MaxLineDist2 is ignored.

'lines_circles' **or** *'lines_ellipses'* The initial polygonal approximation is done by using the Ramer algorithm (see gen_polygons_xld) with a maximum distance of MaxLineDist1. If the maximum distance of the resulting arc to the contour is smaller than the maximum distance of the two line segments, the two line segments are replaced with the arc. This procedure is iterated until no more changes occur. After this, the parts of the contour that are still approximated by line segments are again segmented with a polygonal approximation with maximum distance MaxLineDist2, and the newly created line segments are merged to circular or elliptical arcs where possible. This changes the output only if MaxLineDist2 differs from MaxLineDist1. This two-step approach is more efficient than a one-step approach with MaxLineDist2, since fewer line segments are generated in the first step, and hence the circle or ellipse fit has to be performed less often. Therefore, parts of the input contours that can be approximated by long arcs can be found more efficiently. In the second step, parts of the input contours that can be approximated by short arcs are found and the end parts of long arcs are refined. In order to benefit from this two-step approach it is recommended to set MaxLineDist2 $<$ MaxLineDist1.

The resulting contours are at least 3 pixel long and contain at least 6 successive points of the input contour. All input contours with a length less than 3 pixel or with less than 6 contour points are copied to the output contours without modifications.

Parameters

- ▷ **Contours** (input_object) . . . . . xld_cont(-array) ⇝ *object*
  Contours to be segmented.
- ▷ **ContoursSplit** (output_object) . . . . . xld_cont-array ⇝ *object*
  Segmented contours.
- ▷ **Mode** (input_control) . . . . . string ⇝ *string*
  Mode for the segmentation of the contours.
  Default: 'lines_circles'
  List of values: Mode ∈ {'lines', 'lines_circles', 'lines_ellipses'}
- ▷ **SmoothCont** (input_control) . . . . . integer ⇝ *integer*
  Number of points used for smoothing the contours.
  Default: 5
  Suggested values: SmoothCont ∈ {0, 3, 5, 7, 9}
  Restriction: SmoothCont == 0 || SmoothCont >= 3 && odd(SmoothCont)
- ▷ **MaxLineDist1** (input_control) . . . . . real ⇝ *real*
  Maximum distance between a contour and the approximating line (first iteration).
  Default: 4.0
  Suggested values: MaxLineDist1 ∈ {1.0, 1.5, 2.0, 2.5, 3.0, 3.5}
  Restriction: MaxLineDist1 >= 0.0
- ▷ **MaxLineDist2** (input_control) . . . . . real ⇝ *real*
  Maximum distance between a contour and the approximating line (second iteration).
  Default: 2.0
  Suggested values: MaxLineDist2 ∈ {1.0, 1.5, 2.0, 2.5, 3.0, 3.5}
  Restriction: MaxLineDist2 >= 0.0

Example

```
read_image (Image, 'pumpe')
edges_sub_pix (Image, Edges, 'canny', 1.5, 15, 40)
segment_contours_xld (Edges, ContoursSplit, 'lines_circles', 5, 4, 2)
count_obj (ContoursSplit, Number)
gen_empty_obj (Lines)
gen_empty_obj (Circles)
for I := 1 to Number by 1
  select_obj (ContoursSplit, Contour, I)
  get_contour_global_attrib_xld (Contour, 'cont_approx', Type)
  if (Type == -1)
      concat_obj (Lines, Contour, Lines)
  else
      concat_obj (Circles, Contour, Circles)
  endif
endfor
fit_line_contour_xld (Lines, 'tukey', -1, 0, 5, 2, RowBegin, ColBegin, \
                      RowEnd, ColEnd, Nr, Nc, Dist)
fit_circle_contour_xld (Circles, 'atukey', -1, 2, 0, 3, 2, Row, Column, \
                        Radius, StartPhi, EndPhi, PointOrder)
```

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, edges_sub_pix

Possible Successors

fit_line_contour_xld, fit_ellipse_contour_xld, fit_circle_contour_xld, get_contour_global_attrib_xld

See also

split_contours_xld, get_contour_global_attrib_xld, smooth_contours_xld, gen_polygons_xld

Module

Foundation

shape_trans_xld (XLD : XLDTrans : Type :)

Transform the shape of contours or polygons.

shape_trans transforms the input contours or polygons depending on the parameter Type:

'convex' Convex hull.

'ellipse' Ellipse with the same moments and area as the input contour or polygon. The closed input contour or polygon must not intersect itself, otherwise the resulting ellipse is not meaningful(Whether the input contour or polygon intersects itself or not can be determined with test_self_intersection_xld).

'outer_circle' Smallest enclosing circle.

'rectangle1' Smallest enclosing rectangle parallel to the coordinate axes.

'rectangle2' Smallest enclosing rectangle.

Parameters

▷ **XLD** (input_object) ..... xld(-array) ⇝ *object*
Contours or polygons to be transformed.

▷ **XLDTrans** (output_object) ..... xld(-array) ⇝ *object*
Transformed contours respectively polygons.

▷ **Type** (input_control) ..... string ⇝ *string*
Type of transformation.
Default: 'convex'
List of values: Type ∈ {'convex', 'ellipse', 'outer_circle', 'rectangle1', 'rectangle2'}

Result

shape_trans_xld returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no contours given) can be set via set_system('no_object_result',<Result>). If necessary, an exception is raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Automatically parallelized on tuple level.

Possible Predecessors

gen_contours_skeleton_xld, edges_sub_pix, threshold_sub_pix, gen_contour_polygon_xld, test_self_intersection_xld

Alternatives

smallest_circle_xld, smallest_rectangle1_xld, smallest_rectangle2_xld, elliptic_axis_xld

See also

smallest_circle_xld, smallest_rectangle1_xld, smallest_rectangle2_xld, elliptic_axis_xld

Module

Foundation

smooth_contours_xld (Contours : SmoothedContours : NumRegrPoints :)

Smooth an XLD contour.

smooth_contours_xld smooths the input XLD contours Contours and returns the smoothed contours in SmoothedContours. The smoothing is done by projecting the contours' points onto a local regression line (i.e., a least-squares approximating line), which is computed from NumRegrPoints on each side of the current contour point. This operator should be called, for example, before contours are scaled.

Parameters

▷ **Contours** (input_object) ..... xld_cont-array ⇝ *object*
Contour to be smoothed.

▷ **SmoothedContours** (output_object) ..... xld_cont-array ⇝ *object*
Smoothed contour.

▷ **NumRegrPoints** (input_control) ..... integer ⇝ *integer*
Number of points used to calculate the regression line.
Default: 5
Suggested values: NumRegrPoints ∈ {3, 5, 7, 9}
Restriction: NumRegrPoints >= 3 && odd(NumRegrPoints)

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix

Possible Successors

affine_trans_contour_xld, gen_polygons_xld, local_max_contours_xld

See also

get_contour_xld

Module

Foundation

sort_contours_xld (Contours : SortedContours : SortMode, Order, RowOrCol :)

Sort contours with respect to their relative position.

The operator sort_contours_xld sorts the contours with respect to their relative position. RowOrCol specifies the sorting criteria: *'row'* sorts the contours with respect to the row coordinate of their position first and, if the row coordinates are identical, with respect to the column coordinate. In contrast, *'column'* sorts contours according to the column coordinate first. SortMode defines the position of a contour by individual reference points. The following parameter values are available:

'upper_left': The position is determined by the upper left corner of the surrounding rectangle.

'upper_right': The position is determined by the upper right corner of the surrounding rectangle.

'lower_left': The position is determined by the lower left corner of the surrounding rectangle.

'lower_right': The position is determined by the lower right corner of the surrounding rectangle.

'character': The position is determined by the upper left corner of the surrounding rectangle. In contrast to *'upper_left'*, the contours are also sorted according to the remaining coordinate, if they overlap in the direction of the coordinate which is specified by the parameter RowOrCol.

The parameter Order determines whether the sorting order is increasing or decreasing: using *'true'* the order will be increasing, using *'false'* the order will be decreasing.

Parameters

▷ **Contours** (input_object) ..... xld_cont-array ⇝ *object*
Contours to be sorted.

▷ **SortedContours** (output_object) ..... xld_cont-array ⇝ *object*
Sorted contours.

▷ **SortMode** (input_control) ..... string ⇝ *string*
Kind of sorting.
Default: 'upper_left'
List of values: SortMode ∈ {'character', 'upper_left', 'lower_left', 'upper_right', 'lower_right'}

▷ **Order** (input_control) ..... string ⇝ *string*
Increasing or decreasing sorting order.
Default: 'true'
List of values: Order ∈ {'true', 'false'}

▷ **RowOrCol** (input_control) ..... string ⇝ *string*
Sorting first with respect to row, then to column.
Default: 'row'
List of values: RowOrCol ∈ {'row', 'column'}

Result

If the parameters are correct, the operator sort_contours_xld returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

See also

sort_region

Module

Foundation

split_contours_xld (Polygons : Contours : Mode, Weight, Smooth :)

Split XLD contours at dominant points.

split_contours_xld splits the contours which were used to generate the polygons Polygons at prominent points. If the mode *'polygon'* is selected, the contours are split at the polygons' control points. In mode *'dominant'*, they are split at dominant points, i.e., at points for which the calculated change of direction is larger than the (empirically determined) threshold 2πWeight$/$*contour length*, and for which in the (empirically determined) neighborhood of $\sqrt{\text{Smooth}} * \log(\textit{contour length})$ points no larger change of direction occurs. The contour direction is determined from the direction of the regression line (i.e., the least-squares approximating line) through all points in a neighborhood of Smooth points around a contour point. The directions thus determined are smoothed with a Gaussian of width Smooth. Weight is a weighting factor for the sensitiveness of the operator. The larger Weight is selected, the less dominant points are found.

Each polygon needs a reference to a contour. If the reference is missing, because the polygon is e.g., read from a DXF-File, split_contours_xld returns an error.

Parameters

▷ **Polygons** (input_object) ..... xld_poly(-array) ⇝ *object*
Polygons for which the corresponding contours are to be split.

▷ **Contours** (output_object) ..... xld_cont(-array) ⇝ *object*
Split contours.

▷ **Mode** (input_control) ..... string ⇝ *string*
Mode for the splitting of the contours.
Default: 'polygon'
List of values: Mode ∈ {'polygon', 'dominant'}

▷ **Weight** (input_control) ..... integer ⇝ *integer*
Weight for the sensitiveness.
Default: 1

▷ **Smooth** (input_control) ..... integer ⇝ *integer*
Width of the smoothing mask.
Default: 5

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

gen_polygons_xld

Possible Successors

regress_contours_xld

See also

gen_contours_skeleton_xld, lines_gauss, lines_facet, edges_sub_pix

Module

Foundation

union_adjacent_contours_xld (Contours : UnionContours : MaxDistAbs, MaxDistRel, Mode :)

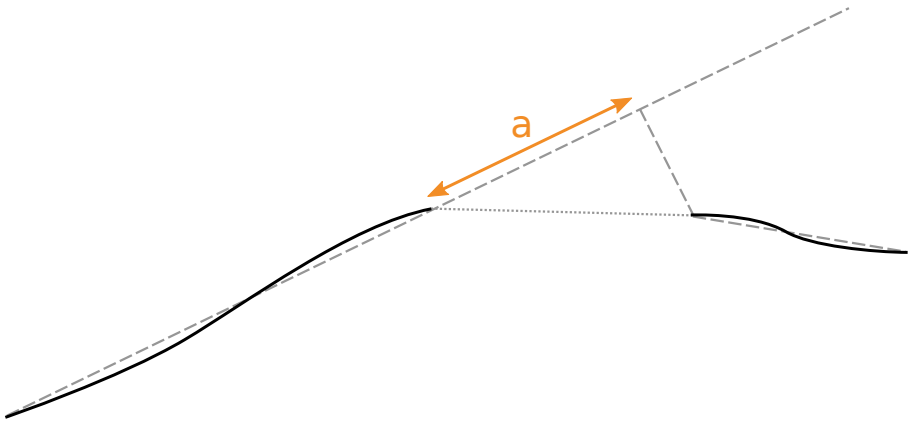
Compute the union of contours whose end points are close together.

The operator union_adjacent_contours_xld unifies all contours of the input XLD contour array (Contours) whose end points are close together. The unified contour consists of the concatenation of the contour points of the input contours. If necessary, the order of these input contour points is flipped, so that the end points of the contours that have to be connected are direct neighbors in the resulting point list. This operation is repeated until there are no more unconnected adjacent contours left. As a result all the contours that are newly created by unification, as well as the input contours that could not be connected with any other contour, are returned in UnionContours.

Parameters

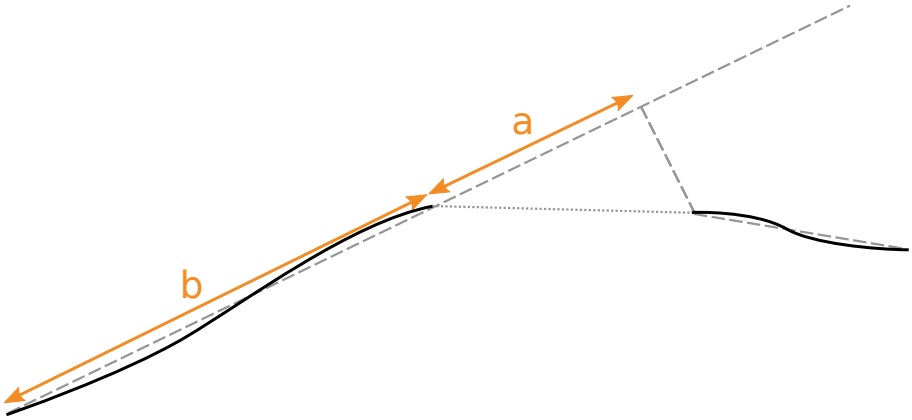
The parameters MaxDistAbs and MaxDistRel are used to define the conditions for the proximity of two contours. In principle, the measures for these parameters depend on the order in which each pair of contours is evaluated, i.e., which contour is used as reference contour that is to be joined with the second contour. To avoid this dependency, the respective measures are evaluated in both directions and the order of contours is chosen that results in the smaller value for these measures. Note that in the illustrations below, the contour on the left is always used as the reference contour. The parameter Mode controls the handling of the attributes of the input contours.

MaxDistAbs The parameter MaxDistAbs defines the maximum accepted absolute distance between the two contours. The distance is measured along the regression line of the reference contour. Thus, it is the length of the projection of the gap between the two contours onto the regression line of the reference contour.



$a \leq$ MaxDistAbs

MaxDistRel The parameter MaxDistRel defines the maximum accepted relative distance between the two contours. The relative distance is calculated by dividing the distance a (see the description of the parameter MaxDistAbs) by the length b of the reference contour.



$\frac{a}{b} \leq$ MaxDistRel

The order for connecting adjacent contours depends mainly on the distance of the neighbored end points: the contours with the smallest distance are connected first. If there are two pairs of contours with the same distance, first the contour pair that does not contain the shortest contour is connected.

Mode Finally, by the parameter Mode it is possible to control how to handle the attributes that may come with input contours. For example, the operator edges_sub_pix attaches to every contour point attributes like the local orientation, the edge response, and the edge direction. Choosing the default value *'attr_keep'*, all attributes are copied to the output, and – if a contour has to be flipped for connecting it with another one – they are adapted to the new orientation. With a great number of input contours, however, it may be sensible to ignore the attributes for performance reasons, if they are not needed for further calculations. For this the value *'attr_forget'* has to be passed.

Parameters

▷ **Contours** (input_object) ..... xld_cont-array ⇝ *object*
Input XLD contours.

▷ **UnionContours** (output_object) ..... xld_cont-array ⇝ *object*
Output XLD contours.

▷ **MaxDistAbs** (input_control) ..... real ⇝ *real*
Maximum distance of the contours' end points.
Default: 10.0
Value range: 0.0 ≤ MaxDistAbs

▷ **MaxDistRel** (input_control) ..... real ⇝ *real*
Maximum distance of the contours' end points in relation to the length of the longer contour.
Default: 1.0
Value range: 0.0 ≤ MaxDistRel

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Mode describing the treatment of the contours' attributes.
Default: 'attr_keep'
List of values: Mode ∈ {'attr_keep', 'attr_forget'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

split_contours_xld, select_contours_xld

Alternatives

union_collinear_contours_xld, union_collinear_contours_ext_xld, union_cocircular_contours_xld, union_straight_contours_xld, union_cotangential_contours_xld

See also

edges_sub_pix, threshold_sub_pix, gen_polygons_xld, split_contours_xld, select_contours_xld, get_contour_xld, get_contour_attrib_xld

Module

Foundation

```
union_cocircular_contours_xld ( Contours : UnionContours : MaxArcAngleDiff, MaxArcOverlap,
    MaxTangentAngle, MaxDist, MaxRadiusDiff, MaxCenterDist,
    MergeSmallContours, Iterations : )
```

Compute the union of contours that belong to the same circle.

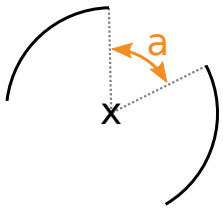
The operator union_cocircular_contours_xld merges all contours that belong to the same circle. The result is a set of contours in which contours on the same circle are connected.

The algorithm tries to merge contours by first fitting circles to each contour and then examining the result by means of radius, circle center, and gap on the circular arc. The list of contours is processed in sequence of increasing radii. Contours to which no circle could be fitted are optionally merged in a second pass.

The threshold parameters are used to define whether contours belong to the same circle. All thresholds must be fulfilled simultaneously for two contours to be merged.

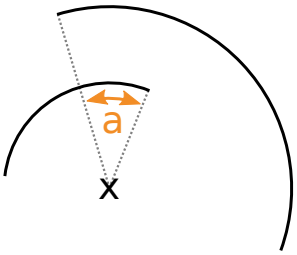
Parameters

MaxArcAngleDiff The parameter MaxArcAngleDiff defines the maximum angular distance in radians between the end point of one contour and the start point of a second contour on a circle.



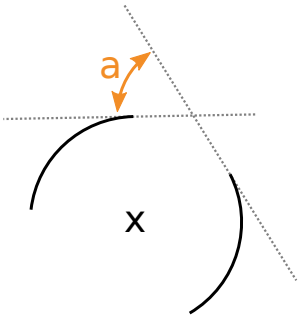
$a \leq$ MaxArcAngleDiff

MaxArcOverlap MaxArcOverlap denotes the maximum angle by which contours may overlap.



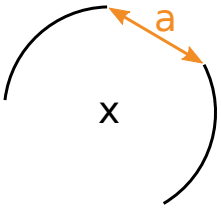
$a \leq$ MaxArcOverlap

MaxTangentAngle MaxTangentAngle describes the maximum angle between circle tangents and the connecting line of two contours.



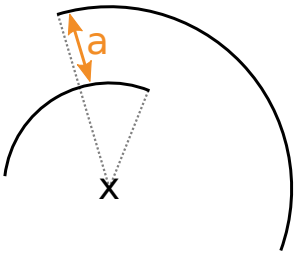
$a \leq$ MaxTangentAngle

MaxDist MaxDist denotes the maximum absolute distance in pixels of the end and start points of two contours.



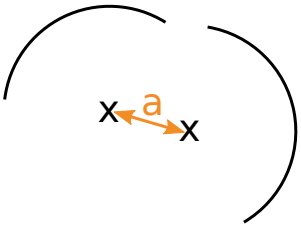
$a \leq$ MaxDist

MaxRadiusDiff MaxRadiusDiff is the maximum absolute difference of the radii of circles fitted to the contours.



$a \leq$ MaxRadiusDiff

MaxCenterDist MaxCenterDist is the limit of the Euclidean distance of the circle centers.



$a \leq$ MaxCenterDist

MergeSmallContours If the parameter MergeSmallContours is set to *'true'*, contours without fitted circles are also merged. For a small contour to match, each point of the contour needs to have a distance to

the circle center which differs from the radius by not more then MaxRadiusDiff. The angle between a line between two consecutive points on the contour and the circle tangent must be within MaxTangentAngle. Additionally, the condition for MaxDist must be fulfilled for two contours to be merged.

Iterations Merging of contours leads to new circle parameters. Therefore, an iteration can lead to further merges. The parameter Iterations controls the number of iterations. More than two iterations are seldom needed.

For each possible merge of two contours a cost is calculated by summing up the distances corresponding to the different threshold values. For comparability, the distances are scaled by the thresholds to a value between 0.0 and 1.0. If two or more contour start points match to the same end point of another contour, the contour with the lower cost is merged.

You should make sure that the input contours can be approximated by lines and circular arcs, for example by preprocessing them with split_contours_xld.

Attention

Note that already closed contours are not considered for a union anymore. That is, even if a contour and the already closed contour share the same circle, they are returned as separate contours.

Parameters

- ▷ **Contours** (input_object) . . . xld_cont-array ⇝ *object*
  Contours to be merged.
- ▷ **UnionContours** (output_object) . . . xld_cont-array ⇝ *object*
  Merged contours.
- ▷ **MaxArcAngleDiff** (input_control) . . . angle.rad ⇝ *real* / integer
  Maximum angular distance of two circular arcs.
  Default: 0.5
  Suggested values: MaxArcAngleDiff ∈ {0.25, 0.5, 0.75, 1.0}
- ▷ **MaxArcOverlap** (input_control) . . . angle.rad ⇝ *real* / integer
  Maximum overlap of two circular arcs.
  Default: 0.1
  Suggested values: MaxArcOverlap ∈ {0.0, 0.1, 0.2}
- ▷ **MaxTangentAngle** (input_control) . . . angle.rad ⇝ *real* / integer
  Maximum angle between the connecting line and the tangents of circular arcs.
  Default: 0.2
  Suggested values: MaxTangentAngle ∈ {0.1, 0.2, 0.3, 0.4, 0.5}
- ▷ **MaxDist** (input_control) . . . number ⇝ *real* / integer
  Maximum length of the gap between two circular arcs in pixels.
  Default: 30
  Suggested values: MaxDist ∈ {10, 30, 50, 70}
- ▷ **MaxRadiusDiff** (input_control) . . . number ⇝ *real* / integer
  Maximum radius difference of the circles fitted to two arcs.
  Default: 10
  Suggested values: MaxRadiusDiff ∈ {10, 20, 30}
- ▷ **MaxCenterDist** (input_control) . . . number ⇝ *real* / integer
  Maximum center distance of the circles fitted to two arcs.
  Default: 10
  Suggested values: MaxCenterDist ∈ {10, 20, 30}
- ▷ **MergeSmallContours** (input_control) . . . string ⇝ *string*
  Determine whether small contours without fitted circles should also be merged.
  Default: 'true'
  List of values: MergeSmallContours ∈ {'true', 'false'}
- ▷ **Iterations** (input_control) . . . integer ⇝ *integer*
  Number of iterations.
  Default: 1
  Suggested values: Iterations ∈ {1, 2}

Complexity

$O(n^2)$ for n contours.

Result

union_cocircular_contours_xld returns 2 (H_MSG_TRUE) if all parameters are correct.

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

split_contours_xld, select_contours_xld

Alternatives

union_collinear_contours_xld, union_cotangential_contours_xld, union_straight_contours_xld, union_adjacent_contours_xld, union_collinear_contours_ext_xld

Module

Foundation

```
union_collinear_contours_ext_xld ( Contours : UnionContours : MaxDistAbs, MaxDistRel, MaxShift,
    MaxAngle, MaxOverlap, MaxRegrError, MaxCosts, WeightDist,
    WeightShift, WeightAngle, WeightLink, WeightRegr, Mode : )
```

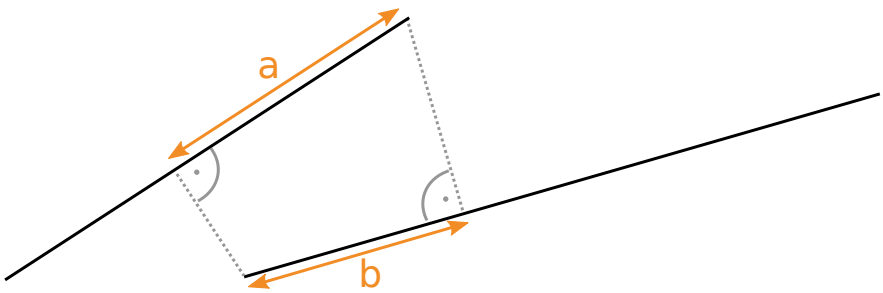
Compute the union of collinear contours (operator with extended functionality).

Like union_collinear_contours_xld, the operator union_collinear_contours_ext_xld detects all the collinear contours within the input array containing straight contours (Contours) and returns the results in UnionContours. The basic behavior is identical to the operator union_collinear_contours_xld and is described there. union_collinear_contours_ext_xld, however, provides several additional parameters to control the process of contour unification more precisely.

Parameters

The parameters MaxDistAbs, MaxDistRel, MaxShift, and MaxAngle operate as described for the operator union_collinear_contours_xld.

MaxOverlap In addition, the parameter MaxOverlap allows to define an overlap between two contours, in order to connect contours which slightly overlap. Internally, the operator assures – irrespective of the passed value for MaxOverlap – that the overlap is not bigger than a third of the length of both contours.



$min(a, b) \leq$ MaxOverlap

MaxRegrError The parameter MaxRegrError is not used at the moment. Its aim is to assure that after the unification of two contours, the resulting contour does not deviate too much from its regression line. This parameter is by default switched off (*-1*), because of its high time consumption and because of the fact that it is normally not necessary. In some cases, however, result contours may be created with points diverging to much from the regression line (winding input contours, shift and angle differences close to the – quite high adjusted – thresholds, recursive unification of a lot of small contours).

MaxCosts With the help of the parameter MaxCosts it is possible to limit the maximum costs for connecting two contours. So, not only contour pairs with one value beyond a certain threshold may be excluded from unification, but also contour pairs where all values are quite close to the limits. The costs are always normalized at a value between 0 and 1, where 1 means that all values exactly meet the threshold.

WeightDist, WeightShift, WeightAngle, WeightLink, and WeightRegr The influence of the different values on the total costs is adjusted by passing appropriate weights with the parameters WeightDist, WeightShift, WeightAngle, WeightLink, WeightRegr. Any positive number can be chosen for these weights, the resulting costs are always automatically normalized to fit the range from 0 to 1. The parameter WeightRegr is not used.

Mode The parameter Mode controls – as for union_collinear_contours_xld – how to handle the attributes of the input contours.

Parameters

▷ **Contours** (input_object) ..... xld_cont-array ⇝ *object*
Input XLD contours.

▷ **UnionContours** (output_object) ..... xld_cont-array ⇝ *object*
Output XLD contours.

▷ **MaxDistAbs** (input_control) ..... real ⇝ *real*
Maximum distance of the contours' end points in the direction of the reference regression line.
Default: 10.0
Value range: 0.0 ≤ MaxDistAbs

▷ **MaxDistRel** (input_control) ..... real ⇝ *real*
Maximum distance of the contours' end points in the direction of the reference regression line in relation to the length of the contour which is to be elongated.
Default: 1.0
Value range: 0.0 ≤ MaxDistRel

▷ **MaxShift** (input_control) ..... real ⇝ *real*
Maximum distance of the contour from the reference regression line (i.e., perpendicular to the line).
Default: 2.0
Value range: 0.0 ≤ MaxShift

▷ **MaxAngle** (input_control) ..... real ⇝ *real*
Maximum angle difference between the two contours.
Default: 0.1
Value range: 0.0 ≤ MaxAngle ≤ 0.78539816339

▷ **MaxOverlap** (input_control) ..... real ⇝ *real*
Maximum range of the overlap.
Default: 0.0
Value range: 0.0 ≤ MaxOverlap

▷ **MaxRegrError** (input_control) ..... real ⇝ *real*
Maximum regression error of the resulting contours (NOT USED).
Default: -1.0

▷ **MaxCosts** (input_control) ..... real ⇝ *real*
Threshold for reducing the total costs of unification.
Default: 1.0
Value range: 0.0 ≤ MaxCosts

▷ **WeightDist** (input_control) ..... real ⇝ *real*
Influence of the distance in the line direction on the total costs.
Default: 1.0
Value range: 0.0 ≤ WeightDist

▷ **WeightShift** (input_control) ..... real ⇝ *real*
Influence of the distance from the regression line on the total costs.
Default: 1.0
Value range: 0.0 ≤ WeightShift

▷ **WeightAngle** (input_control) ..... real ⇝ *real*
Influence of the angle difference on the total costs.
Default: 1.0
Value range: 0.0 ≤ WeightAngle

▷ **WeightLink** (input_control) . . . . . real ⇝ *real*
Influence of the line disturbance by the linking segment (overlap and angle difference) on the total costs.
Default: 1.0
Value range: 0.0 ≤ WeightLink

▷ **WeightRegr** (input_control) . . . . . real ⇝ *real*
Influence of the regression error on the total costs (NOT USED).
Default: 0.0
Value range: 0.0 ≤ WeightRegr

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Mode describing the treatment of the contours' attributes
Default: 'attr_keep'
List of values: Mode ∈ {'attr_keep', 'attr_forget'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

split_contours_xld, select_contours_xld

Alternatives

union_collinear_contours_xld, union_cocircular_contours_xld, union_cotangential_contours_xld, union_adjacent_contours_xld, union_straight_contours_xld

See also

edges_sub_pix, threshold_sub_pix, gen_polygons_xld, split_contours_xld, select_contours_xld, get_contour_xld, get_contour_attrib_xld

Module

Foundation

```
union_collinear_contours_xld ( Contours : UnionContours : MaxDistAbs, MaxDistRel, MaxShift,
    MaxAngle, Mode : )
```

Unite approximately collinear contours.

The operator union_collinear_contours_xld unites all contours of the input XLD contour array Contours that are approximately collinear, i.e., which lie approximately on the same straight line. The united contours consist of the concatenation of the contour points of the respective input contours. They are returned, together with the contours that could not be united with any other contour, in the output XLD contour array UnionContours.

Typical applications

union_collinear_contours_xld can be used to bridge gaps in extracted edges or lines, which may be caused, e.g., by locally low image contrast. It can also be used to unite collinear edge or line segments that are separated, e.g., because of junctions. Typically, union_collinear_contours_xld is part of the process of grouping and selecting extracted edges or lines to achieve meaningful entities.

Example

The following example shows the effect of union_collinear_contours_xld. On the left hand side, a set of disconnected input contours is shown. The contours that lie on the same border of the rectangle are approximately collinear. On the right hand side, the result of union_collinear_contours_xld shows that the contours that were approximately collinear have been united. Note that the contours that are approximately perpendicular to each other have not been united.

(1) (2)

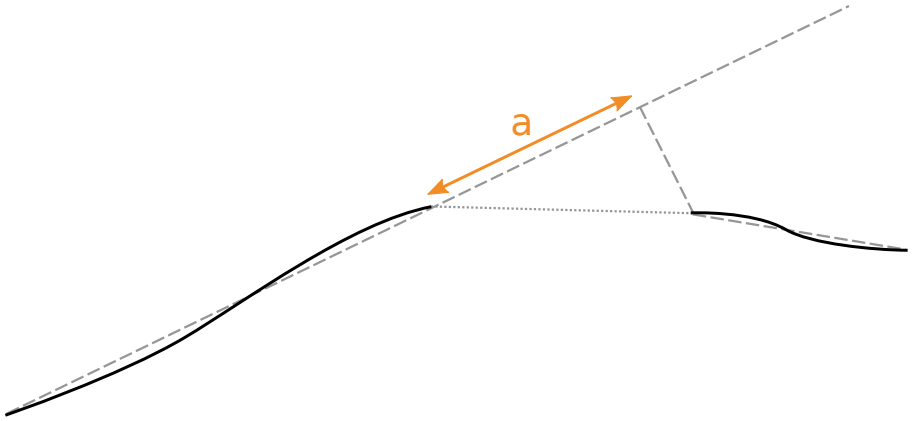
(1) Disconnected input contours. (2) Resulting contours, where approximately collinear contours have been united.

Parameters

The parameters MaxDistAbs, MaxDistRel, MaxShift, and MaxAngle are used to define the conditions for the collinearity of two contours. The parameter Mode controls the handling of the attributes of the input contours.

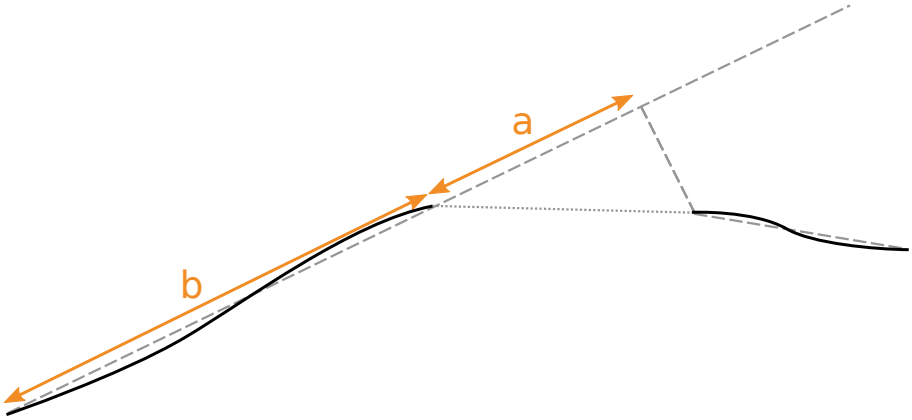
In principle, the measures for the first three parameters depend on the order in which each pair of contours is evaluated, i.e., which contour is used as reference contour that is to be joined with the second contour. To avoid this dependency, the respective measures are evaluated in both directions and the order of contours is chosen that results in the smaller value for these measures. Note that in the illustrations below, the contour on the left is always used as the reference contour.

MaxDistAbs The parameter MaxDistAbs defines the maximum accepted absolute distance between the two contours. The distance is measured along the regression line of the reference contour. Thus, it is the length of the projection of the gap between the two contours onto the regression line of the reference contour.



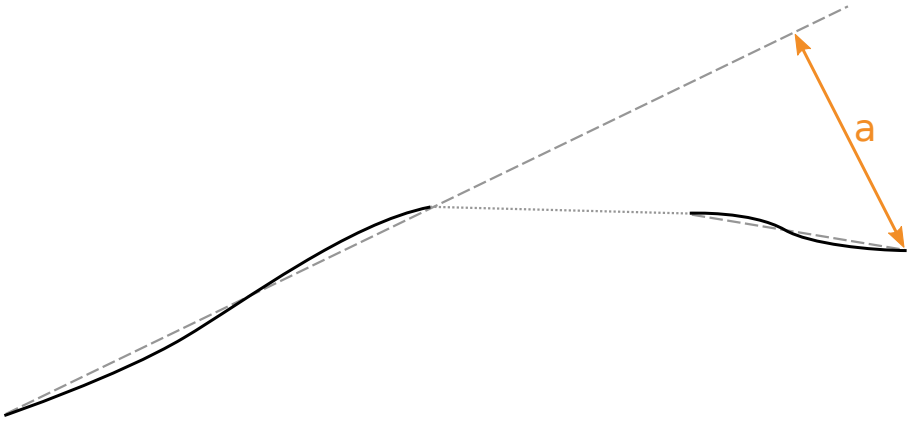
$a \leq$ MaxDistAbs

MaxDistRel The parameter MaxDistRel defines the maximum accepted relative distance between the two contours. The relative distance is calculated by dividing the distance a (see the description of the parameter MaxDistAbs) by the length b of the reference contour.



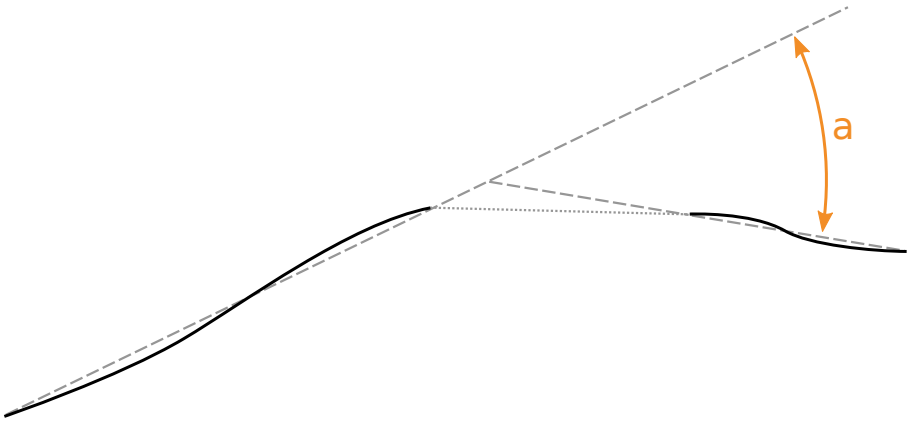
$\frac{a}{b} \leq$ MaxDistRel

MaxShift The parameter MaxShift defines the maximum distance of the second contour from the regression line of the reference contour. This distance is measured perpendicular to the regression line of the reference contour.



$a \leq$ MaxShift

MaxAngle The parameter MaxAngle defines the maximum angle (in radians) between the regression lines of the two contours.



$a \leq$ MaxAngle

Mode The parameter Mode controls the handling of the attributes that may be available for the input contours, e.g., if the contours have been created by edges_sub_pix.

'attr_keep' All attributes are copied to the output contours.
Global attributes are only kept for those contours that are not united with other contours. In general, it is not possible to determine the global attributes of a united contour from the global attributes of the individual contours.

'attr_forget' The output contours will not contain any attributes. This mode may be chosen for performance reasons if the attributes will not be required for further calculations.

To query, which attributes are available, query_contour_attribs_xld can be used.

Implementation details

The input contours are analyzed and united pairwise. This pairwise process is repeated until there are no more unconnected collinear contours left. If necessary, the order of the input contour points is flipped, so that the end points of the contours that have been connected are direct neighbors in the resulting point list.

Two contours are united only if all criteria are fulfilled, i.e., all values must be lower than the specified thresholds. As another precondition, contours must not overlap each other significantly (see union_collinear_contours_ext_xld for an illustration of overlap). An overlap of at most 0.5 pixels is tolerated. To allow a larger overlap, union_collinear_contours_ext_xld can be used.

Besides finding pairs of contours to be united, it is important for the behavior of union_collinear_contours_xld to determine the order in which contours are united. For this, all contour pairs that may be united are rated by calculating costs for their unification. The costs are the sum of the measures, each normalized by the respective threshold, defined by the parameters MaxDistAbs, MaxDistRel,

MaxShift, and MaxAngle. The contour pair with the lowest costs is united first. Then, the costs are updated with respect to the newly created contour and the contour list is searched for the next best pair of contours to be united.

Note that the definition of collinearity as used in union_collinear_contours_xld depends on the regression lines of the input contours. To avoid unexpected results, make sure that the input contours are approximately straight lines, e.g., by splitting them into line segments with segment_contours_xld. Note also that closed contours are not united with any other contour.

Limitations and alternatives

union_collinear_contours_xld can only be used if the contours that should be joined lie approximately on a straight line. If the contours lie approximately on a circle, union_cocircular_contours_xld can be used instead. If the contours represent a free-form, union_cotangential_contours_xld can be used. If only very small gaps between the contours should be closed, regardless of the orientation of the contours, union_adjacent_contours_xld can be used.

If the parameter adjustments available with union_collinear_contours_xld are not sufficient, union_collinear_contours_ext_xld can be used to define the criteria for the unification more specifically. Among others, union_collinear_contours_ext_xld allows to set the tolerance for the overlap of contours as well as a limit for the total costs to avoid the connection of contours that are close to the limits of all criteria.

Parameters

▷ **Contours** (input_object) . . . . . xld_cont-array ⇝ *object*
Input XLD contours.

▷ **UnionContours** (output_object) . . . . . xld_cont-array ⇝ *object*
Output XLD contours.

▷ **MaxDistAbs** (input_control) . . . . . real ⇝ *real*
Maximum length of the gap between two contours, measured along the regression line of the reference contour.
Default: 10.0
Value range: 0.0 ≤ MaxDistAbs

▷ **MaxDistRel** (input_control) . . . . . real ⇝ *real*
Maximum length of the gap between two contours, relative to the length of the reference contour, both measured along the regression line of the reference contour.
Default: 1.0
Value range: 0.0 ≤ MaxDistRel

▷ **MaxShift** (input_control) . . . . . real ⇝ *real*
Maximum distance of the second contour from the regression line of the reference contour.
Default: 2.0
Value range: 0.0 ≤ MaxShift

▷ **MaxAngle** (input_control) . . . . . real ⇝ *real*
Maximum angle between the regression lines of two contours.
Default: 0.1
Value range: 0.0 ≤ MaxAngle ≤ 0.78539816339

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Mode that defines the treatment of contour attributes, i.e., if the contour attributes are kept or discarded.
Default: 'attr_keep'
List of values: Mode ∈ {'attr_keep', 'attr_forget'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

edges_sub_pix, lines_gauss, segment_contours_xld, select_contours_xld, select_shape_xld, split_contours_xld, threshold_sub_pix

Possible Successors

fit_line_contour_xld, select_contours_xld, select_shape_xld

Alternatives

union_collinear_contours_ext_xld, union_straight_contours_xld, union_cocircular_contours_xld, union_cotangential_contours_xld, union_adjacent_contours_xld

See also

get_contour_xld, get_contour_attrib_xld, query_contour_attribs_xld

Module

Foundation

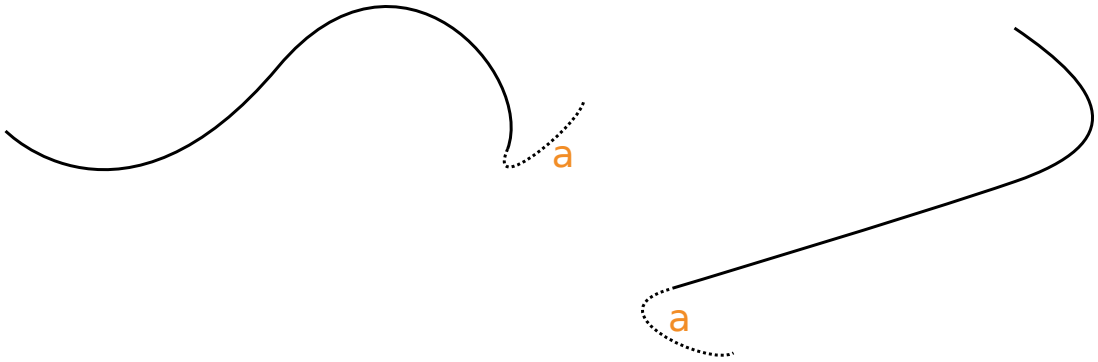
union_cotangential_contours_xld (Contours : UnionContours : FitClippingLength, FitLength, MaxTangAngle, MaxDist, MaxDistPerp, MaxOverlap, Mode :)

Compute the union of cotangential contours.

The operator union_cotangential_contours_xld unifies contours of the input XLD contour array Contours according to the local curvatures at their endpoints.

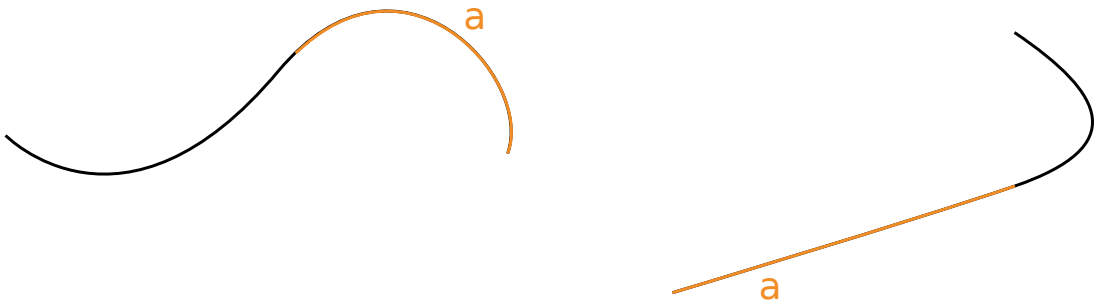
Parameters

FitClippingLength How does the union work? First, end pieces for each input contour in Contours are determined. The parameter FitClippingLength specifies the Euclidean length that is temporarily clipped at both ends of a contour. With this parameter, it is possible to ignore artifacts at the endings of the input contours.



$a \leq$ FitClippingLength

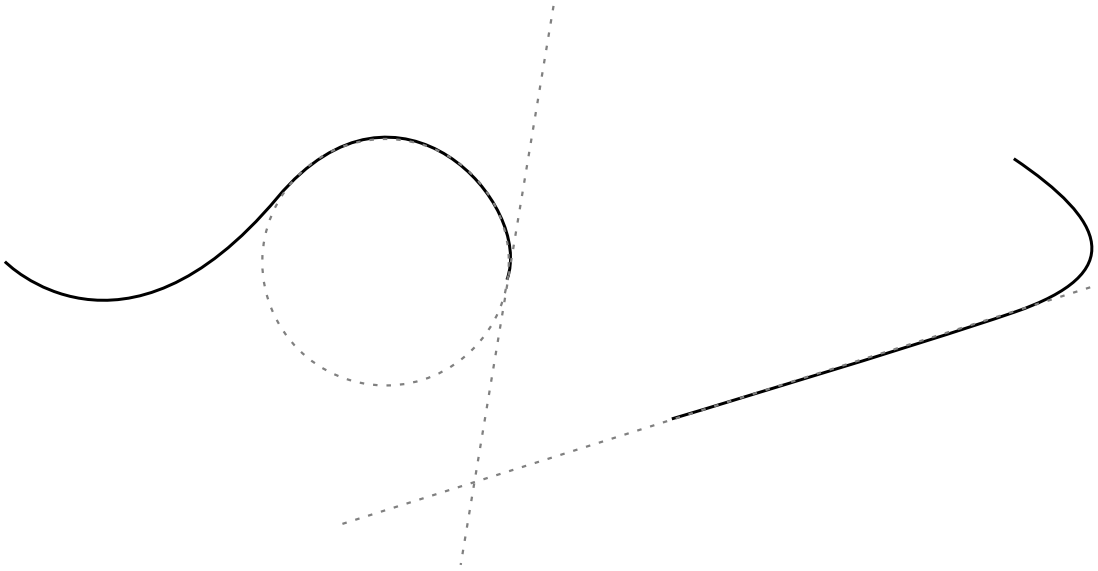
FitLength With the parameter FitLength the Euclidean length of the end pieces that will be examined for the union is given. At each end of a contour at least that many points are assigned to an end piece that its length is greater or equal to FitLength. The value *'auto'* can be used here. The operator then tries to segment the contours into straights and arcs and takes the first and last segment as end pieces. As this is computationally intensive, it is recommended to use numerically specified values for the lengths whenever possible.



$a \leq$ FitLength

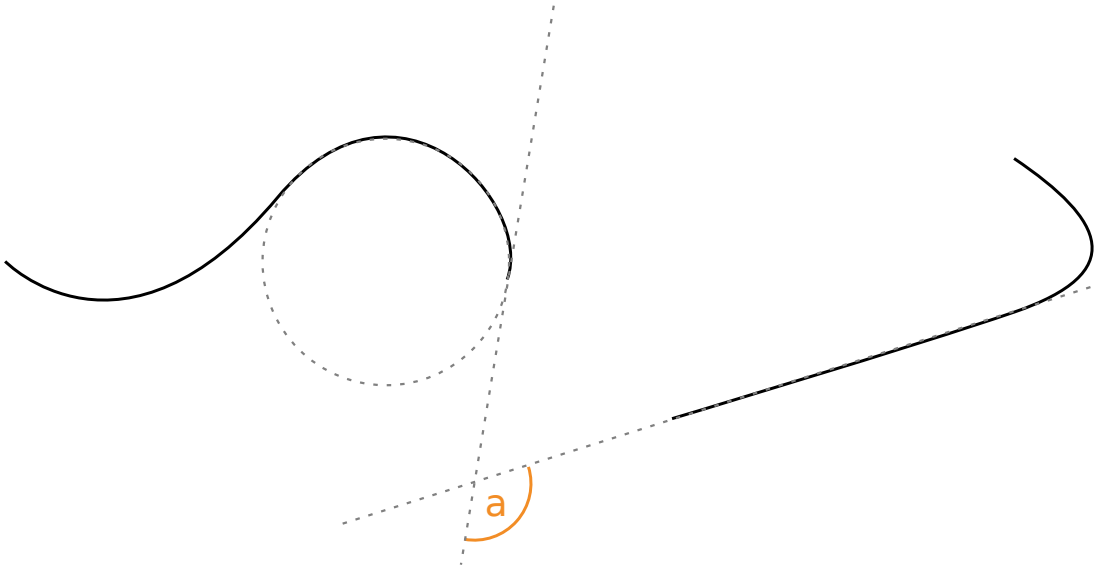
The operator tries to fit a circle into each end piece. Then, the endpoints of the input contours are determined that lie near enough to these circles (measured against the root mean square error of the circle fit). The tangents at these points correspond to the local curvature at the endings of the input contours.

If a circle fit fails, the operator tries to fit a regression line. Then, the local curvature of the endings of a contour corresponds to the direction of the regression line through the points of the end pieces.



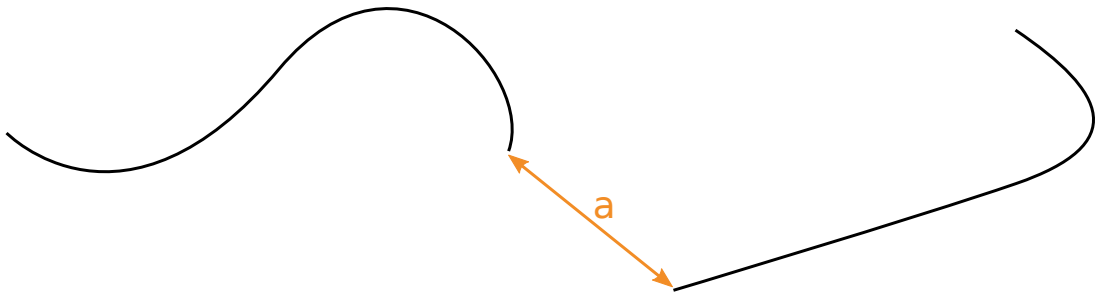
The input contours are going to be unified according to the calculated local curvatures at their endpoints. Two contours are considered for the union if multiple thresholds are fulfilled at the same time:

MaxTangAngle The parameter MaxTangAngle specifies the maximum angle between the tangents of two contours.



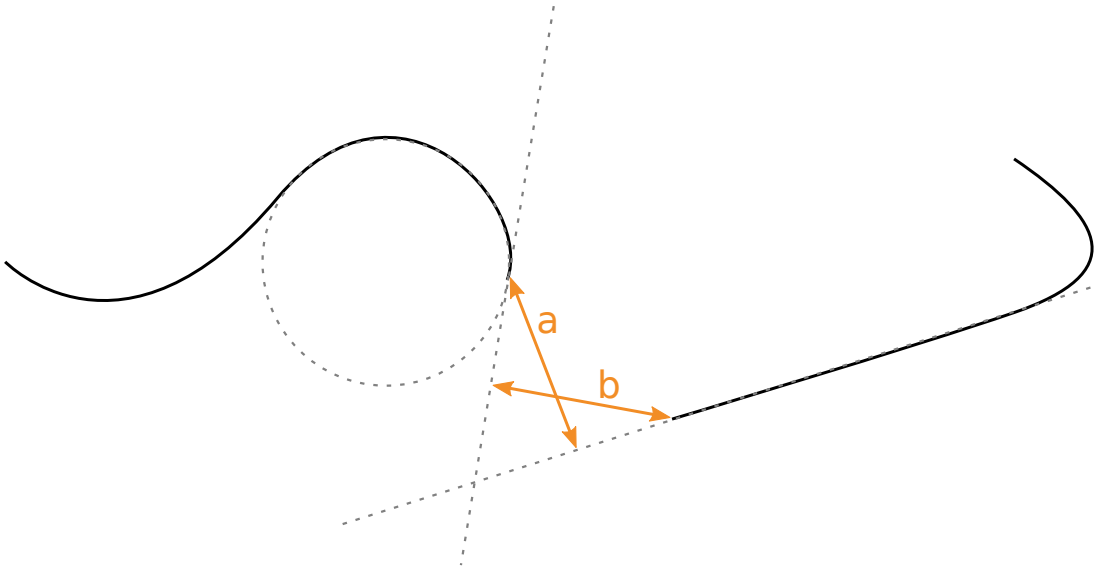
$a \leq$ MaxTangAngle

MaxDist The maximum distance of two endpoints is given with the parameter MaxDist.



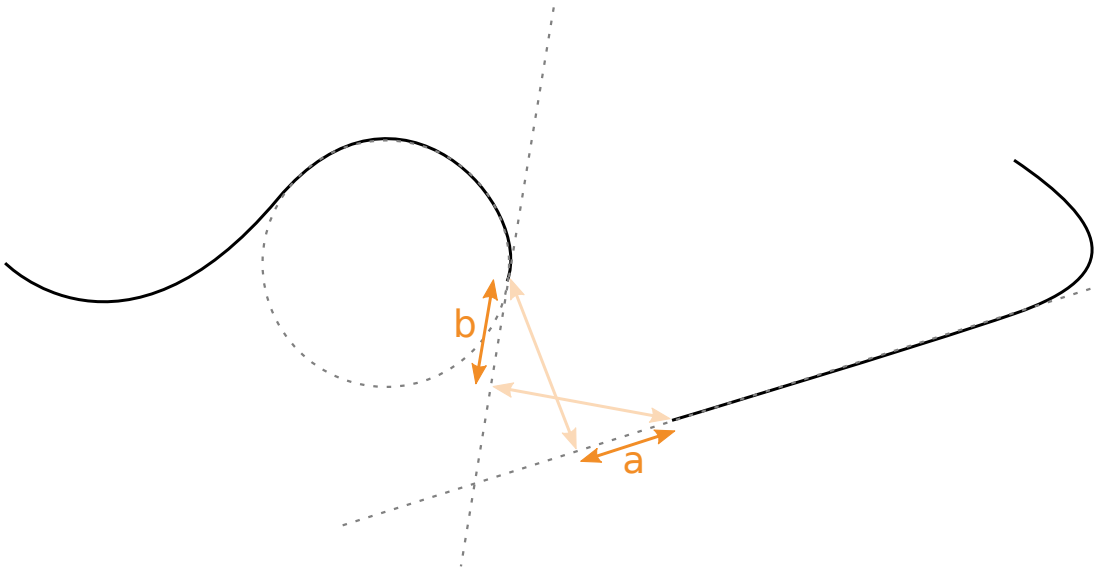
$a \leq$ MaxDist

MaxDistPerp The maximum distance perpendicular to the tangents is specified with the parameter MaxDistPerp. This distance is the minimum of the distances of the endpoints of the contours perpendicular to the other contour.



$min(a, b) \leq$ MaxDistPerp

MaxOverlap As a last threshold, the overlap of the contours is considered. The amount of the overlap is determined by the projection of one endpoint onto the tangent of another contour and vice versa. The distances between the endpoints and their projections must be shorter than MaxOverlap. Contrary to the other threshold values, MaxOverlap can become negative. Negative values denote the minimum distance of the endpoints in the direction of the tangents.



$min(a, b) \leq$ MaxOverlap. In this case, a and b are negative. See the reference manual entry of union_collinear_contours_ext_xld for an illustration with positive values for a and b.

It is possible, that a single contour is closed by this operator or multiple contours are merged into a single closed contour, if the criteria named above are fulfilled for the end pieces. Be aware that already closed contours are not considered for a union anymore.

If all thresholds are fulfilled between contours, a score value is calculated. The closer the calculated values for the angles and distances are to 0, the higher is the score value. The unification is then executed in the order of the score values. Thus, if there are multiple candidates for the unification, the best fitting contours are merged first.

The parameter Mode states, whether existing attributes of the input contours should be copied to the output contours. *'attr_keep'* copies the attributes, which can have negative impacts on the performance of the operator.

'attr_forget' causes the operator to ignore existing attributes. For further details see the description of the parameter Mode of the operator union_adjacent_contours_xld.

Parameters

▷ **Contours** (input_object) . . . . . xld_cont-array ⇝ *object*
Input XLD contours.

▷ **UnionContours** (output_object) . . . . . xld_cont-array ⇝ *object*
Output XLD contours.

▷ **FitClippingLength** (input_control) . . . . . real ⇝ *real*
Length of the part of a contour to skip for the determination of tangents.
Default: 0.0
Value range: 0.0 ≤ FitClippingLength

▷ **FitLength** (input_control) . . . . . real ⇝ *real* / string
Length of the part of a contour to use for the determination of tangents.
Default: 30.0
Suggested values: FitLength ∈ {10.0, 20.0, 30.0, 'auto'}
Value range: 0.0 ≤ FitLength

▷ **MaxTangAngle** (input_control) . . . . . angle.rad ⇝ *real*
Maximum angle difference between two contours' tangents.
Default: 0.78539816
Value range: 0.0 ≤ MaxTangAngle ≤ 3.1415926

▷ **MaxDist** (input_control) . . . . . real ⇝ *real*
Maximum distance of the contours' end points.
Default: 25.0
Suggested values: MaxDist ∈ {5.0, 10.0, 25.0, 50.0}
Value range: 0.0 ≤ MaxDist

▷ **MaxDistPerp** (input_control) . . . . . real ⇝ *real*
Maximum distance of the contours' end points perpendicular to their tangents.
Default: 10.0
Suggested values: MaxDistPerp ∈ {2.0, 5.0, 10.0, 20.0}
Value range: 0.0 ≤ MaxDistPerp

▷ **MaxOverlap** (input_control) . . . . . real ⇝ *real*
Maximum overlap of two contours.
Default: 2.0
Suggested values: MaxOverlap ∈ {2.0, 5.0, 10.0, 20.0}

▷ **Mode** (input_control) . . . . . string ⇝ *string*
Mode describing the treatment of the contours' attributes.
Default: 'attr_forget'
List of values: Mode ∈ {'attr_keep', 'attr_forget'}

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

split_contours_xld, select_contours_xld

Alternatives

union_collinear_contours_xld, union_collinear_contours_ext_xld, union_cocircular_contours_xld, union_straight_contours_xld, union_adjacent_contours_xld

See also

edges_sub_pix, threshold_sub_pix, gen_polygons_xld, get_contour_xld, get_contour_attrib_xld

Module

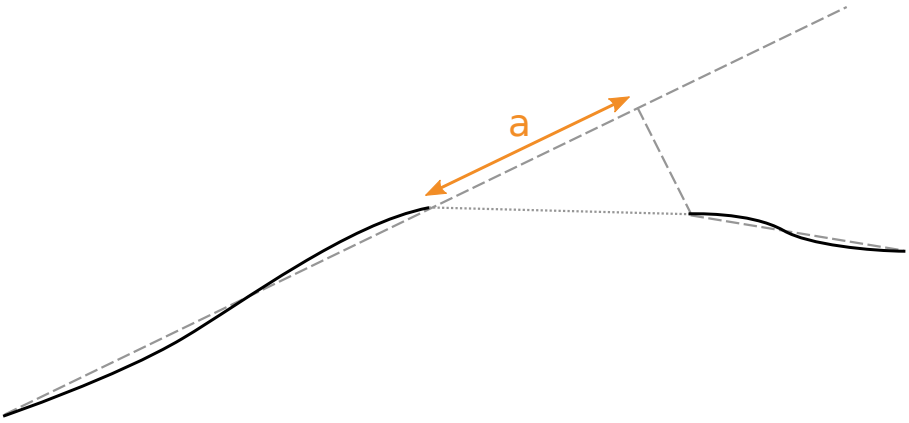
Foundation

union_straight_contours_xld (Contours : UnionContours : MaxDist, MaxDiff, Percent, Mode, Iterations :)

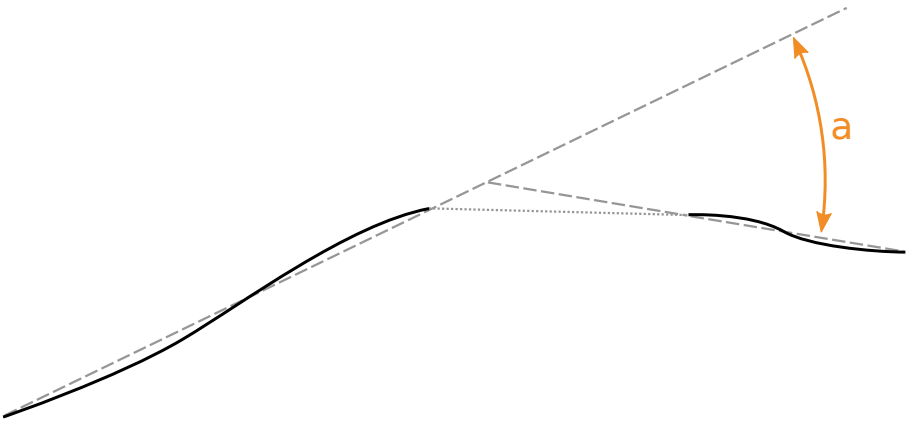
Compute the union of neighboring straight contours that have a similar direction.

union_straight_contours_xld merges neighboring XLD contours Contours if certain criteria are fulfilled. At each iteration, at most two contours that fulfill the given criteria are merged. The parameter Iterations controls how often this step is executed.

Two contours are merged if the shortest distance between their end points (end points are the projections of the contours' first and last points onto its regression line) is smaller than MaxDist, and if the difference in direction (i.e., the regression lines' direction) is smaller than MaxDiff (radians).



$a \leq$ MaxDist



$a \leq$ MaxDiff

If only one of the criteria is fulfilled, the decision on merging can be influenced by the weighting factor Percent, which allows the exceeding of one limit to be balanced by the other value remaining below the limit by the same amount. The end point distance is weighted by Percent, while the directional difference is weighted by $100 -$ Percent.

This means that two contours are merged if they fulfill the following condition:

$$\frac{shortest\ distance}{\texttt{MaxDist}} \cdot \texttt{Percent} + \frac{direction\ difference}{\texttt{MaxDiff}} \cdot (100 - \texttt{Percent}) \leq 100$$

If, for example, two contours have an end point distance of 5.0 and a directional difference of 0.5 (with the limits chosen being MaxDist = *4.0* and MaxDiff = *0.625*), both values differ from the limits by 25%. By choosing Percent = *60%*, the larger end point distance is weighted more than the smaller directional difference, and thus the contours are not merged. Contrary, if Percent = *40%* is chosen, the contours are merged.

For Percent = *100%*, only the end point distance is taken into account, while for Percent = *0%* only the difference of direction is used. If Percent = *50%* both criteria are equally valid.

In case there are parallel contours, there is a danger of merging neighboring contours. If this is to be avoided, Mode = *'noparallel'* has to be chosen, while otherwise Mode = *'paralleltoo'* suffices. For Mode = *'every'*, contours are merged unconditionally. All other parameters have no influence in this case.

The parameters of the regression line are calculated anew for merged contours.

Attention

Before contours can be united by `union_straight_contours_xld`, the parameters of the regression lines to the contours must be calculated by calling `regress_contours_xld`.

Furthermore, note that `union_straight_contours_xld` can not compute the union of contours with different contour point attributes. This can be the case if different operators are used for the contour extraction (e.g., `lines_gauss` and `lines_facet`). For information on the contour attributes and when they are defined, see the respective operator reference. You can query the point attributes of your contour using `query_contour_attribs_xld`.

Parameters

- ▷ **`Contours`** (input_object) ..... xld_cont-array ⇝ *object*
  Input XLD contours.
- ▷ **`UnionContours`** (output_object) ..... xld_cont-array ⇝ *object*
  Output XLD contours.
- ▷ **`MaxDist`** (input_control) ..... real ⇝ *real*
  Maximum distance of the contours' endpoints.
  Default: 5.0
- ▷ **`MaxDiff`** (input_control) ..... angle.rad ⇝ *real*
  Maximum difference in direction.
  Default: 0.5
- ▷ **`Percent`** (input_control) ..... real ⇝ *real*
  Weighting factor for the two selection criteria.
  Default: 50.0
- ▷ **`Mode`** (input_control) ..... string ⇝ *string*
  Should parallel contours be taken into account?
  Default: 'noparallel'
  List of values: `Mode` ∈ {'noparallel', 'paralleltoo', 'every'}
- ▷ **`Iterations`** (input_control) ..... string ⇝ *string* / integer
  Number of iterations or 'maximum'.
  Default: 'maximum'
  Suggested values: `Iterations` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'maximum'}
  Value range: 1 ≤ `Iterations` ≤ 500 (lin)
  Minimum increment: 1
  Recommended increment: 1

Execution Information

- Multithreading type: reentrant (runs in parallel with non-exclusive operators).
- Multithreading scope: global (may be called from any thread).
- Processed without parallelization.

Possible Predecessors

`regress_contours_xld`

Alternatives

`union_collinear_contours_xld`, `union_collinear_contours_ext_xld`, `union_cocircular_contours_xld`, `union_cotangential_contours_xld`, `union_adjacent_contours_xld`

See also

`fit_line_contour_xld`, `get_contour_xld`, `get_contour_attrib_xld`, `gen_contours_skeleton_xld`, `lines_gauss`, `lines_facet`, `edges_sub_pix`, `get_regress_params_xld`, `get_contour_global_attrib_xld`, `query_contour_global_attribs_xld`

Module

Foundation

Index