

---

## HALCON Application Note

# How to Use Shape-Based Matching to Find and Localize Objects

---

### Provided Functionality

- ▷ Finding objects starting based on a single model image
- ▷ Localizing objects with subpixel accuracy

### Typical Applications

- ▷ Object recognition and localization
- ▷ Intermediate machine vision steps, e.g., alignment of ROIs
- ▷ Completeness check
- ▷ Parts inspection

### Involved Operators

```
create_shape_model, create_scaled_shape_model  
inspect_shape_model, get_shape_model_params  
get_shape_model_contours, set_shape_model_origin, get_shape_model_origin  
find_shape_model, find_shape_models  
find_scaled_shape_model, find_scaled_shape_models  
write_shape_model, read_shape_model  
clear_shape_model, clear_all_shape_models
```

# Overview

HALCON's operators for *shape-based matching* enable you to find and localize objects based on a single model image, i.e., from a *model*. This method is robust to noise, clutter, occlusion, and arbitrary non-linear illumination changes. Objects are localized with subpixel accuracy in 2D, i.e., found even if they are rotated or scaled.

The process of shape-based matching (see [section 1](#) on page 4 for a quick overview) is divided into two distinct phases: In a first phase, you specify and create the model. This model can be stored in a file to be reused in different applications. Detailed information about this phase can be found in [section 2](#) on page 6. In the second phase, the model is used to find and localize an object. [Section 3](#) on page 20 describes how to optimize the outcome of this phase by restricting the search space.

Shape-based matching is a powerful tool for various machine vision tasks, ranging from intermediate image processing, e.g., to place ROIs automatically or to align them to a moving part, to complex tasks, e.g., recognize and localize a part in a robot vision application. Examples can be found in [section 4](#) on page 30.

Unless specified otherwise, the example programs can be found in the subdirectory `shape_matching` of the directory `%HALCONROOT%\examples\application_guide`.

---

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

Edition 1	June 2002	(HALCON 6.1)
Edition 1a	May 2003	(HALCON 6.1.2)
Edition 2	December 2003	(HALCON 7.0)
Edition 2a	April 2005	(HALCON 7.0.2)

Microsoft, Windows, Windows NT, Windows 2000, and Windows XP are either trademarks or registered trademarks of Microsoft Corporation.

All other nationally and internationally recognized trademarks and tradenames are hereby recognized.

More information about HALCON can be found at:

<http://www.mvtec.com/halcon/>

# Contents

<b>1</b>	<b>A First Example</b>	<b>4</b>
<b>2</b>	<b>Creating a Suitable Model</b>	<b>6</b>
2.1	A Closer Look at the Region of Interest . . . . .	6
2.2	Which Information is Stored in the Model? . . . . .	12
2.3	Synthetic Model Images . . . . .	18
<b>3</b>	<b>Optimizing the Search Process</b>	<b>20</b>
3.1	Restricting the Search Space . . . . .	20
3.2	Searching for Multiple Instances of the Object . . . . .	23
3.3	Searching for Multiple Models Simultaneously . . . . .	24
3.4	A Closer Look at the Accuracy . . . . .	26
3.5	How to Optimize the Matching Speed . . . . .	28
<b>4</b>	<b>Using the Results of Matching</b>	<b>30</b>
4.1	Introducing Affine Transformations . . . . .	30
4.2	Creating and Applying Affine Transformations With HALCON . . . . .	30
4.3	Using the Estimated Position and Orientation . . . . .	32
4.4	Using the Estimated Scale . . . . .	43
<b>5</b>	<b>Miscellaneous</b>	<b>45</b>
5.1	Adapting to a Changed Camera Orientation . . . . .	45
5.2	Reusing Models . . . . .	45

# 1 A First Example

In this section we give a quick overview of the matching process. To follow the example actively, start the HDevelop program `hdevelop\first_example_shape_matching.dev`, which locates the print on an IC; the steps described below start after the initialization of the application (press Run once to reach this point).

## Step 1: Select the object in the model image

```
Row1 := 188
Column1 := 182
Row2 := 298
Column2 := 412
gen_rectangle1 (ROI, Row1, Column1, Row2, Column2)
reduce_domain (ModelImage, ROI, ImageROI)
```

After grabbing the so-called *model image*, i.e., a representative image of the object to find, the first task is to create a region containing the object. In the example program, a rectangular region is created using the operator `gen_rectangle1`; alternatively, you can draw the region interactively using, e.g., `draw_rectangle1` or use a region that results from a previous segmentation process. Then, an image containing just the selected region is created using the operator `reduce_domain`. The result is shown in figure 1.

## Step 2: Create the model

```
inspect_shape_model (ImageROI, ShapeModelImages, ShapeModelRegions, 8, 30)
create_shape_model (ImageROI, NumLevels, 0, rad(360), 0, 'none',
                  'use_polarity', 30, 10, ModelID)
```

With the operator `create_shape_model`, the so-called *model* is created, i.e., the internal data structure describing the searched object. Before this, we recommend to apply the operator `inspect_shape_model`, which helps you to find suitable parameters for the model creation. `in-`

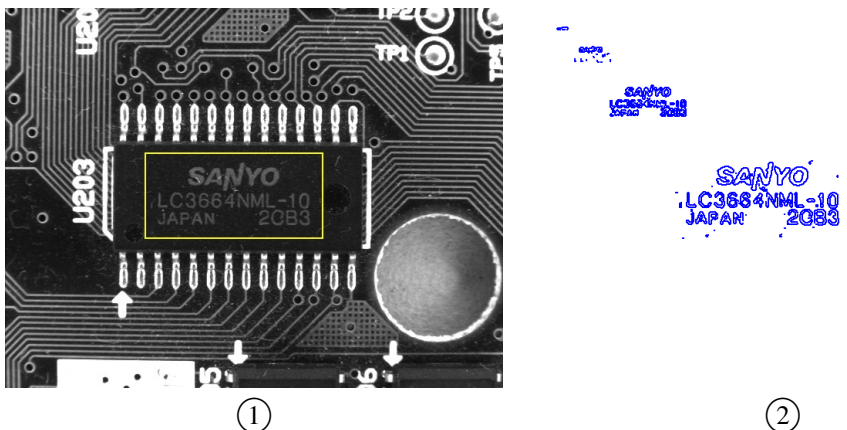


Figure 1: ① specifying the object; ② the internal model (4 pyramid levels).

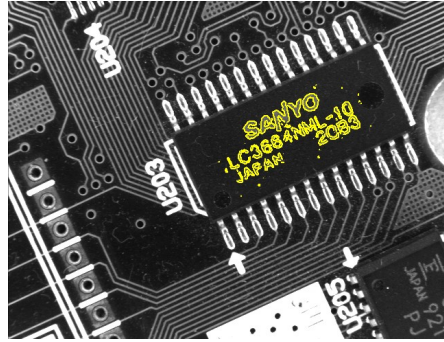


Figure 2: Finding the object in other images.

`spect_shape_model` shows the effect of two parameters: the *number of pyramid levels* on which the model is created, and the minimum *contrast* that object points must have to be included in the model. As a result, the operator `inspect_shape_model` returns the model points on the selected pyramid levels as shown in [figure 1](#); thus, you can check whether the model contains the relevant information to describe the object of interest.

When actually creating the model with the operator `create_shape_model`, you can specify additional parameters besides `NumLevels` and `Contrast`: First of all, you can restrict the range of angles the object can assume (parameters `AngleStart` and `AngleExtent`) and the angle steps at which the model is created (`AngleStep`). With the help of the parameter `Optimization` you can reduce the number of model points; this is useful in the case of very large models. The parameter `Metric` lets you specify whether the *polarity* of the model points must be observed. Finally, you can specify the minimum contrast object points must have in the *search images* to be compared with the model (`MinContrast`). The creation of the model is described in detail in [section 2](#).

As a result, the operator `create_shape_model` returns a *handle* for the newly created model (`ModelID`), which can then be used to specify the model, e.g., in calls to the operator `find_shape_model`. Note that if you use HALCON's COM or C++ interface and call the operator via the classes `HShapeModelX` or `HShapeModel`, no handle is returned because the instance of the class itself acts as your handle.

If not only the orientation but also the scale of the searched object is allowed to vary, you must use the operator `create_scaled_shape_model` to create the model; then, you can describe the allowed range of scaling with three parameters similar to the range of angles.

### Step 3: Find the object again

```
for i := 1 to 20 by 1
  grab_image (SearchImage, FGHandle)
  find_shape_model (SearchImage, ModelID, 0, rad(360), 0.7, 1, 0.5,
    'interpolation', 0, 0.9, RowCheck, ColumnCheck,
    AngleCheck, Score)
endfor
```

To find the object again in a *search image*, all you need to do is call the operator `find_shape_model`; [figure 2](#) shows the result for one of the example images. Besides the already mentioned `ModelID`,

`find_shape_model` provides further parameters to optimize the search process: The parameters `AngleStart`, `AngleExtent`, and `NumLevels`, which you already specified when creating the model, allow you to use more restrictive values in the search process; by using the value 0 for `NumLevels`, the value specified when creating the model is used. With the parameter `MinScore` you can specify how many of the model points must be found; a value of 0.5 means that half of the model must be found. Furthermore, you can specify how many instances of the object are expected in the image (`NumMatches`) and how much two instances of the object may overlap in the image (`MaxOverlap`). To compute the position of the found object with subpixel accuracy the parameter `SubPixel` should be set to a value different from 'none'. Finally, the parameter `Greediness` describes the used search heuristics, ranging from “safe but slow” (value 0) to “fast but unsafe” (value 1). How to optimize the search process is described in detail in [section 3](#) on page 20.

The operator `find_shape_model` returns the position and orientation of the found object instances in the parameters `Row`, `Column`, and `Angle`, and their corresponding `Score`, i.e., how much of the model was found.

If you use the operator `find_scaled_shape_model` (after creating the model using `create_scaled_shape_model`), the scale of the found object is returned `Scale`.

## 2 Creating a Suitable Model

A prerequisite for a successful matching process is, of course, a suitable model for the object you want to find. A model is suitable if it describes the *significant* parts of the object, i.e., those parts that characterize it and allow to discriminate it clearly from other objects or from the background. On the other hand, the model should not contain clutter, i.e., points not belonging to the object (see, e.g., [figure 4](#)).

### 2.1 A Closer Look at the Region of Interest

When creating the model, the first step is to select a *region of interest* (ROI), i.e., the part of the image which serves as the model. In HALCON, a *region* defines an area in an image or, more generally, a set of points. A region can have an arbitrary shape; its points do not even need to be connected. Thus, the region of the model can have an arbitrary shape as well.

The sections below describe how to create simple and more complex regions. The following code fragment shows the typical next steps after creating an ROI:

```
reduce_domain (ModelImage, ROI, ImageROI)
create_shape_model (ImageROI, 0, 0, rad(360), 0, 'none', 'use_polarity',
                  30, 10, ModelID)
```

Note that the region of interest used when creating a shape model influences the matching results: Its center of gravity is used as the *reference point* of the model (see [section 2.1.4](#) on page 12 for more information).

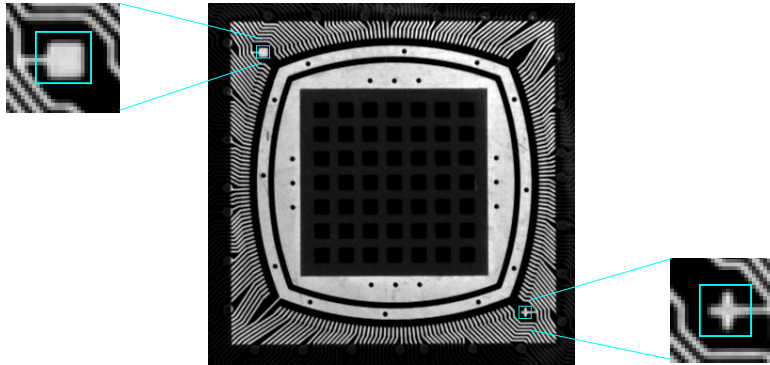


Figure 3: Creating an ROI from two regions.

### 2.1.1 How to Create a Region

HALCON offers multiple operators to create regions, ranging from standard shapes like rectangles ([gen\\_rectangle2](#)) or ellipses ([gen\\_ellipse](#)) to free-form shapes (e.g., [gen\\_region\\_polygon\\_filled](#)). These operators can be found in the HDevelop menu Operators > Regions > Creation.

However, to use these operators you need the “parameters” of the shape you want to create, e.g., the position, size and, orientation of a rectangle or the position and radius of a circle. Therefore, they are typically combined with the operators in the HDevelop menu Operators > Graphics > Drawing, which let you draw a shape on the displayed image and then return the shape parameters:

```
draw_rectangle1 (WindowHandle, ROI1Row1, ROI1Column1, ROI1Row2, ROI1Column2)
gen_rectangle1 (ROI1, ROI1Row1, ROI1Column1, ROI1Row2, ROI1Column2)
```

### 2.1.2 How to Combine and Mask Regions

You can create more complex regions by adding or subtracting standard regions using the operators [union2](#) and [difference](#). For example, to create an ROI containing the square and the cross in [figure 3](#), the following code fragment was used:

```
draw_rectangle1 (WindowHandle, ROI1Row1, ROI1Column1, ROI1Row2,
                 ROI1Column2)
gen_rectangle1 (ROI1, ROI1Row1, ROI1Column1, ROI1Row2, ROI1Column2)
draw_rectangle1 (WindowHandle, ROI2Row1, ROI2Column1, ROI2Row2,
                 ROI2Column2)
gen_rectangle1 (ROI2, ROI2Row1, ROI2Column1, ROI2Row2, ROI2Column2)
union2 (ROI1, ROI2, ROI)
```

Similarly, you can subtract regions using the operator [difference](#). This method is useful to “mask” those parts of a region containing clutter, i.e., high-contrast points that are not part of the object. In [figure 4](#), e.g., the task is to find the three capacitors. When using a single circular ROI, the created model

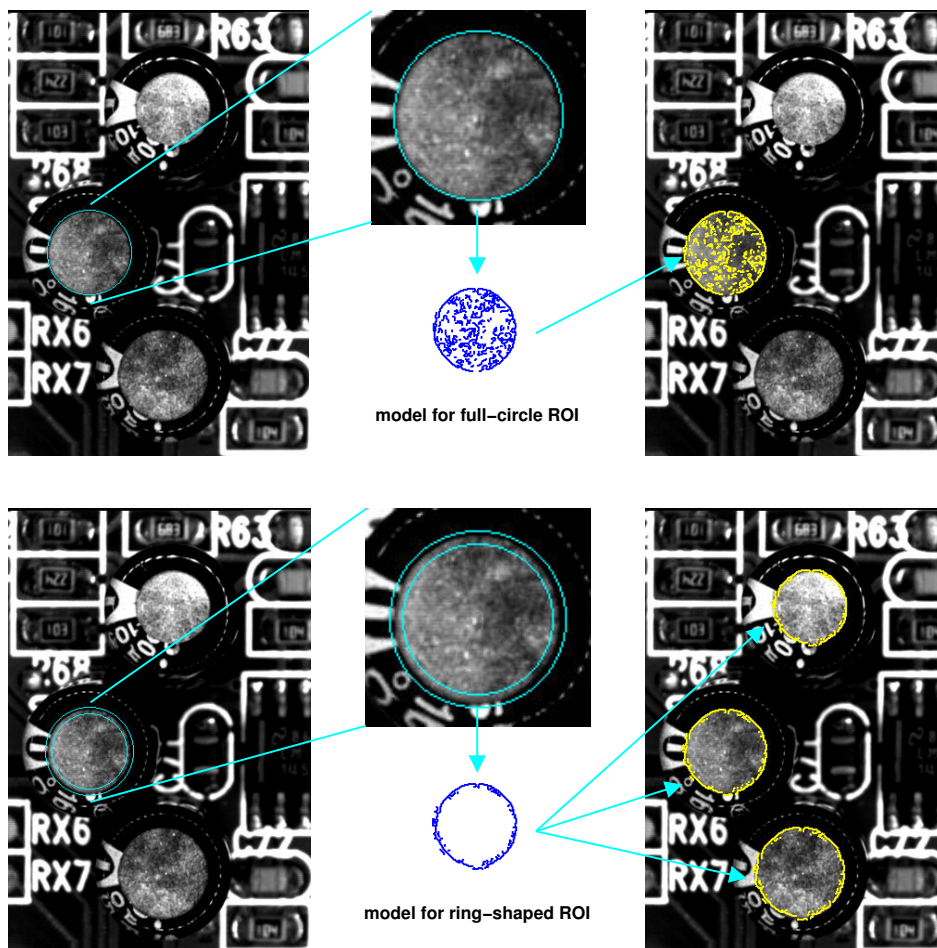


Figure 4: Masking the part of a region containing clutter.

contains many clutter points, which are caused by reflections on the metallic surface. Thus, the other two capacitors are not found. The solution to this problem is to use a ring-shaped ROI, which can be created by the following lines of code:

```
draw_circle (WindowHandle, ROI1Row, ROI1Column, ROI1Radius)
gen_circle (ROI1, ROI1Row, ROI1Column, ROI1Radius)
gen_circle (ROI2, ROI1Row, ROI1Column, ROI1Radius-8)
difference (ROI1, ROI2, ROI)
```

Note that the ROI should not be too “thin”, otherwise it vanishes at higher pyramid levels! As a rule of thumb, an ROI should be  $2^{NumLevels-1}$  pixels wide; in the example, the width of 8 pixels therefore allows to use 4 pyramid levels.



For this task even better results can be obtained by **using a synthetic model image**. This is described in [section 2.3](#) on page 18.



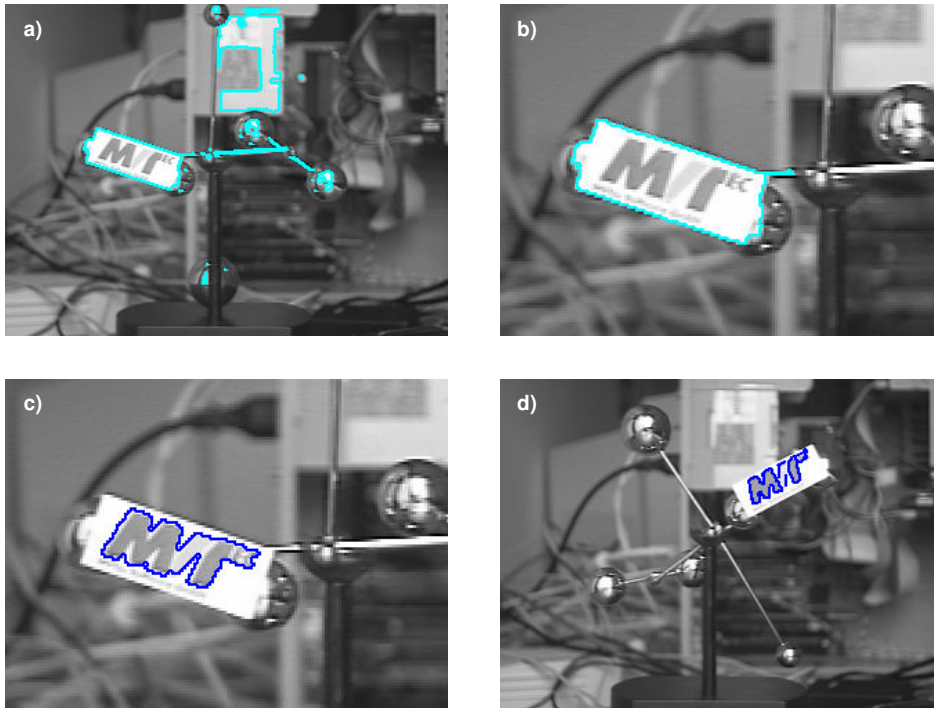


Figure 5: Using image processing to create an ROI: a) extract bright regions; b) select the card; c) the logo forms the ROI; d) result of the matching.

### 2.1.3 Using Image Processing to Create and Modify Regions

In the previous sections, regions were created explicitly by specifying their shape parameters. Especially for complex ROIs this method can be inconvenient and time-consuming. In the following, we therefore show you how to extract and modify regions using image processing operators.

#### Example 1: Determining the ROI Using Blob Analysis

To follow the example actively, start the HDevelop program `hdevelop\create_roi_via_vision.dev`, which locates the MVtec logo on a pendulum (see [figure 5](#)); we start after the initialization of the application (press Run once). The main idea is to “zoom in” on the desired region in multiple steps: First, find the bright region corresponding to the card, then extract the dark characters on it.

##### Step 1: Extract the bright regions

```
threshold (ModelImage, BrightRegions, 200, 255)
connection (BrightRegions, ConnectedRegions)
fill_up (ConnectedRegions, FilledRegions)
```

First, all bright regions are extracted using a simple thresholding operation (`threshold`); the operator

`connection` forms connected components. The extracted regions are then filled up via `fill_up`; thus, the region corresponding to the card also encompasses the dark characters (see [figure 5a](#)).

### Step 2: Select the region of the card

```
select_shape (FilledRegions, Card, 'area', 'and', 1800, 1900)
```

The region corresponding to the card can be selected from the list of regions with the operator `select_shape`. In HDevelop, you can determine suitable features and values using the dialog `Visualization > Region Info`; just click into a region, and the dialog immediately displays its feature values. [Figure 5b](#) shows the result of the operator.

### Step 3: Use the card as an ROI for the next steps

```
reduce_domain (ModelImage, Card, ImageCard)
```

Now, we can restrict the next image processing steps to the region of the card using the operator `reduce_domain`. This iterative focusing has an important advantage: In the restricted region of the card, the logo characters are much easier to extract than in the full image.

### Step 4: Extract the logo

```
threshold (ImageCard, DarkRegions, 0, 230)
connection (DarkRegions, ConnectedRegions)
select_shape (ConnectedRegions, Characters, 'area', 'and', 150, 450)
union1 (Characters, CharacterRegion)
```

The logo characters are extracted similarly to the card itself; as a last step, the separate character regions are combined using the operator `union1`.

### Step 5: Enlarge the region using morphology

```
dilation_circle (CharacterRegion, ROI, 1.5)
reduce_domain (ModelImage, ROI, ImageROI)
create_shape_model (ImageROI, 0, 0, rad(360), 0, 'none', 'use_polarity',
                    30, 10, ModelID)
```

Finally, the region corresponding to the logo is enlarged slightly using the operator `dilation_circle`. [Figure 5c](#) shows the resulting ROI, which is then used to create the shape model.

## Example 2: Further Processing the Result of `inspect_shape_model`

You can also combine the interactive ROI specification with image processing. A useful method in the presence of clutter in the model image is to create a first model region interactively and then process this region to obtain an improved ROI. [Figure 6](#) shows an example; the task is to locate the arrows. To follow the example actively, start the HDevelop program `hdevelop\process_shape_model.dev`; we start after the initialization of the application (press Run once).

### Step 1: Select the arrow

```
gen_rectangle1 (ROI, 361, 131, 406, 171)
```

First, an initial ROI is created around the arrow, without trying to exclude clutter (see [figure 6a](#)).

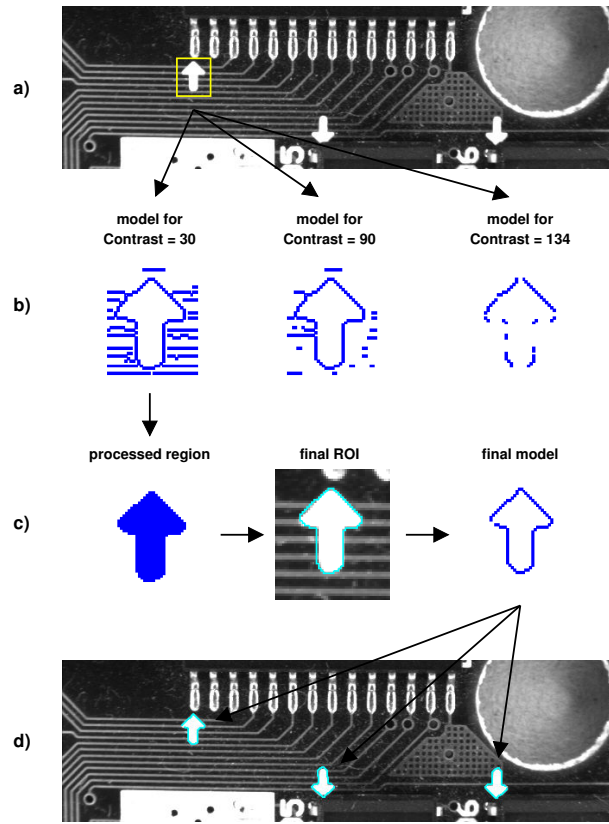


Figure 6: Processing the result of `inspect_shape_model`: a) interactive ROI; b) models for different values of `Contrast`; c) processed model region and corresponding ROI and model; d) result of the search.

### Step 2: Create a first model region

```
reduce_domain (ModelImage, ROI, ImageROI)
inspect_shape_model (ImageROI, ShapeModelImage, ShapeModelRegion, 1, 30)
```

Figure 6b shows the shape model regions that would be created for different values of the parameter `Contrast`. As you can see, you cannot remove the clutter without losing characteristic points of the arrow itself.

### Step 3: Process the model region

```
fill_up (ShapeModelRegion, FilledModelRegion)
opening_circle (FilledModelRegion, ROI, 3.5)
```

You can solve this problem by exploiting the fact that the operator `inspect_shape_model` returns the shape model region; thus, you can process it like any other region. The main idea to get rid of the clutter is to use morphological operator `opening_circle`, which eliminates small regions. Before this, the

operator `fill_up` must be called to fill the inner part of the arrow, because only the boundary points are part of the (original) model region. [Figure 6c](#) shows the resulting region.

#### Step 4: Create the final model

```
reduce_domain (ModelImage, ROI, ImageROI)
create_shape_model (ImageROI, 3, 0, rad(360), 0, 'none', 'use_polarity',
                   30, 10, ModelID)
```

The processed region is then used to create the model; [figure 6c](#) shows the corresponding ROI and the final model region. Now, all arrows are located successfully.

### 2.1.4 How the ROI Influences the Search

Note that the ROI used when creating the model also influences the results of the subsequent matching: By default, the center point of the ROI acts as the so-called *point of reference* of the model for the estimated position, rotation, and scale. After creating a model, you can change its point of reference with the operator `set_shape_model_origin`. Note that this operator expects not the absolute position of the new reference point as parameters, but its *distance* to the default reference point. Please note that by modifying the point of reference, the accuracy of the estimated position may decrease (see [section 3.4](#) on page 26). You can query the reference point using the operator `get_shape_model_origin`

The point of reference also influences the search itself: An object is only found if the point of reference lies within the image, or more exactly, within the *domain* of the image (see also [section 3.1.1](#) on page 20). Please note that this test is always performed for the original point of reference, i.e., the center point of the ROI, even if you modified the reference point using `set_shape_model_origin`.

## 2.2 Which Information is Stored in the Model?

As the name *shape-based pattern matching* suggests, objects are represented and recognized by their *shape*. There exist multiple ways to determine or describe the shape of an object. Here, the shape is extracted by selecting all those points whose *contrast* exceeds a certain threshold; typically, the points correspond to the contours of the object (see, e.g., [figure 1](#) on page 4). [Section 2.2.1](#) takes a closer look at the corresponding parameters.

To speed up the matching process, a so-called *image pyramid* is created, consisting of the original, full-sized image and a set of downsampled images. The model is then created and searched on the different pyramid levels (see [section 2.2.2](#) on page 14 for details).

If the object is allowed to appear rotated or scaled, the corresponding information is used already when creating the model. This also speeds up the matching process, at the cost of higher memory requirements for the created model. [Section 2.2.3](#) on page 16 and [section 2.2.4](#) on page 17 describe the corresponding parameters. If the high memory requirements become a problem, you can modify this behavior via the parameter `Optimization` (see [page 16](#)).

In the following, all parameters belong to the operator `create_shape_model` if not stated otherwise.

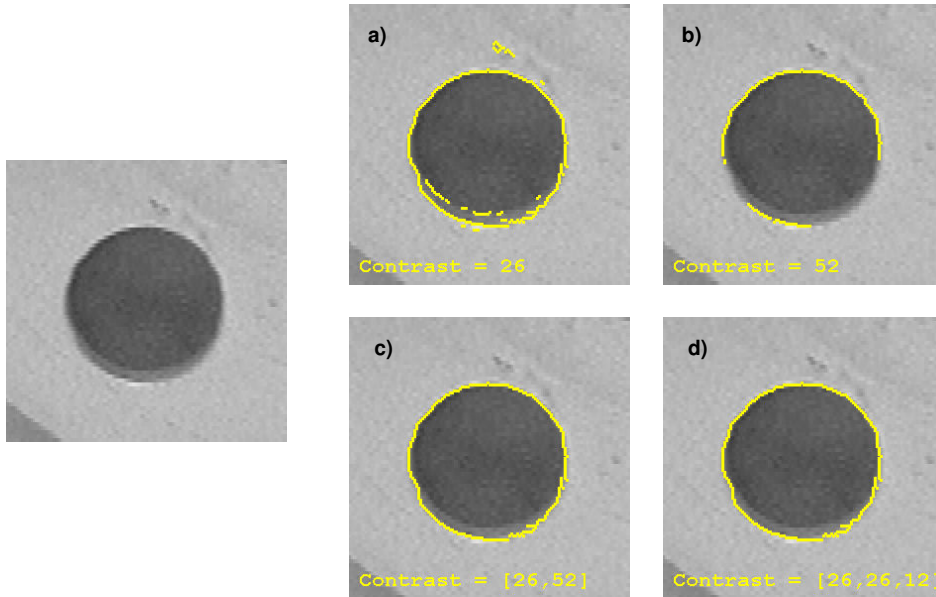


Figure 7: Selecting significant pixels via **Contrast**: a) complete object but with clutter; b) no clutter but incomplete object; c) hysteresis threshold; d) minimum contour size.

### 2.2.1 Which Pixels are Part of the Model?

For the model those pixels are selected whose *contrast*, i.e., gray value difference to neighboring pixels, exceeds a threshold specified by the parameter **Contrast** when calling `create_shape_model`. In order to obtain a suitable model the contrast should be chosen in such a way that the *significant* pixels of the object are included, i.e., those pixels that characterize it and allow to discriminate it clearly from other objects or from the background. Obviously, the model should not contain clutter, i.e., pixels that do not belong to the object.

In some cases it is impossible to find a single value for **Contrast** that removes the clutter but not also parts of the object. [Figure 7](#) shows an example; the task is to create a model for the outer rim of a drill-hole: If the complete rim is selected, the model also contains clutter ([figure 7a](#)); if the clutter is removed, parts of the rim are missing ([figure 7b](#)).

To solve such problems, the parameter **Contrast** provides two additional methods: hysteresis thresholding and selection of contour parts based on their size. Both methods are used by specifying a *tuple* of values for **Contrast** instead of a single value.

*Hysteresis thresholding* (see also the operator `hysteresis_threshold`) uses two thresholds, a lower and an upper threshold. For the model, first pixels that have a contrast higher than the upper threshold are selected; then, pixels that have a contrast higher than the lower threshold and that are connected to a high-contrast pixel, either directly or via another pixel with contrast above the lower threshold, are added. This method enables you to select contour parts whose contrast varies from pixel to pixel. Returning to the example of the drill-hole: As you can see in [figure 7c](#), with a hysteresis threshold you can create a model for the complete rim without clutter. The following line of code shows how to specify the two

thresholds in a tuple:

```
inspect_shape_model (ImageROI, ModelImages, ModelRegions, 1, [26,52])
```

The second method to remove clutter is to specify a minimum size, i.e., number of pixels, for the contour components. [Figure 7d](#) shows the result for the example task. The minimum size must be specified in the third element of the tuple; if you don't want to use a hysteresis threshold, set the first two elements to the same value:

```
inspect_shape_model (ImageROI, ModelImages, ModelRegions, 1, [26,26,12])
```

Alternative methods to remove clutter are to modify the ROI as described in [section 2.1](#) on page 6 or create a synthetic model (see [section 2.3](#) on page 18).

### 2.2.2 How Subsampling is Used to Speed Up the Search

To speed up the matching process, a so-called *image pyramid* is created, both for the model image and for the search images. The pyramid consists of the original, full-sized image and a set of downsampled images. For example, if the the original image (first pyramid level) is of the size 600x400, the second level image is of the size 300x200, the third level 150x100, and so on. The object is then searched first on the highest pyramid level, i.e., in the smallest image. The results of this fast search are then used to limit the search in the next pyramid image, whose results are used on the next lower level until the lowest level is reached. Using this iterative method, the search is both fast and accurate. [Figure 8](#) depicts 4 levels of an example image pyramid together with the corresponding model regions.

You can specify how many pyramid levels are used via the parameter [NumLevels](#). We recommend to choose the highest pyramid level at which the model contains at least 10-15 pixels and in which the shape of the model still resembles the shape of the object. You can inspect the model image pyramid using the operator [inspect\\_shape\\_model](#), e.g., as shown in the HDevelop program `hdevelop\first_example_shape_matching.dev`:

```
inspect_shape_model (ImageROI, ShapeModelImages, ShapeModelRegions, 8, 30)
area_center (ShapeModelRegions, AreaModelRegions, RowModelRegions,
             ColumnModelRegions)
HeightPyramid := |ShapeModelRegions|
for i := 1 to HeightPyramid by 1
    if (AreaModelRegions[i-1] >= 15)
        NumLevels := i
    endif
endfor
create_shape_model (ImageROI, NumLevels, 0, rad(360), 0, 'none',
                   'use_polarity', 30, 10, ModelID)
```

After the call to the operator, the model regions on the selected pyramid levels are displayed in HDevelop's Graphics Window; you can have a closer look at them using the online zooming (menu entry `Visualization ▸ Online Zooming`). The code lines following the operator call loop through the pyramid and determine the highest level on which the model contains at least 15 points. This value is then used in the call to the operator [create\\_shape\\_model](#).



A much easier method is to **let HALCON select a suitable value itself** by specifying the value 0 for

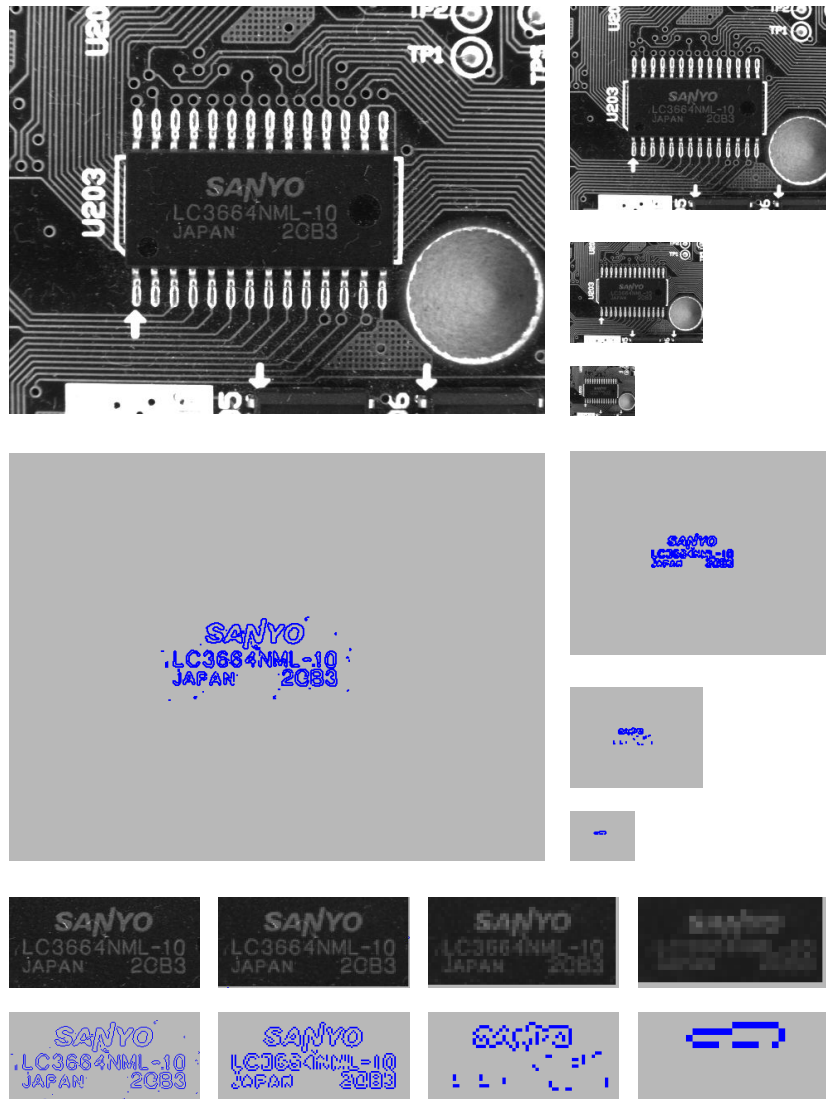


Figure 8: The image and the model region at four pyramid levels (original size and zoomed to equal size).

**NumLevels.** You can then query the used value via the operator `get_shape_model_params`.

The operator `inspect_shape_model` returns the pyramid images in form of an image tuple (array); the individual images can be accessed like the model regions with the operator `select_obj`. Please note that **object tuples start with the index 1, whereas control parameter tuples start with the index 0!**

You can enforce a further reduction of model points via the parameter `Optimization`. This may be useful to speed up the matching in the case of particularly large models. Please note that regardless of your selection all points passing the contrast criterion are displayed, i.e., you cannot check which points





are part of the model.

With an optional second value, you can specify whether the model is pregenerated completely for the allowed range of rotation and scale (see the following sections) or not. By default, the model is pregenerated, because this speeds up the search process. A reason to switch off the pregeneration is if the created model requires too much memory. In such a case, pass 'no\_pregeneration' as the second value of [Optimization](#). Alternatively, you can set this parameter via the operator [set\\_system](#).

### 2.2.3 Allowing a Range of Orientation

If the object's rotation may vary in the search images you can specify the allowed range in the parameter [AngleExtent](#) and the starting angle of this range in the parameter [AngleStart](#) (unit: rad). Note that the range of rotation is defined relative to the model image, i.e., a starting angle of 0 corresponds to the orientation the object has in the model image. Therefore, to allow rotations up to  $\pm 5^\circ$ , e.g., you should set the starting angle to  $-\text{rad}(5)$  and the angle extent to  $\text{rad}(10)$ .

We recommend to limit the allowed range of rotation as much as possible in order to speed up the search process and to minimize the required memory. Note that you can further limit the allowed range when calling the operator [find\\_shape\\_model](#) (see [section 3.1.2](#) on page 21). If you want to reuse a model for different tasks requiring a different range of angles and if memory is not an issue, you can therefore use a large range when creating the model and a smaller range for the search.

If the object is (almost) symmetric you should limit the allowed range. Otherwise, the search process will find multiple, almost equally good matches on the same object at different angles; which match (at which angle) is returned as the best can therefore “jump” from image to image. The suitable range of rotation depends on the symmetry: For a cross-shaped or square object the allowed extent must be less than  $90^\circ$ , for a rectangular object less than  $180^\circ$ , and for a circular object  $0^\circ$ .

To speed up the matching process, the model is precomputed for different angles within the allowed range (if you didn't switch off this behavior as described in the previous section), at steps specified with the parameter [AngleStep](#). If you select the value 0, HALCON automatically chooses an optimal step size  $\phi_{opt}$  to obtain the highest possible accuracy by determining the smallest rotation that is still discernible in the image. The underlying algorithm is explained in [figure 9](#): The rotated version of the cross-shaped object is clearly discernible from the original if the point that lies farthest from the center of the object is moved by at least 2 pixels. Therefore, the corresponding angle  $\phi_{opt}$  is calculated as follows:

$$d^2 = l^2 + l^2 - 2 \cdot l \cdot l \cdot \cos \phi \quad \Rightarrow \quad \phi_{opt} = \arccos \left( 1 - \frac{d^2}{2 \cdot l^2} \right) = \arccos \left( 1 - \frac{2}{l^2} \right)$$

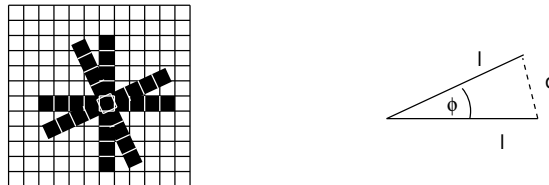


Figure 9: Determining the minimum angle step size from the extent of the model.



with  $l$  being the maximum distance between the center and the object boundary and  $d = 2$  pixels.

The automatically determined angle step size  $\phi_{opt}$  is suitable for most applications; therefore, **we recommend to select the value 0**. You can query the used value after the creation via the operator `get_shape_model_params`. By selecting a higher value you can speed up the search process, however, at the cost of a decreased accuracy of the estimated orientation. Note that for very high values the matching may fail altogether!

The value chosen for `AngleStep` should not deviate too much from the optimal value ( $\frac{1}{3}\phi_{opt} \leq \phi \leq 3\phi_{opt}$ ). Note that choosing a very small step size does not result in an increased angle accuracy!

## 2.2.4 Allowing a Range of Scale

Similarly to the range of orientation, you can specify an allowed range of scale with the parameters `ScaleMin`, `ScaleMax`, and `ScaleStep` of the operator `create_scaled_shape_model`.

Again, we recommend to limit the allowed range of scale as much as possible in order to speed up the search process and to minimize the required memory. Note that you can further limit the allowed range when calling the operator `find_scaled_shape_model` (see section 3.1.2 on page 21). If too much memory is required, you can switch off the pregeneration of the model as described on page 16.

Note that if you are searching for the object on a large range of scales you should **create the model based on a large scale** because HALCON cannot “guess” model points when precomputing model instances at scales larger than the original one. On the other hand, `NumLevels` should be chosen such that the highest level contains enough model points also for the smallest scale.

If you select the value 0 for the parameter `ScaleStep`, HALCON automatically chooses a suitable step size to obtain the highest possible accuracy by determining the smallest scale change that is still discernible in the image. Similarly to the angle step size (see figure 9 on page 16), a scaled object is clearly discernible from the original if the point that lies farthest from the center of the object is moved by at least 2 pixels. Therefore, the corresponding scale change  $\Delta s_{opt}$  is calculated as follows:

$$\Delta s = \frac{d}{l} \quad \Rightarrow \quad \Delta s_{opt} = \frac{2}{l}$$

with  $l$  being the maximum distance between the center and the object boundary and  $d = 2$  pixels.

The automatically determined scale step size is suitable for most applications; therefore, **we recommend to select the value 0**. You can query the used value after the creation via the operator `get_shape_model_params`. By selecting a higher value you can speed up the search process, however, at the cost of a decreased accuracy of the estimated scale. Note that for very high values the matching may fail altogether!

The value chosen for `ScaleStep` should not deviate too much from the optimal value ( $\frac{1}{3}\Delta s_{opt} \leq \Delta s \leq 3\Delta s_{opt}$ ). Note that choosing a very small step size does not result in an increased scale accuracy!

## 2.2.5 Which Pixels are Compared with the Model?

For efficiency reasons the model contains information that influences the search process: With the parameter `MinContrast` you can specify which contrast a point in a search image must at least have in

order to be compared with the model. The main use of this parameter is to exclude noise, i.e., gray value fluctuations, from the matching process. You can determine the noise by examining the gray values with the HDevelop dialog **Visualization** ▸ **Pixel Info**; then, set the minimum contrast to a value larger than the noise.

The parameter **Metric** lets you specify whether the *polarity*, i.e., the direction of the contrast must be observed. If you choose the value 'use\_polarity' the polarity is observed, i.e., the points in the search image must show the same direction of the contrast as the corresponding points in the model. If, for example, the model is a bright object on a dark background, the object is found in the search images only if it is also brighter than the background.

You can choose to ignore the polarity globally by selecting the value 'ignore\_global\_polarity'. In this mode, an object is recognized also if the direction of its contrast reverses, e.g., if your object can appear both as a dark shape on a light background and vice versa. This flexibility, however, is obtained at the cost of a slightly lower recognition speed.

If you select the value 'ignore\_local\_polarity', the object is found even if the contrast changes locally. This mode can be useful, e.g., if the object consists of a part with a medium gray value, within which either darker or brighter sub-objects lie. Please note however, that the recognition speed may decrease dramatically in this mode, especially if you allowed a large range of rotation (see [section 2.2.3](#) on page 16).

## 2.3 Synthetic Model Images

Depending on the application it may be difficult to create a suitable model because there is no “good” model image containing a perfect, easy to extract instance of the object. An example of such a case was already shown in [section 2.1.2](#) on page 7: The task of locating the capacitors seems to be simple at first, as they are prominent bright circles on a dark background. But because of the clutter inside and outside the circle even the model resulting from the ring-shaped ROI is faulty: Besides containing clutter points also parts of the circle are missing.

In such cases, it may be better to use a *synthetic model image*. How to create such an image to locate the capacitors is explained below. To follow the example actively, start the HDevelop program `hdevelop\synthetic_circle.dev`; we start after the initialization of the application (press Run once).

### Step 1: Create an XLD contour

```
RadiusCircle := 43
SizeSynthImage := 2*RadiusCircle + 10
gen_ellipse_contour_xld (Circle, SizeSynthImage / 2, SizeSynthImage / 2, 0,
                        RadiusCircle, RadiusCircle, 0, 6.28318,
                        'positive', 1.5)
```

First, we create a circular region using the operator `gen_ellipse_contour_xld` (see [figure 10a](#)). You can determine a suitable radius by inspecting the image with the HDevelop dialog **Visualization** ▸ **Online Zooming**. Note that the synthetic image should be larger than the region because pixels around the region are used when creating the image pyramid.

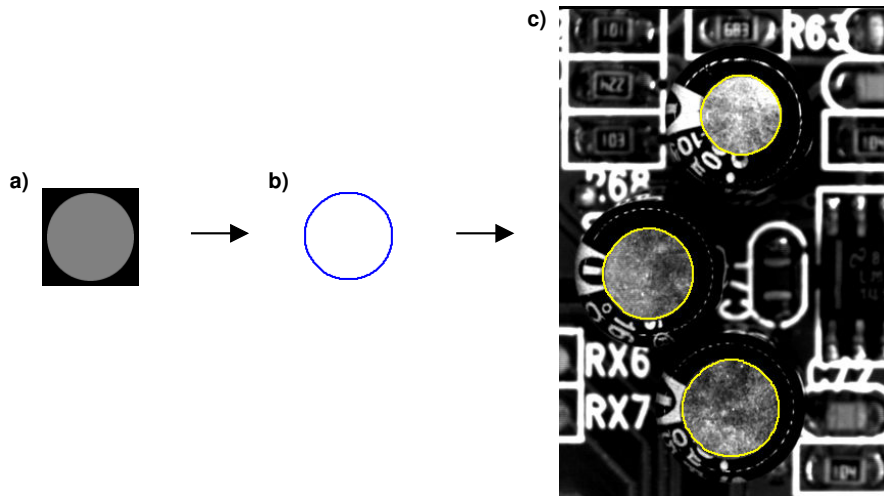


Figure 10: Locating the capacitors using a synthetic model: a) paint region into synthetic image; b) corresponding model; c) result of the search.

### Step 2: Create an image and insert the XLD contour

```
gen_image_const (EmptyImage, 'byte', SizeSynthImage, SizeSynthImage)
paint_xld (Circle, EmptyImage, SyntheticModelImage, 128)
```

Then, we create an empty image using the operator `gen_image_const` and insert the XLD contour with the operator `paint_xld`. In [figure 10a](#) the resulting image is depicted.

### Step 3: Create the model

```
create_scaled_shape_model (SyntheticModelImage, 0, 0, 0, 0.01, 0.8, 1.2, 0,
                           'none', 'use_polarity', 30, 10, ModelID)
```

Now, the model is created from the synthetic image. [Figure 10d](#) shows the corresponding model region, [figure 10e](#) the search results.

Note how the image itself, i.e., its domain, acts as the ROI in this example.

## 3 Optimizing the Search Process

The actual matching is performed by the operators `find_shape_model`, `find_scaled_shape_model`, `find_shape_models`, or `find_scaled_shape_models`. In the following, we show how to select suitable parameters for these operators to adapt and optimize it for your matching task.

### 3.1 Restricting the Search Space

An important concept in the context of finding objects is that of the so-called *search space*. Quite literally, this term specifies where to search for the object. However, this space encompasses not only the 2 dimensions of the image, but also other parameters like the possible range of scales and orientations or the question of how much of the object must be visible. The more you can restrict the search space, the faster the search will be.

#### 3.1.1 Searching in a Region of Interest

The obvious way to restrict the search space is to apply the operator `find_shape_model` to a region of interest only instead of the whole image as shown in [figure 11](#). This can be realized in a few lines of code:

##### Step 1: Create a region of interest

```
Row1 := 141
Column1 := 163
Row2 := 360
Column2 := 477
gen_rectangle1 (SearchROI, Row1, Column1, Row2, Column2)
```

First, you create a region, e.g., with the operator `gen_rectangle1` (see [section 2.1.1](#) on [page 7](#) for more ways to create regions).

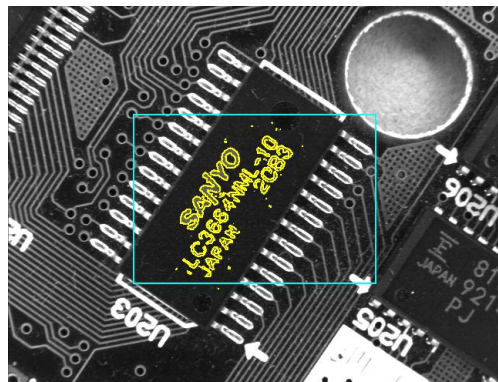


Figure 11: Searching in a region of interest.

**Step 2: Restrict the search to the region of interest**

```

for i := 1 to 20 by 1
  grab_image (SearchImage, FGHandle)
  reduce_domain (SearchImage, SearchROI, SearchImageROI)
  find_shape_model (SearchImageROI, ModelID, 0, rad(360), 0.8, 1, 0.5,
                    'interpolation', 0, 0.9, RowCheck, ColumnCheck,
                    AngleCheck, Score)
endfor

```

The region of interest is then applied to each search image using the operator `reduce_domain`. In this example, the searching speed is almost doubled using this method.

Note that by restricting the search to a region of interest you actually restrict the position of the *point of reference* of the model, i.e., the center of gravity of the model ROI (see [section 2.1.4](#) on page 12). This means that the size of the search ROI corresponds to the extent of the allowed movement; for example, if your object can move  $\pm 10$  pixels vertically and  $\pm 15$  pixels horizontally you can restrict the search to an ROI of the size  $20 \times 30$ . In order to assure a correct boundary treatment on higher pyramid levels, we recommend to enlarge the ROI by  $2^{NumLevels-1}$  pixels; to continue the example, if you specified `NumLevels = 4`, you can restrict the search to an ROI of the size  $36 \times 46$ .

Please note that even if you modify the point of reference using `set_shape_model_origin`, the original one, i.e., the center point of the model ROI, is used during the search. Thus, you must always specify the search ROI relative to the original reference point.

**3.1.2 Restricting the Range of Orientation and Scale**

When creating the model with the operator `create_shape_model` (or `create_scaled_shape_model`), you already specified the allowed range of orientation and scale (see [section 2.2.3](#) on page 16 and [section 2.2.4](#) on page 17). When calling the operator `find_shape_model` (or `find_scaled_shape_model`) you can further limit these ranges with the parameters `AngleStart`, `AngleExtent`, `ScaleMin`, and `ScaleMax`. This is useful if you can restrict these ranges by other information, which can, e.g., be obtained by suitable image processing operations.

Another reason for using a larger range when creating the model may be that you want to reuse the model for other matching tasks.

**3.1.3 Visibility**

With the parameter `MinScore` you can specify how much of the object — more precisely: of the model — must be visible. A typical use of this mechanism is to allow a certain degree of occlusion as demonstrated in [figure 12](#): The security ring is found if `MinScore` is set to 0.7.

Let's take a closer look at the term “visibility”: When comparing a part of a search image with the model, the matching process calculates the so-called *score*, which is a measure of how many model points could be matched to points in the search image (ranging from 0 to 1). A model point may be “invisible” and thus not matched because of multiple reasons:

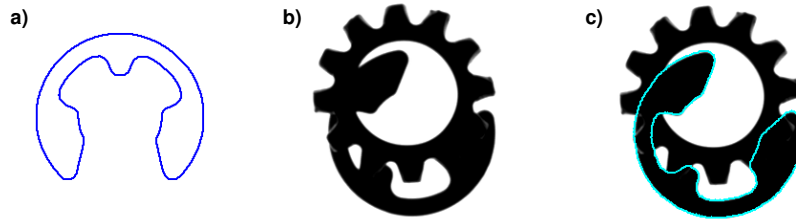


Figure 12: Searching for partly occluded objects: a) model of the security ring; b) search result for `MinScore` = 0.8; c) search result for `MinScore` = 0.7.

- Parts of the object's contour are occluded, e.g., as in [figure 12](#).

Please note that **an object must not be clipped at the image border**; this case is not treated as an occlusion! More precisely, the smallest rectangle surrounding the model must not be clipped.



- Parts of the contour have a contrast lower than specified in the parameter `MinContrast` when creating the model (see [section 2.2.5](#) on page 17).
- The polarity of the contrast changes globally or locally (see [section 2.2.5](#) on page 17).
- If the object is deformed, parts of the contour may be visible but appear at an incorrect position and therefore do not fit the model anymore. Note that this effect also occurs if camera observes the scene under an oblique angle; [section 5.1](#) on page 45 shows how to handle this case.

Besides these obvious reasons, which have their root in the search image, there are some not so obvious reasons caused by the matching process itself:

- As described in [section 2.2.3](#) on page 16, HALCON precomputes the model for intermediate angles within the allowed range of orientation. During the search, a candidate match is then compared to all precomputed model instances. If you select a value for the parameter `AngleStep` that is significantly larger than the automatically selected minimum value, the effect depicted in [figure 13](#) can occur: If the object lies between two precomputed angles, points lying far from the center are not matched to a model point, and therefore the score decreases.

Of course, the same line of reasoning applies to the parameter `ScaleStep` (see [section 2.2.4](#) on page 17).

- Another stumbling block lies in the use of an image pyramid which was introduced in [section 2.2.2](#) on page 14: When comparing a candidate match with the model, the specified minimum score must be reached on each pyramid level. However, on different levels the score may vary, with only the score on the lowest level being returned in the parameter `Score`; this sometimes leads to the apparently paradox situation that `MinScore` must be set significantly lower than the resulting `Score`.



**Recommendation:** The higher `MinScore`, the faster the search!

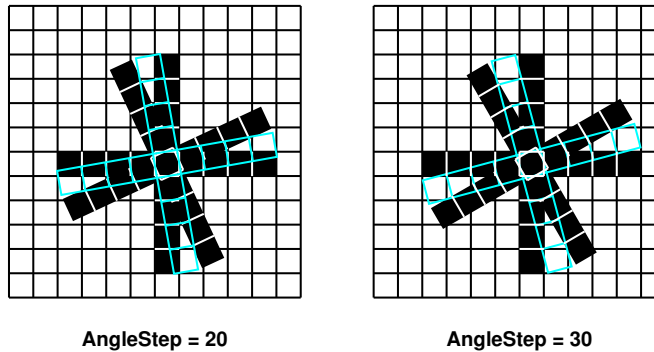


Figure 13: The effect of a large `AngleStep` on the matching.

### 3.1.4 Thoroughness vs. Speed

With the parameter `Greediness` you can influence the search algorithm itself and thereby trade thoroughness against speed. If you select the value 0, the search is thorough, i.e., if the object is present (and within the allowed search space and reaching the minimum score), it will be found. In this mode, however, even very unlikely match candidates are also examined thoroughly, thereby slowing down the matching process considerably.

The main idea behind the “greedy” search algorithm is to break off the comparison of a candidate with the model when it seems unlikely that the minimum score will be reached. In other words, the goal is not to waste time on hopeless candidates. This greediness, however, can have unwelcome consequences: In some cases a perfectly visible object is not found because the comparison “starts out on a wrong foot” and is therefore classified as a hopeless candidate and broken off.

You can adjust the `Greediness` of the search, i.e., how early the comparison is broken off, by selecting values between 0 (no break off: thorough but slow) and 1 (earliest break off: fast but unsafe). Note that the parameters `Greediness` and `MinScore` interact, i.e., you may have to specify a lower minimum score in order to use a greedier search. Generally, you can reach a higher speed with a high greediness and a sufficiently lowered minimum score.

## 3.2 Searching for Multiple Instances of the Object

All you have to do to search for more than one instance of the object is to set the parameter `NumMatches` accordingly. The operator `find_shape_model` (or `find_scaled_shape_model`) then returns the matching results as tuples in the parameters `Row`, `Column`, `Angle`, `Scale`, and `Score`. If you select the value 0, all matches are returned.

Note that a search for multiple objects is only slightly slower than a search for a single object.

A second parameter, `MaxOverlap`, lets you specify how much two matches may overlap (as a fraction). In figure 14b, e.g., the two security rings overlap by a factor of approximately 0.2. In order to speed up the matching as far as possible, however, the overlap is calculated not for the models themselves but for their smallest surrounding rectangle. This must be kept in mind when specifying the maximum overlap; in most cases, therefore a larger value is needed (e.g., compare figure 14b and figure 14d).

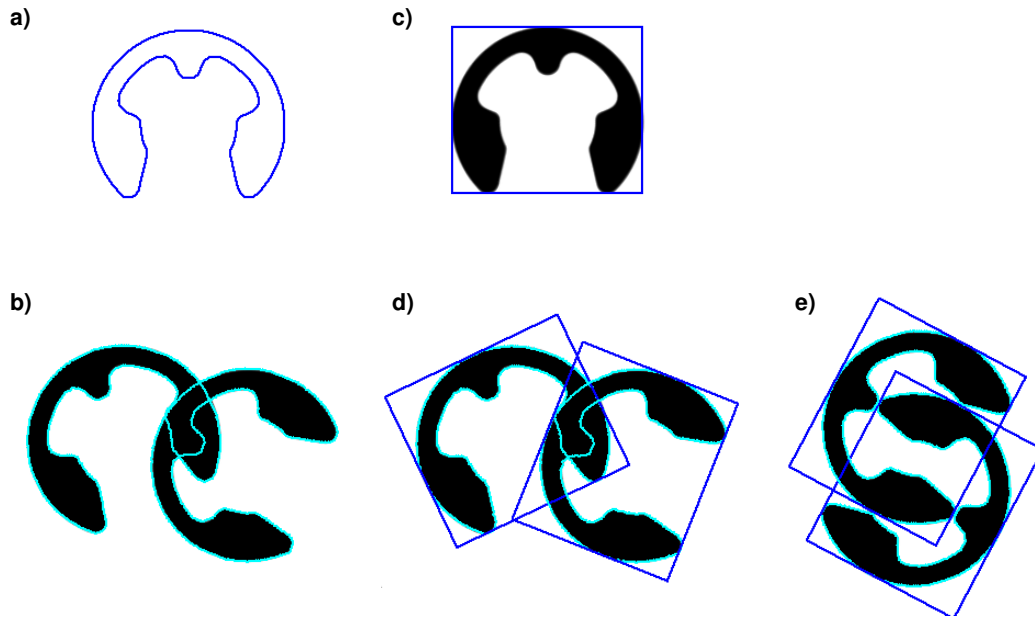


Figure 14: A closer look at overlapping matches: a) model of the security ring; b) model overlap; c) smallest rectangle surrounding the model; d) rectangle overlap; e) pathological case.

Figure 14e shows a “pathological” case: Even though the rings themselves do not overlap, their surrounding rectangles do to a large degree. Unfortunately, this effect cannot be prevented.

### 3.3 Searching for Multiple Models Simultaneously

If you are searching for instances of multiple models in a single image, you can of course call the operator `find_shape_model` (or `find_scaled_shape_model`) multiple times. A much faster alternative is to use the operators `find_shape_models` or `find_scaled_shape_models` instead. These operators expect similar parameters, with the following differences:

- With the parameter `ModelIDs` you can specify a *tuple* of model IDs instead of a single one. As when searching for multiple instances (see [section 3.2](#) on page 23), the matching result parameters `Row` etc. return tuples of values.
- The output parameter `Model` shows to which model each found instance belongs. Note that the parameter does not return the model IDs themselves but the index of the model ID in the tuple `ModelIDs` (starting with 0).
- The search is always performed in a single image. However, you can restrict the search to a certain region for each model individually by passing an image tuple (see below for an example).
- You can either use the same search parameters for each model by specifying single values for `AngleStart` etc., or pass a tuple containing individual values for each model.



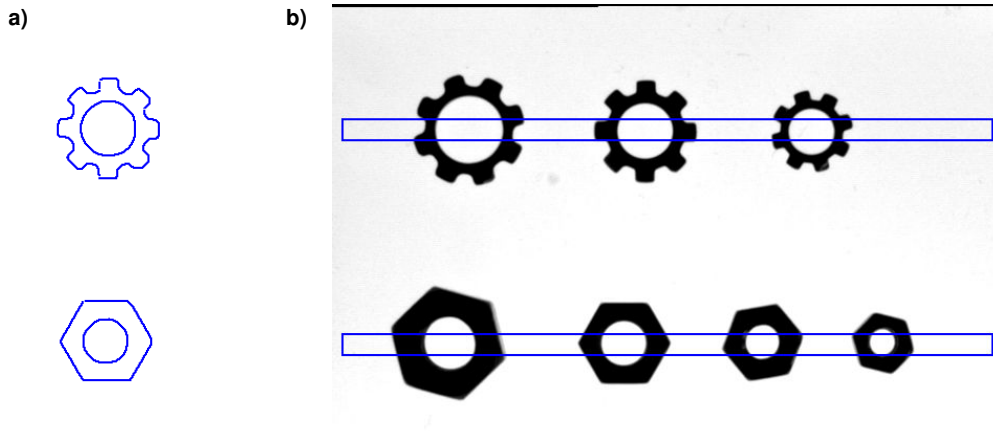


Figure 15: Searching for multiple models : a) models of ring and nut; b) search ROIs for the two models.

- You can also search for multiple instances of multiple models. If you search for a certain number of objects independent of their type (model ID), specify this (single) value in the parameter [NumMatches](#). By passing a tuple of values, you can specify for each model individually how many instances are to be found. In this tuple, you can mix concrete values with the value 0; the tuple `[3, 0]`, e.g., specifies to return the best 3 instances of the first model and all instances of the second model.

Similarly, if you specify a single value for [MaxOverlap](#), the operators check whether a found instance is overlapped by any of the other instances independent of their type. By specifying a tuple of values, each instance is only checked against all other instances of the same type.

The example HDevelop program `hdevelop\multiple_models.dev` uses the operator [find\\_scaled\\_shape\\_models](#) to search simultaneously for the rings and nuts depicted in [figure 15](#).

### Step 1: Create the models

```
create_scaled_shape_model (ImageROIring, 0, -rad(22.5), rad(45), 0, 0.8,
                          1.2, 0, 'none', 'use_polarity', 60, 10,
                          ModelIDRing)
create_scaled_shape_model (ImageROINut, 0, -rad(30), rad(60), 0, 0.6, 1.4,
                          0, 'none', 'use_polarity', 60, 10, ModelIDNut)
ModelIDs := [ModelIDRing, ModelIDNut]
```

First, two models are created, one for the rings and one for the nuts. The two model IDs are then concatenated into a tuple using the operator [assign](#).

## Step 2: Specify individual search ROIs

```
gen_rectangle1 (SearchROIring, 110, 10, 130, Width - 10)
gen_rectangle1 (SearchROInut, 315, 10, 335, Width - 10)
SearchROIs := [SearchROIring, SearchROInut]
add_channels (SearchROIs, SearchImage, SearchImageReduced)
```

In the example, the rings and nuts appear in non-overlapping parts of the search image; therefore, it is possible to restrict the search space for each model individually. As explained in [section 3.1.1](#) on page 20, a search ROI corresponds to the extent of the allowed movement; thus, narrow horizontal ROIs can be used in the example (see [figure 15b](#)).

The two ROIs are concatenated into a region array (tuple) using the operator `concat_obj` and then “added” to the search image using the operator `add_channels`. The result of this operator is an array of two images, both having the same image matrix; the domain of the first image is restricted to the first ROI, the domain of the second image to the second ROI.

## Step 3: Find all instances of the two models

```
find_scaled_shape_models (SearchImageReduced, ModelIDs, [-rad(22.5),
    -rad(30)], [rad(45), rad(60)], [0.8, 0.6], [1.2,
    1.4], 0.7, 0, 0, 'interpolation', 0, 0.9,
    RowCheck, ColumnCheck, AngleCheck, ScaleCheck,
    Score, ModelIndex)
```

Now, the operator `find_scaled_shape_models` is applied to the created image array. Because the two models allow different ranges of rotation and scaling, tuples are specified for the corresponding parameters. In contrast, the other parameters are valid for both models. [Section 4.3.3](#) on page 35 shows how to access the matching results.

## 3.4 A Closer Look at the Accuracy

During the matching process, candidate matches are compared with instances of the model at different positions, angles, and scales; for each instance, the resulting matching score is calculated. If you set the parameter `SubPixel` to 'none', the result parameters `Row`, `Column`, `Angle`, and `Scale` contain the corresponding values of the best match. In this case, the accuracy of the position is therefore 1 pixel, while the accuracy of the orientation and scale is equal to the values selected for the parameters `AngleStep` and `ScaleStep`, respectively, when creating the model (see [section 2.2.3](#) on page 16 and [section 2.2.4](#) on page 17).

If you set the parameter `SubPixel` to 'interpolation', HALCON examines the matching scores at the neighboring positions, angles, and scales around the best match and determines the maximum by interpolation. Using this method, the position is therefore estimated with subpixel accuracy ( $\approx \frac{1}{20}$  pixel in typical applications). The accuracy of the estimated orientation and scale depends on the size of the object, like the optimal values for the parameters `AngleStep` and `ScaleStep` (see [section 2.2.3](#) on page 16 and [section 2.2.4](#) on page 17): The larger the size, the more accurately the orientation and scale can be determined. For example, if the maximum distance between the center and the boundary is 100 pixel, the orientation is typically determined with an accuracy of  $\approx \frac{1}{10}^\circ$ .



**Recommendation:** Because the interpolation is very fast, you can set `SubPixel` to 'interpolation'

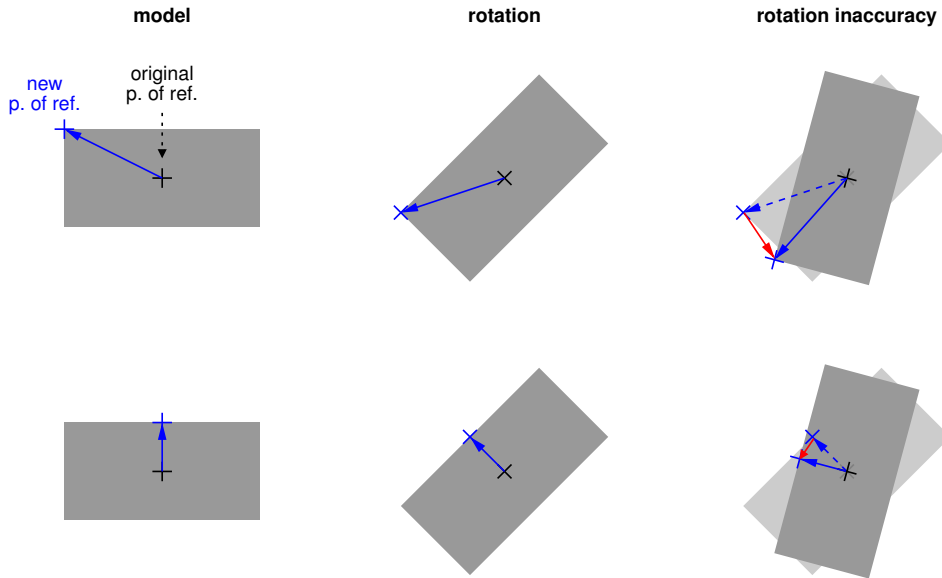


Figure 16: Effect of inaccuracy of the estimated orientation on a moved point of reference.

in most applications.

When you choose the values `'least_squares'`, `'least_squares_high'`, or `'least_squares_very_high'`, a least-squares adjustment is used instead of an interpolation, resulting in a higher accuracy. However, this method requires additional computation time.

Please note that the **accuracy of the estimated position may decrease if you modify the point of reference** using `set_shape_model_origin`! This effect is visualized in [figure 16](#): As you can see in the right-most column, an inaccuracy in the estimated orientation “moves” the modified point of reference, while the original point of reference is not affected. The resulting positional error depends on multiple factors, e.g., the offset of the reference point and the orientation of the found object. The main point to keep in mind is that the error increases linearly with the *distance* of the modified point of reference from the original one (compare the two rows in [figure 16](#)).

An inaccuracy in the estimated scale also results in an error in the estimated position, which again increases linearly with the distance between the modified and the original reference point.

For maximum accuracy in case the reference point is moved, the position should be determined using the least-squares adjustment. Note that the accuracy of the estimated orientation and scale is not influenced by modifying the reference point.



### 3.5 How to Optimize the Matching Speed



In the following, we show how to optimize the matching process in two steps. Please note that in order to optimize the matching it is very important to have a **set of representative test images from your application** in which the object appears in all allowed variations regarding its position, orientation, occlusion, and illumination.

#### Step 1: Assure that all objects are found

Before tuning the parameters for speed, we recommend to find settings such that the matching succeeds in all test images, i.e., that all object instances are found. If this is not the case when using the default values, check whether one of the following situations applies:

- ? Is the object clipped at the image border?**  
Unfortunately, this failure cannot be prevented, i.e., you must assure that the object is not clipped (see [section 3.1.3](#) on page 21).
- ? Is the search algorithm “too greedy”?**  
As described in [section 3.1.4](#) on page 23, in some cases a perfectly visible object is not found if the [Greediness](#) is too high. Select the value 0 to force a thorough search.
- ? Is the object partly occluded?**  
If the object should be recognized in this state nevertheless, reduce the parameter [MinScore](#).
- ? Does the matching fail on the highest pyramid level?**  
As described in [section 3.1.3](#) on page 21, in some cases the minimum score is not reached on the highest pyramid level even though the score on the lowest level is much higher. Test this by reducing [NumLevels](#) in the call to [find\\_shape\\_model](#). Alternatively, reduce the [MinScore](#).
- ? Does the object have a low contrast?**  
If the object should be recognized in this state nevertheless, reduce the parameter [MinContrast](#) (operator [create\\_shape\\_model!](#)).
- ? Is the polarity of the contrast inverted globally or locally?**  
If the object should be recognized in this state nevertheless, use the appropriate value for the parameter [Metric](#) when creating the model (see [section 2.2.5](#) on page 17). If only a small part of the object is affected, it may be better to reduce the [MinScore](#) instead.
- ? Does the object overlap another instance of the object?**  
If the object should be recognized in this state nevertheless, increase the parameter [MaxOverlap](#) (see [section 3.2](#) on page 23).
- ? Are multiple matches found on the same object?**  
If the object is almost symmetric, restrict the allowed range of rotation as described in [section 2.2.3](#) on page 16 or decrease the parameter [MaxOverlap](#) (see [section 3.2](#) on page 23).

#### Step 2: Tune the parameters regarding speed

The speed of the matching process depends both on the model and on the search parameters. To make matters more difficult, the search parameters depend on the chosen model parameters. We recommend the following procedure:

- Increase the [MinScore](#) as far as possible, i.e., as long as the matching succeeds.
- Now, increase the [Greediness](#) until the matching fails. Try reducing the [MinScore](#); if this does not help restore the previous values.
- If possible, use a larger value for [NumLevels](#) when creating the model.
- Restrict the allowed range of rotation and scale as far as possible as described in [section 2.2.3](#) on page 16 and [section 2.2.4](#) on page 17. Alternatively, adjust the corresponding parameters when calling [find\\_shape\\_model](#) or [find\\_scaled\\_shape\\_model](#).
- Restrict the search to a region of interest as described in [section 3.1.1](#) on page 20.

The following methods are more “risky”, i.e., the matching may fail if you choose unsuitable parameter values.

- Increase the [MinContrast](#) as long as the matching succeeds.
- If you are searching for a particularly large object, it sometimes helps to select a higher point reduction with the parameter [Optimization](#) (see [section 2.2.2](#) on page 14).
- Increase the [AngleStep](#) (and the [ScaleStep](#)) as long as the matching succeeds.

## 4 Using the Results of Matching

As results, the operators `find_shape_model`, `find_scaled_shape_model` etc. return

- the position of the match in the parameters `Row` and `Column`,
- its orientation in the parameter `Angle`,
- the scaling factor in the parameter `Scale`, and
- the matching score in the parameter `Score`.

The matching score, which is a measure of the similarity between the model and the matched object, can be used “as it is”, since it is an absolute value.

In contrast, the results regarding the position, orientation, and scale are worth a closer look as they are determined relative to the created model. Before this, we introduce HALCON’s powerful operators for the so-called *affine transformations*, which, when used together with the shape-based matching, enable you to easily realize applications like image rectification or the alignment of ROIs with a few lines of code.

### 4.1 Introducing Affine Transformations

“Affine transformation” is a technical term in mathematics describing a certain group of transformations. [Figure 17](#) shows the types that occur in the context of the shape-based matching: An object can be *translated* (moved) along the two axes, *rotated*, and *scaled*. In [figure 17d](#), all three transformations were applied in a sequence.

Note that for the rotation and the scaling there exists a special point, called *fixed point* or *point of reference*. The transformation is performed around this point. In [figure 17b](#), e.g., the IC is rotated around its center, in [figure 17e](#) around its upper right corner. The point is called fixed point because it remains unchanged by the transformation.

The transformation can be thought of as a mathematical instruction that defines how to calculate the coordinates of object points after the transformation. Fortunately, you need not worry about the mathematical part; HALCON provides a set of operators that let you specify and apply transformations in a simple way.

### 4.2 Creating and Applying Affine Transformations With HALCON

HALCON allows to transform not only regions, but also images and XLD contours by providing the operators `affine_trans_region`, `affine_trans_image`, and `affine_trans_contour_xld`. The transformation in [figure 17d](#) corresponds to the line

```
affine_trans_region (IC, TransformedIC, ScalingRotationTranslation,
                  'false')
```

The parameter `ScalingRotationTranslation` is a so-called *homogeneous transformation matrix* that describes the desired transformation. You can create this matrix by adding simple transformations step by step. First, an identity matrix is created:

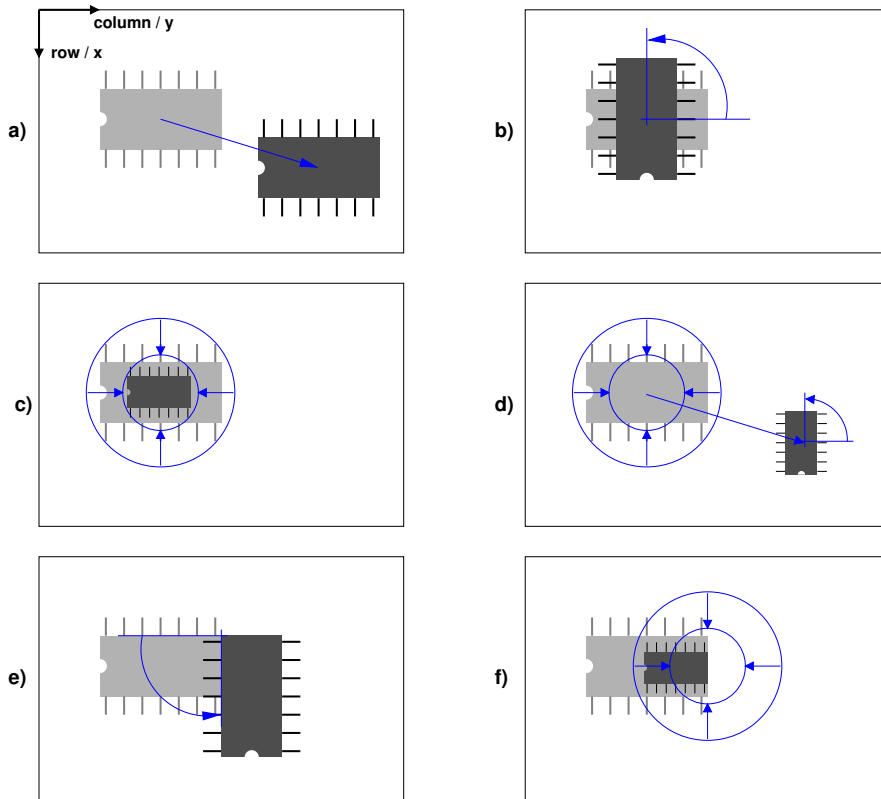


Figure 17: Typical affine transformations: a) translation along two axes; b) rotation around the IC center; c) scaling around the IC center; d) combining a, b, and c; e) rotation around the upper right corner; f) scaling around the right IC center.

```
hom_mat2d_identity (EmptyTransformation)
```

Then, the scaling around the center of the IC is added:

```
hom_mat2d_scale (EmptyTransformation, 0.5, 0.5, RowCenterIC,
                ColumnCenterIC, Scaling)
```

Similarly, the rotation and the translation are added:

```
hom_mat2d_rotate (Scaling, rad(90), RowCenterIC, ColumnCenterIC,
                 ScalingRotation)
hom_mat2d_translate (ScalingRotation, 100, 200, ScalingRotationTranslation)
```

Please note that in these operators the coordinate axes are labeled with x and y instead of Row and Column! [Figure 17a](#) clarifies the relation.

Transformation matrices can also be constructed by a sort of “reverse engineering”. In other words, if the result of the transformation is known for some points of the object, you can determine the corresponding

transformation matrix. If, e.g., the position of the IC center and its orientation after the transformation is known, you can get the corresponding matrix via the operator `vector_angle_to_rigid`.

```
vector_angle_to_rigid (RowCenterIC, ColumnCenterIC, 0,
                      TransformedRowCenterIC, TransformedColumnCenterIC,
                      rad(90), RotationTranslation)
```

and then use this matrix to compute the transformed region:

```
affine_trans_region (IC, TransformedIC, RotationTranslation, 'false')
```

## 4.3 Using the Estimated Position and Orientation

Even if the task is just to check whether an object is present in the image, you will typically use the returned position and orientation: to display the found instance. This basic use is described in [section 4.3.1](#). More advanced applications are to align ROIs for other inspection tasks, e.g., measuring ([section 4.3.4](#) on page 36), or to transform the search image so that the object is positioned as in the model image ([section 4.3.5](#) on page 39). [Section 4.4](#) on page 43 shows how to locate grasping points on nuts, which could then be passed on to a robot.



There are two things to keep in mind about the position and orientation returned in the parameters [Row](#), [Column](#), and [Angle](#): Most important, contrary to expectation **the estimated position is not exactly the position of the point of reference** but only close to it. Instead, it is optimized for creating the transformation matrix with which the applications described above can be realized.

Secondly, in the model image the object is taken as not rotated, i.e., its angle is 0, even if it seems to be rotated, e.g., as in [figure 18b](#).

### 4.3.1 Displaying the Matches

Especially during the development of a matching application it is useful to display the matching results overlaid on the search image. This can be realized in a few steps (see, e.g., the HDevelop program `hdevelop\first_example_shape_matching.dev`):

#### Step 1: Access the XLD contour containing the model

```
create_shape_model (ImageROI, NumLevels, 0, rad(360), 0, 'none',
                  'use_polarity', 30, 10, ModelID)
get_shape_model_contours (ShapeModel, ModelID, 1)
```

Below, we want to display the model at the extracted position and orientation. As shown in [section 1](#) on page 4, the corresponding region can be accessed via the operator `inspect_shape_model`. This is useful to display the model in the model image. However, for the search images we recommend to use the XLD version of the model, because XLD contours can be transformed more precisely and quickly. You can access the XLD model by calling the operator `get_shape_model_contours` after creating the model. Note that the XLD model is located in the origin of the image, not on the position of the model in the model image.



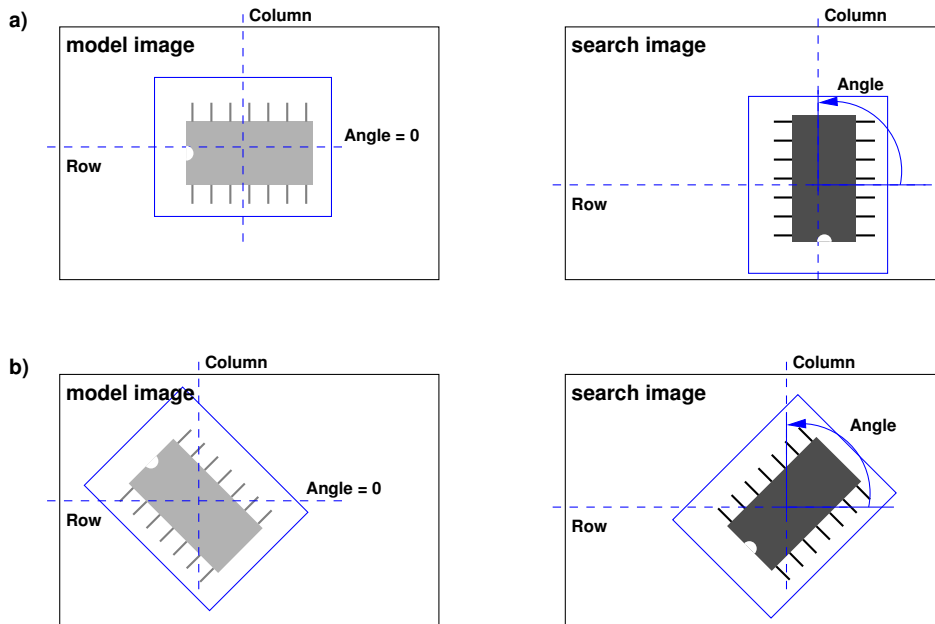


Figure 18: The position and orientation of a match: a) The center of the ROI acts as the default point of reference; b) In the model image, the orientation is always 0.

### Step 2: Determine the affine transformation

```
find_shape_model (SearchImage, ModelID, 0, rad(360), 0.7, 1, 0.5,
                  'interpolation', 0, 0.9, RowCheck, ColumnCheck,
                  AngleCheck, Score)

if (|Score| = 1)
    vector_angle_to_rigid (0, 0, 0, RowCheck, ColumnCheck, AngleCheck,
                          MovementOfObject)
```

After the call of the operator `find_shape_model`, the results are checked; if the matching failed, empty tuples are returned in the parameters `Score` etc. For a successful match, the corresponding affine transformation can be constructed with the operator `vector_angle_to_rigid` from the position and orientation of the match (see [section 4.2](#) on page 30). In the first 2 parameters, you pass the “relative” position of the reference point, i.e., its distance to the default reference point (the center of gravity of the ROI, see [section 2.1.4](#) on page 12). By default, this relative position is (0,0); [section 4.3.4](#) on page 36 shows the values in the case of a modified reference point.

### Step 3: Transform the XLD

```
affine_trans_contour_xld (ShapeModel, ModelAtNewPosition,
                          MovementOfObject)

dev_display (ModelAtNewPosition)
```

Now, you can apply the transformation to the XLD version of the model using the operator `affine_trans_contour_xld` and display it; [figure 2](#) on page 5 shows the result.

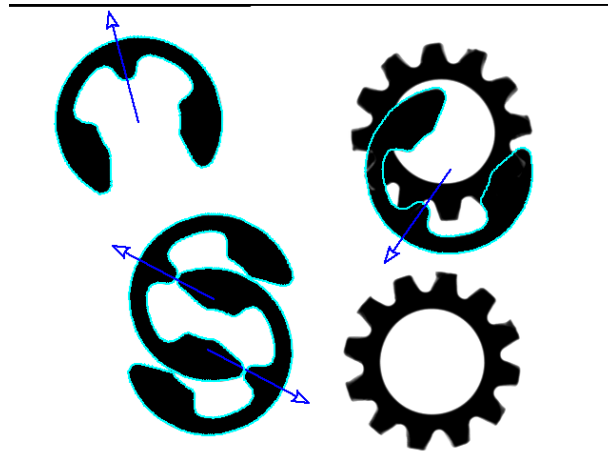


Figure 19: Displaying multiple matches; the used model is depicted in [figure 12a](#) on page 22 .

### 4.3.2 Dealing with Multiple Matches

If multiple instances of the object are searched and found, the parameters [Row](#), [Column](#), [Angle](#), and [Score](#) contain *tuples*. The HDevelop program `hdevelop\multiple_objects.dev` shows how to access these results in a loop:

#### Step 1: Determine the affine transformation

```
find_shape_model (SearchImage, ModelID, 0, rad(360), 0.75, 0, 0.55,
                  'interpolation', 0, 0.8, RowCheck, ColumnCheck,
                  AngleCheck, Score)
for j := 0 to |Score| - 1 by 1
    vector_angle_to_rigid (0, 0, 0, RowCheck[j], ColumnCheck[j],
                          AngleCheck[j], MovementOfObject)
    affine_trans_contour_xld (ShapeModel, ModelAtNewPosition,
```

The transformation corresponding to the movement of the match is determined as in the previous section; the only difference is that the position of the match is extracted from the tuple via the loop variable.

#### Step 2: Use the transformation

```
affine_trans_pixel (MovementOfObject, -120, 0, RowArrowHead,
                  ColumnArrowHead)
disp_arrow (WindowHandle, RowCheck[j], ColumnCheck[j],
            RowArrowHead, ColumnArrowHead, 2)
```

In this example, the transformation is also used to display an arrow that visualizes the orientation (see [figure 19](#)). For this, the position of the arrow head is transformed using `affine_trans_pixel` with the same transformation matrix as the XLD model.



Note that **you must use the operator** `affine_trans_pixel` and not `affine_trans_point_2d`, because the latter uses a different image coordinate system than `affine_trans_pixel`, `affine_trans_contour_xld`, `affine_trans_region`, and `affine_trans_image`.

### 4.3.3 Dealing with Multiple Models

When searching for multiple models simultaneously as described in [section 3.3](#) on page 24, it is useful to store the information about the models, i.e., the XLD models, in tuples. The following example code stems from the already partly described HDevelop program `hdevelop\multiple_models.dev`, which uses the operator `find_scaled_shape_models` to search simultaneously for the rings and nuts depicted in [figure 15](#) on page 25.

#### Step 1: Access the XLD models

```
create_scaled_shape_model (ImageROIring, 0, -rad(22.5), rad(45), 0, 0.8,
                          1.2, 0, 'none', 'use_polarity', 60, 10,
                          ModelIDRing)
get_shape_model_contours (ShapeModelRing, ModelIDRing, 1)
create_scaled_shape_model (ImageROInut, 0, -rad(30), rad(60), 0, 0.6, 1.4,
                          0, 'none', 'use_polarity', 60, 10, ModelIDNut)
get_shape_model_contours (ShapeModelNut, ModelIDNut, 1)
```

As in the previous sections, the XLD contours corresponding to the two models are accessed with the operator `get_shape_model_contours`.

#### Step 2: Save the information about the models in tuples

```
NumContoursRing := |ShapeModelRing|
NumContoursNut := |ShapeModelNut|
ShapeModels := [ShapeModelRing, ShapeModelNut]
StartContoursInTuple := [1, NumContoursRing+1]
NumContoursInTuple := [NumContoursRing, NumContoursNut]
```

To facilitate the access to the shape models later, the XLD contours are saved in tuples in analogy to the model IDs (see [section 3.3](#) on page 24). However, when concatenating XLD contours with the operator `concat_obj`, one must keep in mind that XLD objects are already tuples as they may consist of multiple contours! To access the contours belonging to a certain model, you therefore need the number of contours of a model and the starting index in the concatenated tuple. The former is determined using the operator `count_obj`; the contours of the ring start with the index 1, the contours of the nut with the index 1 plus the number of contours of the ring.

### Step 3: Access the found instances

```

find_scaled_shape_models (SearchImageReduced, ModelIDs, [-rad(22.5),
    -rad(30)], [rad(45), rad(60)], [0.8, 0.6], [1.2,
    1.4], 0.7, 0, 0, 'interpolation', 0, 0.9,
    RowCheck, ColumnCheck, AngleCheck, ScaleCheck,
    Score, ModelIndex)
for i := 0 to |Score| - 1 by 1
    Model := ModelIndex[i]
    vector_angle_to_rigid (0, 0, 0, RowCheck[i], ColumnCheck[i],
        AngleCheck[i], MovementOfObject)
    hom_mat2d_scale (MovementOfObject, ScaleCheck[i], ScaleCheck[i],
        RowCheck[i], ColumnCheck[i], MoveAndScalingOfObject)
    copy_obj (ShapeModels, ShapeModel, StartContoursInTuple[Model],
        NumContoursInTuple[Model])
    affine_trans_contour_xld (ShapeModel, ModelAtNewPosition,
        MoveAndScalingOfObject)
    dev_display (ModelAtNewPosition)
endfor

```

As described in [section 4.3.2](#) on page 34, in case of multiple matches the output parameters `Row` etc. contain tuples of values, which are typically accessed in a loop, using the loop variable as the index into the tuples. When searching for multiple models, a second index is involved: The output parameter `Model` indicates to which model a match belongs by storing the index of the corresponding model ID in the tuple of IDs specified in the parameter `ModelIDs`. This may sound confusing, but can be realized in an elegant way in the code: For each found instance, the model ID index is used to select the corresponding information from the tuples created above.

As already noted, the XLD representing the model can consist of multiple contours; therefore, you cannot access them directly using the operator `select_obj`. Instead, the contours belonging to the model are selected via the operator `copy_obj`, specifying the start index of the model in the concatenated tuple and the number of contours as parameters. Note that `copy_obj` does not copy the contours, but only the corresponding HALCON objects, which can be thought of as references to the contours.

#### 4.3.4 Aligning Other ROIs

The results of the matching can be used to *align* ROIs for other image processing steps. i.e., to position them relative to the image part acting as the model. This method is very useful, e.g., if the object to be inspected is allowed to move or if multiple instances of the object are to be inspected at once as in the example application described below.

In the example application `hdevelop\align_measurements.dev`, the task is to inspect razor blades by measuring the width and the distance of their “teeth”. [Figure 20a](#) shows the model ROI, [figure 20b](#) the corresponding model region.

The inspection task is realized with the following steps:

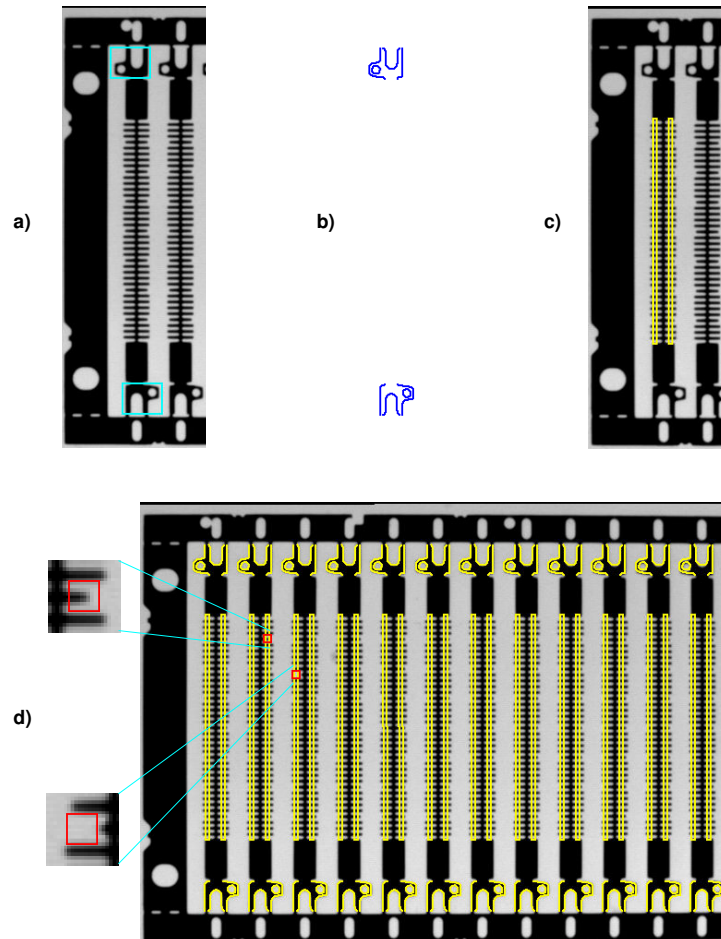


Figure 20: Aligning ROIs for inspecting parts of a razor: a) ROIs for the model; b) the model; c) measuring ROIs; d) inspection results with zoomed faults.

### Step 1: Position the measurement ROIs for the model blade

```

Rect1Row := 244
Rect1Col := 73
DistColRect1Rect2 := 17
Rect2Row := Rect1Row
Rect2Col := Rect1Col + DistColRect1Rect2
RectPhi := rad(90)
RectLength1 := 122
RectLength2 := 2

```

First, two rectangular measurement ROIs are placed over the teeth of the razor blade acting as the model as shown in [figure 20c](#). To be able to transform them later along with the XLD model, they are moved to lie on the XLD model, whose reference point is the origin of the image. Note that before moving the

regions the clipping must be switched off.

```
area_center (ModelROI, Area, CenterROIRow, CenterROIColumn)
get_system ('clip_region', OriginalClipRegion)
set_system ('clip_region', 'false')
move_region (MeasureROI1, MeasureROI1Ref, - CenterROIRow,
             - CenterROIColumn)
move_region (MeasureROI2, MeasureROI2Ref, - CenterROIRow,
             - CenterROIColumn)
set_system ('clip_region', OriginalClipRegion)
DistRect1CenterRow := Rect1Row - CenterROIRow
DistRect1CenterCol := Rect1Col - CenterROIColumn
DistRect2CenterRow := Rect2Row - CenterROIRow
DistRect2CenterCol := Rect2Col - CenterROIColumn
```

### Step 2: Find all razor blades

```
find_shape_model (SearchImage, ModelID, 0, 0, 0.8, 0, 0.5, 'interpolation',
                  0, 0.7, RowCheck, ColumnCheck, AngleCheck, Score)
```

Then, all instances of the model object are searched for in the image.

### Step 3: Determine the affine transformation

```
for i := 0 to |Score|-1 by 1
  vector_angle_to_rigid (0, 0, 0, RowCheck[i], ColumnCheck[i],
                        AngleCheck[i], MovementOfObject)
  affine_trans_contour_xld (ShapeModel, ModelAtNewPosition,
                           MovementOfObject)
```

For each razor blade, the transformation representing its position and orientation is calculated.

### Step 4: Create measurement objects at the corresponding positions

```
affine_trans_pixel (MovementOfObject, DistRect1CenterRow,
                   DistRect1CenterCol, Rect1RowCheck,
                   Rect1ColCheck)
affine_trans_pixel (MovementOfObject, DistRect2CenterRow,
                   DistRect2CenterCol, Rect2RowCheck,
                   Rect2ColCheck)
```



Now, the new positions of the measure ROIs are calculated using the operator `affine_trans_pixel` with the moved ROI coordinates. As remarked in [section 4.3.2](#) on page 34, **you must use** `affine_trans_pixel` and not `affine_trans_point_2d`. Then, the new measure objects are created.

```

RectPhiCheck := RectPhi + AngleCheck[i]
gen_measure_rectangle2 (Rect1RowCheck, Rect1ColCheck,
                        RectPhiCheck, RectLength1, RectLength2,
                        Width, Height, 'bilinear',
                        MeasureHandle1)
gen_measure_rectangle2 (Rect2RowCheck, Rect2ColCheck,
                        RectPhiCheck, RectLength1, RectLength2,
                        Width, Height, 'bilinear',
                        MeasureHandle2)

```

In the example application, the individual razor blades are only translated but not rotated relative to the model position. Instead of applying the full affine transformation to the measure ROIs and then creating new measure objects, one can therefore use the operator `translate_measure` to translate the measure objects themselves. The example program contains the corresponding code; you can switch between the two methods by modifying a variable at the top of the program.

#### Step 5: Measure the width and the distance of the “teeth”

```

measure_pairs (SearchImage, MeasureHandle1, 2, 25, 'negative',
              'all', RowEdge11, ColEdge11, Amp11, RowEdge21,
              ColEdge21, Amp21, Width1, Distance1)
measure_pairs (SearchImage, MeasureHandle2, 2, 25, 'negative',
              'all', RowEdge12, ColEdge12, Amp12, RowEdge22,
              ColEdge22, Amp22, Width2, Distance2)

```

Now, the actual measurements are performed using the operator `measure_pairs`.

#### Step 6: Inspect the measurements

```

NumberTeeth1 := |Width1|
if (NumberTeeth1 < 37)
  for j := 0 to NumberTeeth1 - 2 by 1
    if (Distance1[j] > 4.0)
      RowFault := round(0.5*(RowEdge11[j+1] + RowEdge21[j]))
      ColFault := round(0.5*(ColEdge11[j+1] + ColEdge21[j]))
      disp_rectangle2 (WindowHandle, RowFault, ColFault, 0,
                      4, 4)

```

Finally, the measurements are inspected. If a “tooth” is too short or missing completely, no edges are extracted at this point resulting in an incorrect number of extracted edge pairs. In this case, the faulty position can be determined by checking the distance of the teeth. Figure 20d shows the inspection results for the example.

Please note that the example program is not able to display the fault if it occurs at the first or the last tooth.

### 4.3.5 Rectifying the Search Results

In the previous section, the matching results were used to determine the so-called *forward transformation*, i.e., how objects are transformed from the model into the search image. Using this transformation, ROIs specified in the model image can be positioned correctly in the search image.

You can also determine the *inverse transformation* which transforms objects from the search image back into the model image. With this transformation, you can *rectify* the search image (or parts of it), i.e., transform it such that the matched object is positioned as it was in the model image. This method is useful if the following image processing step is not invariant against rotation, e.g., OCR or the variation model. Note that image rectification can also be useful *before* applying shape-based matching, e.g., if the camera observes the scene under an oblique angle; see [section 5.1](#) on page 45 for more information.

The inverse transformation can be determined and applied in a few steps, which are described below; in the corresponding example application of the HDevelop program `hdevelop\rectify_results.dev` the task is to extract the serial number on CD covers (see [figure 21](#)).

### Step 1: Calculate the inverse transformation

```
vector_angle_to_rigid (CenterROIRow, CenterROIColumn, 0, RowCheck,
                      ColumnCheck, AngleCheck, MovementOfObject)
hom_mat2d_invert (MovementOfObject, InverseMovementOfObject)
```

You can invert a transformation easily using the operator `hom_mat2d_invert`. Note that in contrast to the previous sections, the transformation is calculated based on the absolute coordinates of the reference point, because here we want to transform the results such that they appear as in the model image.

### Step 2: Rectify the search image

```
affine_trans_image (SearchImage, RectifiedSearchImage,
                  InverseMovementOfObject, 'constant', 'false')
```

Now, you can apply the inverse transformation to the search image using the operator `affine_trans_image`. [Figure 21d](#) shows the resulting rectified image of a different CD; undefined pixels are marked in grey.

### Step 3: Extract the numbers

```
reduce_domain (RectifiedSearchImage, NumberROI,
              RectifiedNumberROIImage)
threshold (RectifiedNumberROIImage, Numbers, 0, 128)
connection (Numbers, IndividualNumbers)
```

Now, the serial number is positioned correctly within the original ROI and can be extracted without problems. [Figure 21e](#) shows the result, which could then, e.g., be used as the input for OCR.

Unfortunately, the operator `affine_trans_image` transforms the full image even if you restrict its domain with the operator `reduce_domain`. In a time-critical application it may therefore be necessary to crop the search image before transforming it. The corresponding steps are visualized in [figure 22](#).

### Step 1: Crop the search image

```
affine_trans_region (NumberROI, NumberROIAtNewPosition,
                  MovementOfObject, 'false')
smallest_rectangle1 (NumberROIAtNewPosition, Row1, Column1, Row2,
                  Column2)
crop_rectangle1 (SearchImage, CroppedNumberROIImage, Row1, Column1,
                Row2, Column2)
```

First, the smallest axis-parallel rectangle surrounding the transformed number ROI is computed using





Figure 21: Rectifying the search results: a) ROIs for the model and for the number extraction; b) the model; c) number ROI at matched position; d) rectified search image (only relevant part shown); e) extracted numbers.

the operator `smallest_rectangle1`, and the search image is cropped to this part. Figure 22b shows the resulting image overlaid on a grey rectangle to facilitate the comparison with the subsequent images.

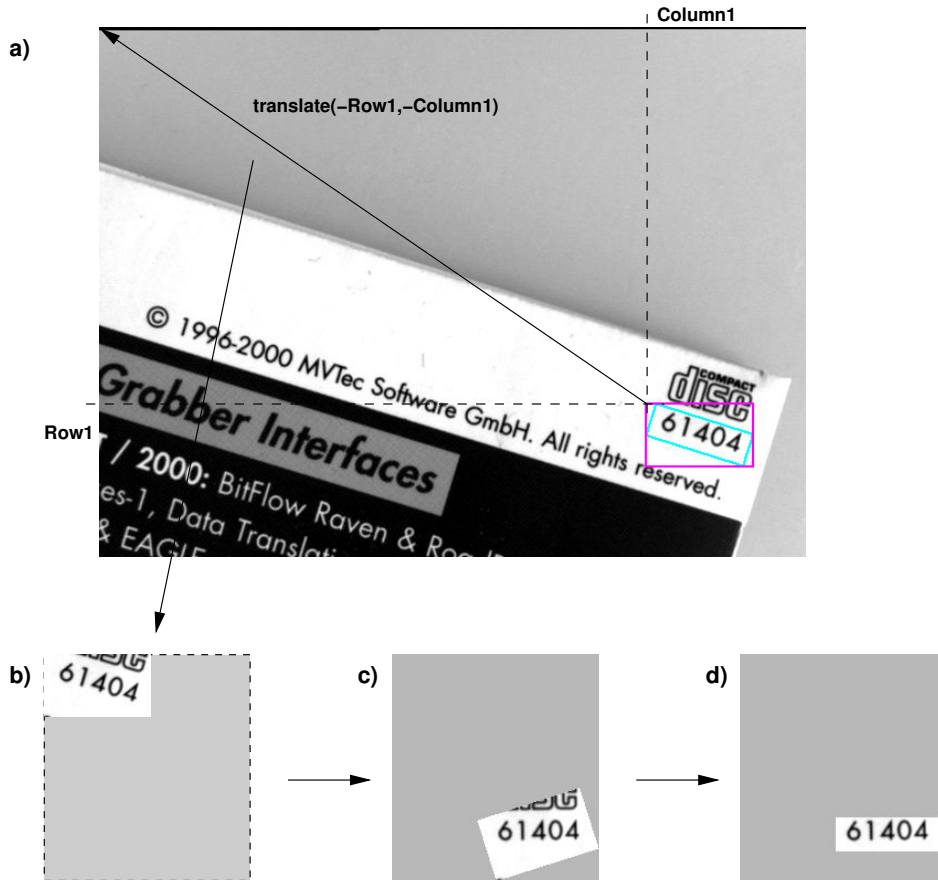


Figure 22: Rectifying only part of the search image: a) smallest image part containing the ROI; b) cropped search image; c) result of the rectification; d) rectified image reduced to the original number ROI.

### Step 2: Create an extended affine transformation

```
hom_mat2d_translate (MovementOfObject, - Row1, - Column1,
                    MoveAndCrop)
hom_mat2d_invert (MoveAndCrop, InverseMoveAndCrop)
```

In fact, the cropping can be interpreted as an additional affine transformation: a translation by the negated coordinates of the upper left corner of the cropping rectangle (see [figure 22a](#)). We therefore “add” this transformation to the transformation describing the movement of the object using the operator `hom_mat2d_translate`, and then invert this extended transformation with the operator `hom_mat2d_invert`.

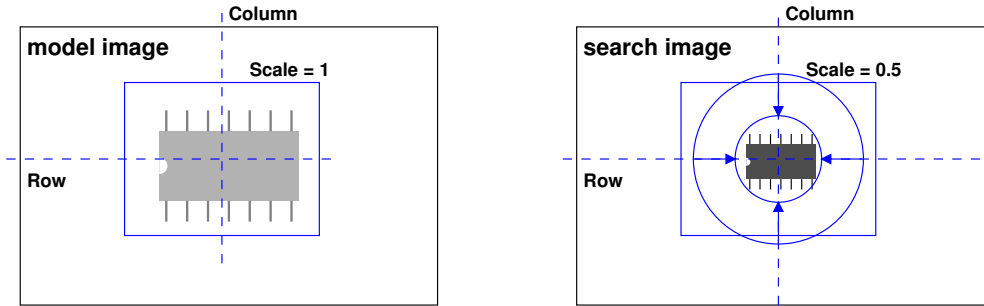


Figure 23: The center of the ROI acts as the point of reference for the scaling.

### Step 3: Transform the cropped image

```
affine_trans_image (CroppedNumberROIImage, RectifiedROIImage,
                   InverseMoveAndCrop, 'constant', 'true')
reduce_domain (RectifiedROIImage, NumberROI,
               RectifiedNumberROIImage)
```

Using the inverted extended transformation, the cropped image can easily be rectified with the operator `affine_trans_image` (figure 22c) and then be reduced to the original number ROI (figure 22d) in order to extract the numbers.

## 4.4 Using the Estimated Scale

Similarly to the rotation (compare section 4.3 on page 32), the scaling is performed around the center of the ROI – if you didn't use `set_shape_model_origin`, that is. This is depicted in figure 23a at the example of an ROI whose center does not coincide with the center of the IC.

The estimated scale, which is returned in the parameter `Scale`, can be used similarly to the position and orientation. However, there is no convenience operator like `vector_angle_to_rigid` that creates an affine transformation including the scale; therefore, the scaling must be added separately. How to achieve this is explained below; in the corresponding example HDevelop program `hdevelop\multiple_scales.dev`, the task is to find nuts of varying sizes and to determine suitable points for grasping them (see figure 24).

### Step 1: Specify grasping points

```
RowUpperPoint := 284
ColUpperPoint := 278
RowLowerPoint := 362
ColLowerPoint := 278
```

In the example program, the grasping points are specified directly in the model image; they are marked with arrows in figure 24c. To be able to transform them together with the XLD model, their coordinates must be moved so that they lie on the XLD model:

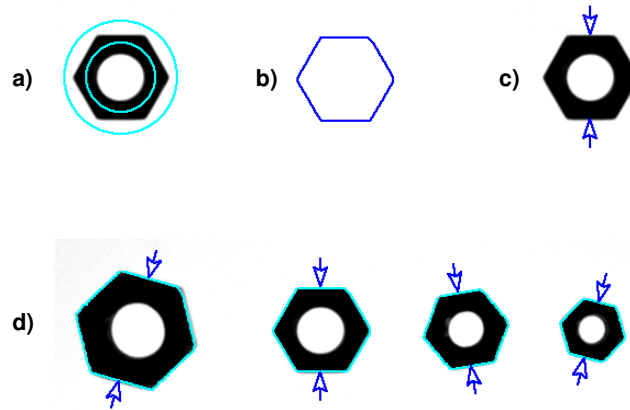


Figure 24: Determining grasping points on nuts of varying sizes: a) ring-shaped ROI; b) model; c) grasping points defined on the model nut; d) results of the matching.

```
area_center (ModelROI, Area, CenterROIRow, CenterROIColumn)
RowUpperPointRef := RowUpperPoint - CenterROIRow
ColUpperPointRef := ColUpperPoint - CenterROIColumn
RowLowerPointRef := RowLowerPoint - CenterROIRow
ColLowerPointRef := ColLowerPoint - CenterROIColumn
```

## Step 2: Determine the complete transformation


```
find_scaled_shape_model (SearchImage, ModelID, -rad(30), rad(60), 0.6, 1.4,
                        0.9, 0, 0, 'interpolation', 0, 0.8, RowCheck,
                        ColumnCheck, AngleCheck, ScaleCheck, Score)
for i := 0 to |Score| - 1 by 1
    vector_angle_to_rigid (0, 0, 0, RowCheck[i], ColumnCheck[i],
                          AngleCheck[i], MovementOfObject)
    hom_mat2d_scale (MovementOfObject, ScaleCheck[i], ScaleCheck[i],
                    RowCheck[i], ColumnCheck[i], MoveAndScalingOfObject)
    affine_trans_contour_xld (ShapeModel, ModelAtNewPosition,
                              MoveAndScalingOfObject)
```

After the matching, first the translational and rotational part of the transformation is determined with the operator `vector_angle_to_rigid` as in the previous sections. Then, the scaling is added using the operator `hom_mat2d_scale`. Note that the position of the match is used as the point of reference; this becomes necessary because the scaling is performed “after” the translation and rotation. The resulting, complete transformation can be used as before to display the model at the position of the matches.

### Step 3: Calculate the transformed grasping points

```
affine_trans_pixel (MoveAndScalingOfObject, RowUpperPointRef,
                  ColUpperPointRef, RowUpperPointCheck,
                  ColUpperPointCheck)
affine_trans_pixel (MoveAndScalingOfObject, RowLowerPointRef,
                  ColLowerPointRef, RowLowerPointCheck,
                  ColLowerPointCheck)
```

Of course, the affine transformation can also be applied to other points in the model image with the operator `affine_trans_pixel`. In the example, this is used to calculate the position of the grasping points for all nuts; they are marked with arrows in [figure 24d](#).

As noted in [section 4.3.2](#) on page 34, **you must use `affine_trans_pixel` and not `affine_trans_point_2d`**. 

## 5 Miscellaneous

### 5.1 Adapting to a Changed Camera Orientation

As shown in the sections above, HALCON's shape-based matching allows to localize objects even if their position and orientation in the image or their scale changes. However, the shape-based matching fails if the camera observes the scene under an oblique angle, i.e., if it is not pointed perpendicularly at the plane in which the objects move, because an object then appears distorted due to perspective projection; even worse, the distortion changes with the position and orientation of the object.

In such a case we recommend to rectify images *before* applying the matching. This is a three-step process: First, you must *calibrate* the camera, i.e., determine its position and orientation and other parameters, using the operator `camera_calibration`. Secondly, the calibration data is used to create a mapping function via the operator `gen_image_to_world_plane_map`, which is then applied to images with the operator `map_image`. For detailed information please refer to the [Application Note on 3D Machine Vision](#).

### 5.2 Reusing Models

If you want to reuse created models in other HALCON applications, all you need to do is to store the relevant information in files and then read it again. The following example code stems from the HDevelop program `hdevelop\reuse_model.dev`. First, a model is created:

```
create_scaled_shape_model (ImageROI, 0, -rad(30), rad(60), 0, 0.6, 1.4, 0,
                          'none', 'use_polarity', 60, 10, ModelID)
```

Then, the model is stored in a file using the operator `write_shape_model`. With the model, HALCON automatically saves the XLD contour, the reference point, and the parameters that were used in the call to `create_shape_model`.

```
write_shape_model (ModelID, ModelFile)
```

In the example program, all shape models are cleared to represent the start of another application.

The model, the XLD contour, and the reference point are now read from the files using the operator `read_shape_model`. Then, the XLD contours and the reference are accessed using `get_shape_model_contours` and `get_shape_model_origin`, respectively. Furthermore, the parameters used to create the model are accessed with the operator `get_shape_model_params`:

```
read_shape_model (ModelFile, ReusedModelID)
get_shape_model_contours (ReusedShapeModel, ReusedModelID, 1)
get_shape_model_origin (ReusedModelID, ReusedRefPointRow,
                        ReusedRefPointCol)
get_shape_model_params (ReusedModelID, NumLevels, AngleStart, AngleExtent,
                        AngleStep, ScaleMin, ScaleMax, ScaleStep, Metric,
                        MinContrast)
```

Now, the model can be used as if it was created in the application itself:

```
find_scaled_shape_model (SearchImage, ReusedModelID, AngleStart,
                        AngleExtent, ScaleMin, ScaleMax, 0.9, 0, 0,
                        'interpolation', 0, 0.8, RowCheck, ColumnCheck,
                        AngleCheck, ScaleCheck, Score)
for i := 0 to |Score| - 1 by 1
    vector_angle_to_rigid (ReusedRefPointRow, ReusedRefPointCol, 0,
                        RowCheck[i], ColumnCheck[i], AngleCheck[i],
                        MovementOfObject)
    hom_mat2d_scale (MovementOfObject, ScaleCheck[i], ScaleCheck[i],
                    RowCheck[i], ColumnCheck[i], MoveAndScalingOfObject)
    affine_trans_contour_xld (ReusedShapeModel, ModelAtNewPosition,
                            MoveAndScalingOfObject)
    dev_display (ModelAtNewPosition)
endfor
```