

# Adaptive Vision Studio 4.10

---

## Introduction

Created: 24.08.2018

Product version: 4.10.2.62669

Table of content:

- Image Processing
- Blob Analysis
- 1D Edge Detection
- 1D Edge Detection – Subpixel Precision
- Shape Fitting
- Template Matching
- Using Local Coordinate Systems
- Camera Calibration
- Calibration of World Coordinates
- Golden Template

# Image Processing

## Introduction

There are two major goals of Image Processing techniques:

1. To enhance an image for better human perception
2. To make the information it contains more salient or easier to extract

It should be kept in mind that in the context of computer vision only the second point is important. Preparing images for human perception is not part of computer vision; it is only part of information visualization. In typical machine vision applications this comes only at the end of the program and usually does not pose any problem.

The first and the most important advice for machine vision engineers is: **avoid image transformations designed for human perception when the goal is to extract information**. Most notable examples of transformations that are not only not interesting, but can even be highly disruptive, are:

- JPEG compression (creates artifacts not visible by human eye, but disruptive for algorithms)
- CIE Lab and CIE XYZ color spaces (specifically designed for human perception)
- Edge enhancement filters (which improve only the "apparent sharpness")
- Image thresholding performed before edge detection (precludes sub-pixel precision)

Examples of image processing operations that can really improve information extraction are:

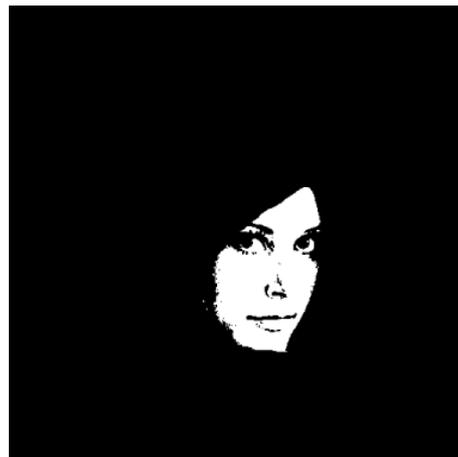
- Gaussian image smoothing (removes noise, while preserving information about local features)
- Image morphology (can remove unwanted details)
- Gradient and high-pass filters (highlight information about object contours)
- Basic color space transformations like HSV (separate information about chromaticity and brightness)
- Pixel-by-pixel image composition (e.g. can highlight image differences in relation to a reference image)

## Regions of Interest

The image processing tools provided by Adaptive Vision have a special *inRoi* input (of **Region** type), that can limit the spatial scope of the operation. The region can be of any shape.



*An input image and the inRoi.*



*Result of an operation performed within inRoi.*

Remarks:

- The output image will be black outside of the *inRoi* region.
- To obtain an image that has its pixels modified in *inRoi* and copied outside of it, one can use the **ComposeImages** filter.
- The default value for *inRoi* is *Auto* and causes the entire image to be processed.
- Although *inRoi* can be used to significantly speed up processing, it should be used with care. The performance gain may be far from proportional to the *inRoi* area, especially in comparison to processing the entire image (*Auto*). This is due to the fact, that in many cases more SSE optimizations are possible when *inRoi* is not used.

Some filters have a second region of interest called *inSourceRoi*. While *inRoi* defines the range of pixels that will be written in the output image, the *inSourceRoi* parameter defines the range of pixels that can be read from the input image.

## Image Boundary Processing

Some image processing filters, especially those from the **ImageLocalTransforms** category, use information from some local neighborhood of a pixel. This causes a problem near the image borders as not all input data is available. The policy applied in our tools is:

- Never assume any specific value outside of the image, unless specifically defined by the user.
- If only partial information is available, it is better not to detect anything, than detect something that does not exist.

In particular, the filters that use information from a local neighborhood just use smaller (cropped) neighbourhood near the image borders. This is something, however, that has to be taken into account, when relying on the results – for example results of the smoothing filters can be up to 2 times less smooth at the image borders (due to half of the neighborhood size), whereas results of the morphological filters may "stick" to the image borders. If the highest reliability is required, the general rule is: **use appropriate regions of interest to ignore image processing results that come from incomplete information** (near the image borders).

## Toolset

## Image Combinators

The filters from the **ImageCombinators** category take two images and perform a pixel-by-pixel transformation into a single image. This can be used for example to highlight differences between images or to normalize brightness – as in the example below:



*Input image with high reflections.*



*Image of the reflections (calibrating).*

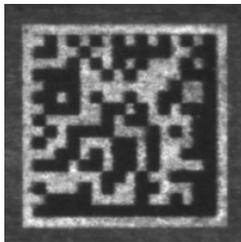


*The result of applying **DivideImages** with `inScale = 128` (`inRoi` was used).*

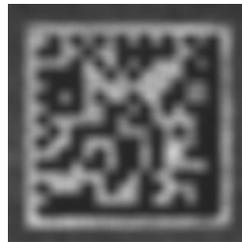
## Image Smoothing

The main purpose of the image smoothing filters (located in the **ImageLocalTransforms** category) is removal of noise. There are several different ways to perform this task with different trade-offs. On the example below three methods are presented:

1. Mean smoothing – simply takes the average pixel value from a rectangular neighborhood; it is the fastest method.
2. Median smoothing – simply takes the median pixel value from a rectangular neighborhood; preserves edges, but is relatively slow.
3. Gauss smoothing – computes a weighted average of the pixel values with Gaussian coefficients as the weights; its advantage is isotropy and reasonable speed for small kernels.



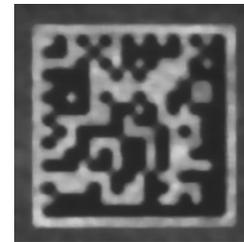
*Input image with some noise.*



*Result of applying **SmoothImage\_Mean**.*



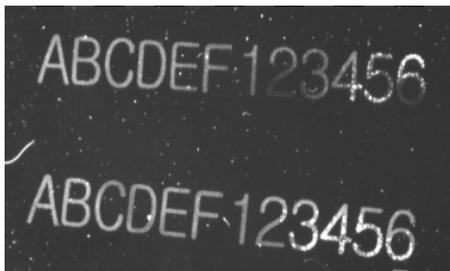
*Result of applying **SmoothImage\_Gauss**.*



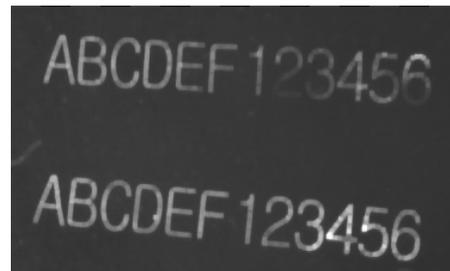
*Result of applying **SmoothImage\_Median**.*

## Image Morphology

Basic morphological operators – **DilateImage** and **ErodeImage** – transform the input image by choosing maximum or minimum pixel values from a local neighborhood. Other morphological operators combine these two basic operations to perform more complex tasks. Here is an example of using the **OpenImage** filter to remove salt and pepper noise from an image:



*Input image with salt-and-pepper noise.*



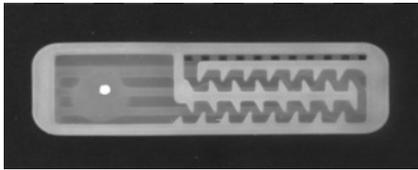
*Result of applying **OpenImage**.*

## Gradient Analysis

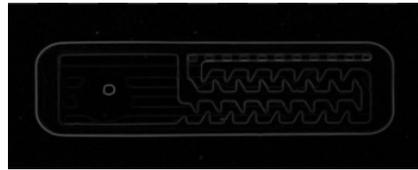
An image gradient is a vector describing direction and magnitude (strength) of local brightness changes. Gradients are used inside of many computer vision tools – for example in object contour detection, edge-based template matching and in barcode and DataMatrix detection.

Available filters:

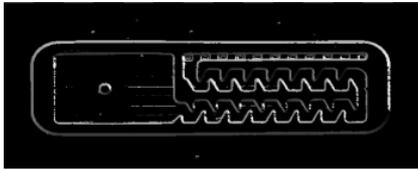
- **GradientImage** – produces a 2-channel image of signed values; each pixel denotes a gradient vector.
- **GradientMagnitudesImage** – produces a single channel image of gradient magnitudes, i.e. the lengths of the vectors (or their approximations).
- **GradientDirAndPresenceImage** – produces a single channel image of gradient directions mapped into the range from 1 to 255; 0 means no significant gradient.



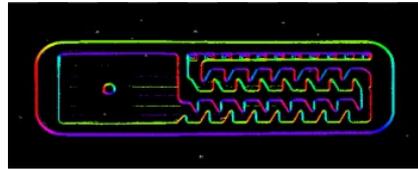
An input image.



Result of **GradientMagnitudeImage**.



Result of **GradientDirAndPresenceImage**.



Diagnostic output of **GradientImage** showing hue-coded directions.

### Spatial Transforms

Spatial transforms modify an image by changing locations, but not values, of pixels. Here are sample results of some of the most basic operations:



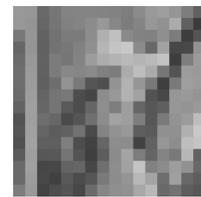
Result of **MirrorImage**.



Result of **RotateImage**.



Result of **ShearImage**.



Result of **DownsampleImage**.



Result of **TransposeImage**.



Result of **TranslateImage**.



Result of **CropImage**.



Result of **UncropImage** applied to the result of **CropImage**.

There are also interesting spatial transform tools that allow to transform a two dimensional vision problem into a 1.5-dimensional one, which can be very useful for further processing:



An input image and a path.



Result of **ImageAlongPath**.

### Spatial Transform Maps

The spatial transform tools perform a task that consist of two steps for each pixel:

1. compute the destination coordinates (and some coefficients when interpolation is used),
2. copy the pixel value.

In many cases the transformation is constant – for example we might be rotating an image always by the same angle. In such cases the first step – computing the coordinates and coefficients – can be done once, before the main loop of the program. Adaptive Vision provides the **ImageSpatialTransformsMaps** category of filters for exactly that purpose. When you are able to compute the transform beforehand, in the main loop only the **RemapImage** filter has to be executed, which is much faster than the standard spatial transform tools.

## Image Thresholding

The task of **Image Thresholding** filters is to classify image pixel values as foreground (white) or background (black). The basic filters **ThresholdImage** and **ThresholdToRegion** use just a simple range of pixel values – a pixel value is classified as foreground iff it belongs to the range. The **ThresholdImage** filter just transforms an image into another image, whereas the **ThresholdToRegion** filter creates a region corresponding to the foreground pixels. Other available filters allow more advanced classification:

- **ThresholdImage\_Dynamic** and **ThresholdToRegion\_Dynamic** use average local brightness to compensate global illumination variations.
- **ThresholdImage\_RGB** and **ThresholdToRegion\_RGB** select pixel values matching a range defined in the RGB (the standard) color space.
- **ThresholdImage\_HSx** and **ThresholdToRegion\_HSx** select pixel values matching a range defined in the HSx color space.
- **ThresholdImage\_Relative** and **ThresholdToRegion\_Relative** allow to use a different threshold value at each pixel location.



Input image with uneven light.



Result of **ThresholdImage** – the bars can not be recognized.



Result of **ThresholdImage\_Dynamic** – the bars are correct.

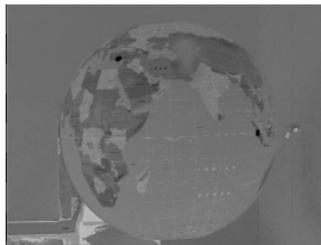
There is also an additional filter **SelectThresholdValue** which implements a number of methods for automatic threshold value selection. It should, however, be used with much care, because there is no universal method that works in all cases and even a method that works well for a particular case might fail in special cases.

## Image Pixel Analysis

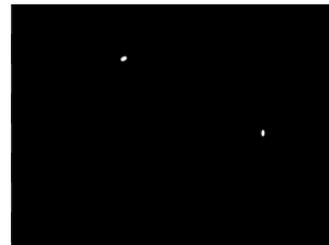
When reliable object detection by color analysis is required, there are two filters that can be useful: **ColorDistance** and **ColorDistanceImage**. These filters compare colors in the RGB space, but internally separate analysis of brightness and chromaticity. This separation is very important, because in many cases variations in brightness are much higher than variations in chromaticity. Assigning more significance to the latter (high value of the *inChromaAmount* input) allows to detect areas having the specified color even in presence of highly uneven illumination:



Input image with uneven light.



Result of **ColorDistanceImage** for the red color with *inChromaAmount* = 1.0. Dark areas correspond to low color distance.



Result of thresholding reveals the location of the red dots on the globe.

## Image Features

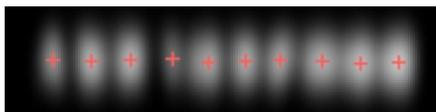
**ImageFeatures** is a category of image processing tools that are already very close to computer vision – they transform pixel information into simple higher-level data structures. Most notable examples are: **ImageLocalMaxima** which finds the points at which the brightness is locally the highest, **ImageProjection** which creates a profile from sums of pixel values in columns or in rows, **ImageAverage** which averages pixel values in the entire region of interest. Here is an example application:



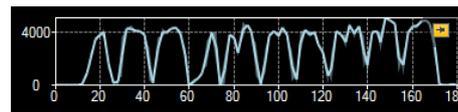
Input image with digits to be segmented.



Result of preprocessing with **CloseImage**.



Digit locations extracted by applying **SmoothImage\_Gauss** and **ImageLocalMaxima**.



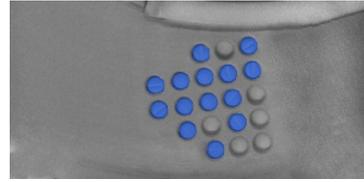
Profile of the vertical projection revealing regions of digits and the boundaries between them.

# Blob Analysis

## Introduction

*Blob Analysis* is a fundamental technique of machine vision based on analysis of consistent image regions. As such it is a tool of choice for applications in which the objects being inspected are clearly discernible from the background. Diverse set of Blob Analysis methods allows to create tailored solutions for a wide range of visual inspection problems.

Main advantages of this technique include high flexibility and excellent performance. Its limitations are: clear background-foreground relation requirement (see [Template Matching](#) for an alternative) and pixel-precision (see [1D Edge Detection](#) for an alternative).



## Concept

Let us begin by defining the notions of *region* and *blob*.

- *Region* is any subset of image pixels. In Adaptive Vision Studio regions are represented using Region data type.
- *Blob* is a connected region. In Adaptive Vision Studio blobs (being a special case of region) are represented using the same Region data type. They can be obtained from any region using a single **SplitRegionIntoBlobs** filter or (less frequently) directly from an image using image segmentation filters from category **ImageAnalysis** techniques.



*An example image.*



*Region of pixels darker than 128.*



*Decomposition of the region into array of blobs.*

The basic scenario of the Blob Analysis solution consists of the following steps:

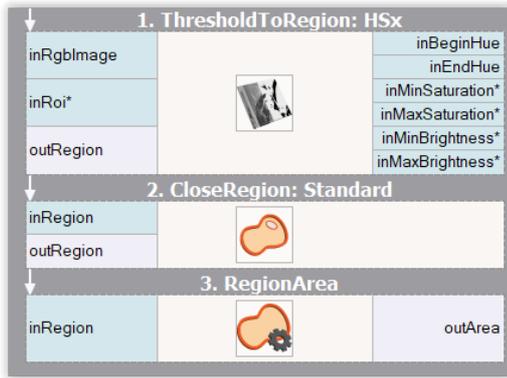
1. **Extraction** - in the initial step one of the Image Thresholding techniques is applied to obtain a region corresponding to the objects (or single object) being inspected.
2. **Refinement** - the extracted region is often flawed by noise of various kind (e.g. due to inconsistent lightning or poor image quality). In the Refinement step the region is enhanced using region transformation techniques.
3. **Analysis** - in the final step the refined region is subject to measurements and the final results are computed. If the region represents multiple objects, it is split into individual blobs each of which is inspected separately.

## Examples

The following examples illustrate the general schema of Blob Analysis algorithms. Each of the techniques represented in the examples (thresholding, morphology, calculation of region features, etc.) is inspected in detail in later sections.

### Rubber Band

In this, idealized, example we analyze a picture of an electronic device wrapped in a rubber band. The aim here is to compute the area of the visible part of the band (e.g. to decide whether it was assembled correctly).

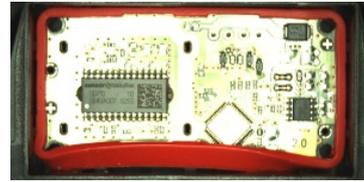


In this case each of the steps: Extraction, Refinement and Analysis is represented by a single filter.

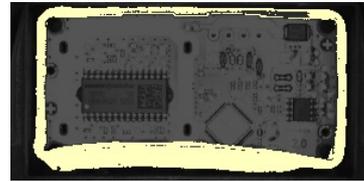
**Extraction** - to obtain a region corresponding to the red band a Color-based Thresholding technique is applied. The **ThresholdToRegion\_HSx** filter is capable of finding the region of pixels of given color characteristics - in this case it is targeted to detect red pixels.

**Refinement** - the problem of filling the gaps in the extracted region is a standard one. Classic solutions for it are the region morphology techniques. Here, the **CloseRegion** filter is used to fill the gaps.

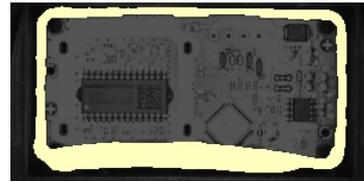
**Analysis** - finally, a single **RegionArea** filter is used to compute the area of the obtained region.



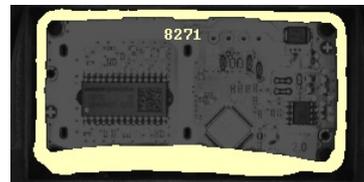
Initial image



Extraction



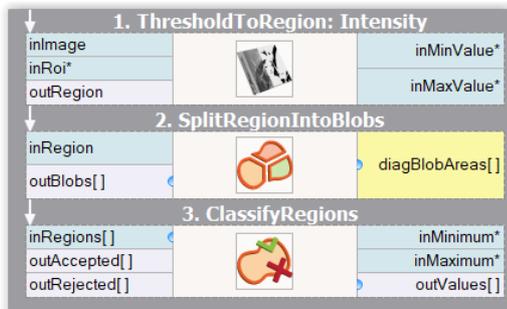
Refinement



Results

## Mounts

In this example a picture of a set of mounts is inspected to identify the damaged ones.



**Extraction** - as the lightning in the image is uniform, the objects are consistently dark and the background is consistently bright, the extraction of the region corresponding to the objects is a simple task. A basic **ThresholdToRegion** filter does the job, and does it so well that no **Refinement** phase is needed in this example.

**Analysis** - as we need to analyze each of the blobs separately, we start by applying the **SplitRegionIntoBlobs** filter to the extracted region.

To distinguish the bad parts from the correct parts we need to pick a property of a region (e.g. area, circularity, etc.) that we expect to be high for the good parts and low for the bad parts (or conversely). Here, the area would do, but we will pick a somewhat more sophisticated rectangularity feature, which will compute the similarity-to-rectangle factor for each of the blobs.

Once we have chosen the rectangularity feature of the blobs, all that needs to be done is to feed the regions to be classified to the **ClassifyRegions** filter (and to set its **inMinimum** value parameter). The blobs of too low rectangularity are available at the **outRejected** output of the classifying filter.



Input image



Extraction



Analysis



Results

## Extraction

There are two techniques that allow to extract regions from an image:

- **Image Thresholding** - commonly used methods that compute a region as a set of pixels that meet certain condition dependent on the specific operator (e.g. region of pixels brighter than given value, or brighter than the average brightness in their neighborhood). Note that the resulting data is always a single region, possibly representing numerous objects.
- **Image Segmentation** - more specialized set of methods that compute a set of blobs corresponding to areas in the image that meet certain condition. The resulting data is always an array of connected regions (blobs).

### Thresholding

Image Thresholding techniques are preferred for common applications (even those in which a set of objects is inspected rather than a single object) because of their simplicity and excellent performance. In Adaptive Vision Studio there are six filters for image-to-region thresholding, each of them implementing a different thresholding method.

Brightness-based (basic)	<table border="1"> <thead> <tr> <th colspan="3">ThresholdToRegion</th> </tr> </thead> <tbody> <tr> <td>inImage</td> <td rowspan="3"></td> <td>inMinValue</td> </tr> <tr> <td>inRoi</td> <td>inMaxValue</td> </tr> <tr> <td>outRegion</td> <td></td> </tr> </tbody> </table>	ThresholdToRegion			inImage		inMinValue	inRoi	inMaxValue	outRegion		<table border="1"> <thead> <tr> <th colspan="3">ThresholdToRegion_Dynamic</th> </tr> </thead> <tbody> <tr> <td>inImage</td> <td rowspan="3"></td> <td>inMinRelativeValue</td> </tr> <tr> <td>inSourceRoi</td> <td rowspan="2">inMaxRelativeValue</td> </tr> <tr> <td>diagBaseImage</td> </tr> <tr> <td>outRegion</td> <td></td> <td></td> </tr> </tbody> </table>	ThresholdToRegion_Dynamic			inImage		inMinRelativeValue	inSourceRoi	inMaxRelativeValue	diagBaseImage	outRegion									
	ThresholdToRegion																														
inImage		inMinValue																													
inRoi		inMaxValue																													
outRegion																															
ThresholdToRegion_Dynamic																															
inImage		inMinRelativeValue																													
inSourceRoi		inMaxRelativeValue																													
diagBaseImage																															
outRegion																															
Brightness-based (additional)	<table border="1"> <thead> <tr> <th colspan="3">ThresholdToRegion_Relative</th> </tr> </thead> <tbody> <tr> <td>inImage</td> <td rowspan="3"></td> <td>inMinRelativeValue</td> </tr> <tr> <td>inRoi</td> <td rowspan="2">inMaxRelativeValue</td> </tr> <tr> <td>inBaseImage</td> </tr> <tr> <td>outRegion</td> <td></td> <td></td> </tr> </tbody> </table>	ThresholdToRegion_Relative			inImage		inMinRelativeValue	inRoi	inMaxRelativeValue	inBaseImage	outRegion																				
	ThresholdToRegion_Relative																														
inImage		inMinRelativeValue																													
inRoi		inMaxRelativeValue																													
inBaseImage																															
outRegion																															
Color-based	<table border="1"> <thead> <tr> <th colspan="3">ThresholdToRegion_RGB</th> </tr> </thead> <tbody> <tr> <td>inRgbImage</td> <td rowspan="6"></td> <td>inMinRed</td> </tr> <tr> <td rowspan="4">inRoi</td> <td>inMaxRed</td> </tr> <tr> <td>inMinGreen</td> </tr> <tr> <td>inMaxGreen</td> </tr> <tr> <td>inMinBlue</td> </tr> <tr> <td>outRegion</td> <td>inMaxBlue</td> </tr> </tbody> </table>	ThresholdToRegion_RGB			inRgbImage		inMinRed	inRoi	inMaxRed	inMinGreen	inMaxGreen	inMinBlue	outRegion	inMaxBlue	<table border="1"> <thead> <tr> <th colspan="3">ThresholdToRegion_HSx</th> </tr> </thead> <tbody> <tr> <td>inRgbImage</td> <td rowspan="6"></td> <td>inBeginHue</td> </tr> <tr> <td rowspan="4">inRoi</td> <td>inEndHue</td> </tr> <tr> <td>inMinSaturation</td> </tr> <tr> <td>inMaxSaturation</td> </tr> <tr> <td>inMinBrightness</td> </tr> <tr> <td>diagHSxImage</td> <td>inMaxBrightness</td> </tr> <tr> <td>outRegion</td> <td></td> <td></td> </tr> </tbody> </table>	ThresholdToRegion_HSx			inRgbImage		inBeginHue	inRoi	inEndHue	inMinSaturation	inMaxSaturation	inMinBrightness	diagHSxImage	inMaxBrightness	outRegion		
	ThresholdToRegion_RGB																														
inRgbImage		inMinRed																													
inRoi		inMaxRed																													
		inMinGreen																													
		inMaxGreen																													
		inMinBlue																													
outRegion		inMaxBlue																													
ThresholdToRegion_HSx																															
inRgbImage		inBeginHue																													
inRoi		inEndHue																													
		inMinSaturation																													
		inMaxSaturation																													
		inMinBrightness																													
diagHSxImage		inMaxBrightness																													
outRegion																															

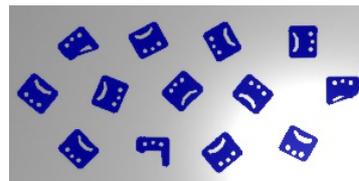
### Classic Thresholding

**ThresholdToRegion** simply selects the image pixels of the specified brightness. It should be considered a basic tool and applied whenever the intensity of the inspected object is constant, consistent and clearly different from the intensity of the background.



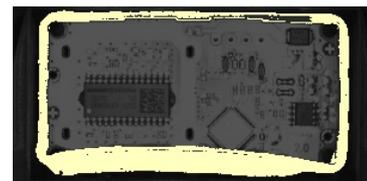
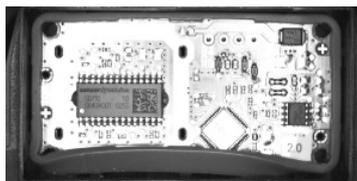
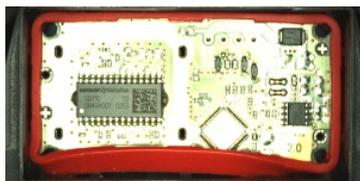
### Dynamic Thresholding

Inconsistent brightness of the objects being inspected is a common problem usually caused by the imperfections of the lighting setup. As we can see in the example below, it is often the case that the objects in one part of the image actually have the same brightness as the background in another part of the image. In such case it is not possible to use the basic **ThresholdToRegion** filter and **ThresholdToRegion\_Dynamic** should be considered instead. The latter selects image pixels that are *locally* bright/dark. Specifically - the filter selects the image pixels of the given *relative local brightness* defined as the difference between the pixel intensity and the average intensity in its neighborhood.



### Color-based Thresholding

When inspection is conducted on color images it may be the case that despite a significant difference in color, the brightness of the objects is actually the same as the brightness of their neighborhood. In such case it is advisable to use Color-based Thresholding filters: **ThresholdToRegion\_RGB**, **ThresholdToRegion\_HSx**. The suffix denote the color space in which we define the desired pixel characteristic and not the space used in the image representation. In other words - both of these filters can be used to process standard RGB color image.



An example image.

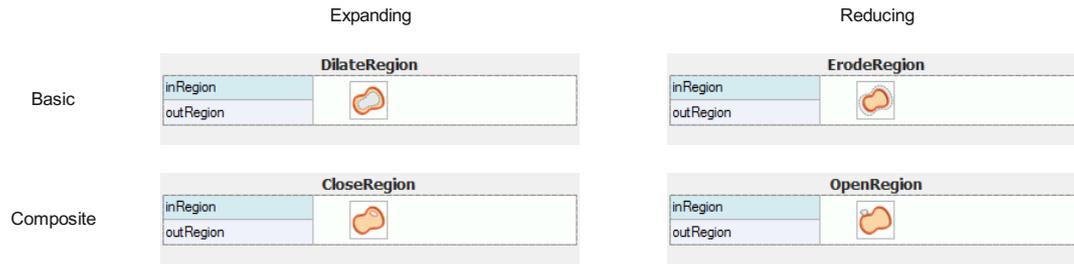
Mono equivalent of the image depicting brightness of its pixels.

Result of the color-based thresholding targeted at red pixels.

### Refinement

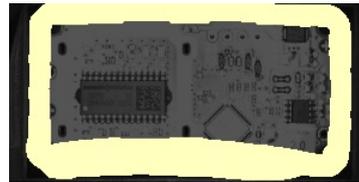
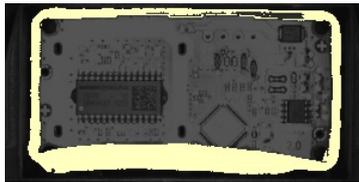
## Region Morphology

Region Morphology is a classic technique of region transformation. The core concept of this toolset is the usage of a *structuring element* also known as the *kernel*. The kernel is a relatively small shape that is repeatedly centered at each pixel within dimensions of the region that is being transformed. Every such pixel is either added to the resulting region or not, depending on operation-specific condition on the minimum number of kernel pixels that have to overlap with actual input region pixels (in the given position of the kernel). See description of **Dilation** for an example.

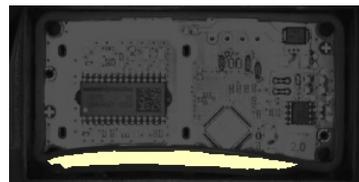
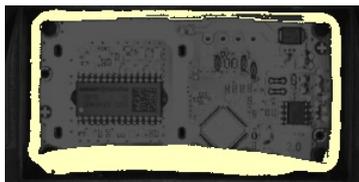


### Dilation and Erosion

**Dilation** is one of two basic morphological transformations. Here each pixel **P** within the dimensions of the region being transformed is added to the resulting region if and only if the structuring element centered at **P** overlaps with *at least* one pixel that belongs to the input region. Note that for a circular kernel such transformation is equivalent to a uniform expansion of the region in every direction.

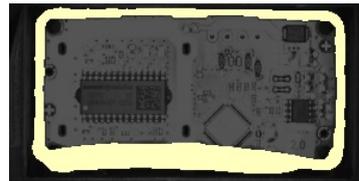
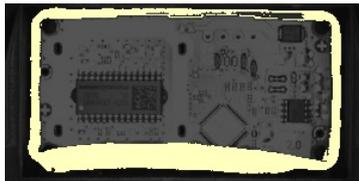


**Erosion** is a dual operation of **Dilation**. Here, each pixel **P** within the dimensions of the region being transformed is added to the resulting region if and only if the structuring element centered at **P** is *fully contained* in the region pixels. Note that for a circular kernel such transformation is equivalent to a uniform reduction of the region in every direction.

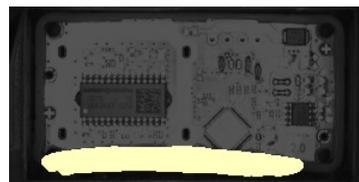
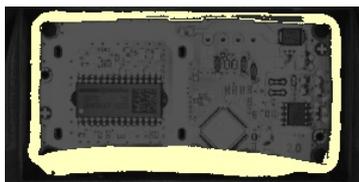


### Closing and Opening

The actual power of the **Region Morphology** lies in its *composite operators* - **Closing** and **Opening**. As we may have recently noticed, during the blind region expansion performed by the **Dilation** operator, the gaps in the transformed region are filled in. Unfortunately, the expanded region no longer corresponds to the objects being inspected. However, we can apply the **Erosion** operator to bring the expanded region back to its original boundaries. The key point is that the gaps that were completely filled during the dilation will stay filled after the erosion. The operation of applying **Erosion** to the result of **Dilation** of the region is called **Closing**, and is a tool of choice for the task of filling the gaps in the extracted region.



**Opening** is a dual operation of **Closing**. Here, the region being transformed is initially eroded and then dilated. The resulting region preserves the form of the initial region, with the exception of thin/small parts, that are removed during the process. Therefore, **Opening** is a tool for removing the thin/outlying parts from a region. We may note that in the example below, the **Opening** does the - otherwise relatively complicated - job of finding the segment of the rubber band of excessive width.



### Other Refinement Methods

#### Analysis

Once we obtain the region that corresponds to the object or the objects being inspected, we may commence the analysis - that is, extract the information we are interested in.

## Region Features

Adaptive Vision Studio allows to compute a wide range of numeric (e.g. area) and non-numeric (e.g. bounding circle) region features. Calculation of the measures describing the obtained region is often the very aim of applying the blob analysis in the first place. If we are to check whether the rectangular packaging box is deformed or not, we may be interested in calculating the *rectangularity factor* of the packaging region. If we are to check if the chocolate coating on a biscuit is broad enough, we may want to know the *area* of the coating region.

It is important to remember, that when the obtained region corresponds to multiple image objects (and we want to inspect each of them separately), we should apply the **SplitRegionIntoBlobs** filter before performing the calculation of features.

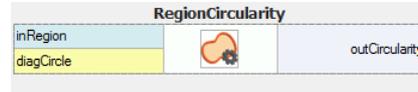
### Numeric Features

Each of the following filters computes a number that expresses a specific property of the region shape.

Annotations in brackets indicate the range of the resulting values.



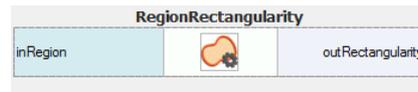
Size of the region (0 -  $\infty$ )



Similarity to a circle (0.0 - 1.0)



Similarity to own convex hull (0.0 - 1.0)



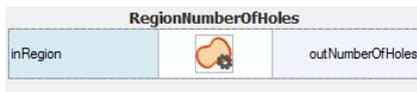
Similarity to a rectangle (0.0 - 1.0)



Similarity to a line (0.0 -  $\infty$ )



Moments of the region (0.0 -  $\infty$ )



Count of the region holes (0 -  $\infty$ )



Orientation of the main region axis (0.0 - 180.0)



Length of the region contour (0.0 -  $\infty$ )

### Non-numeric Features

Each of the following filters computes an object related to the shape of the region. Note that the primitives extracted using these filters can be made subject of further analysis. For instance, we can extract the holes of the region using the **RegionHoles** filter and then measure their areas using the **RegionArea** filter.

Annotations in brackets indicate Adaptive Vision Studio's type of the result.



Smallest axis-aligned rectangle containing the region (Box)



Smallest circle containing the region (Circle2D)



Smallest any-orientation rectangle containing the region (Rectangle2D)



Boundaries of the region (PathArray)



Longest segment connecting two points inside the region (Segment2D)



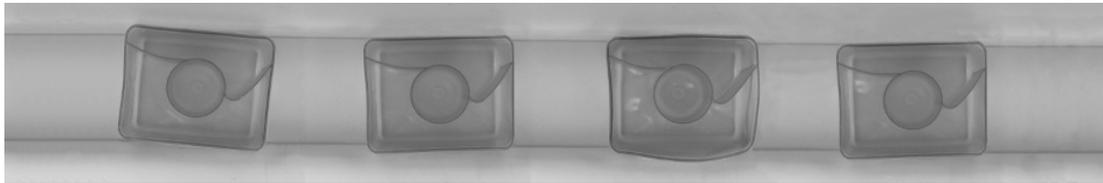
Array of blobs representing gaps in the region (RegionArray)



Skeleton of the region (PathArray)

## Case Studies

### Capsules

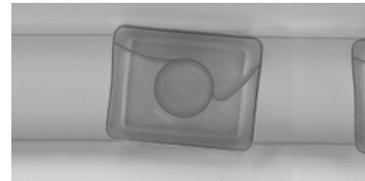
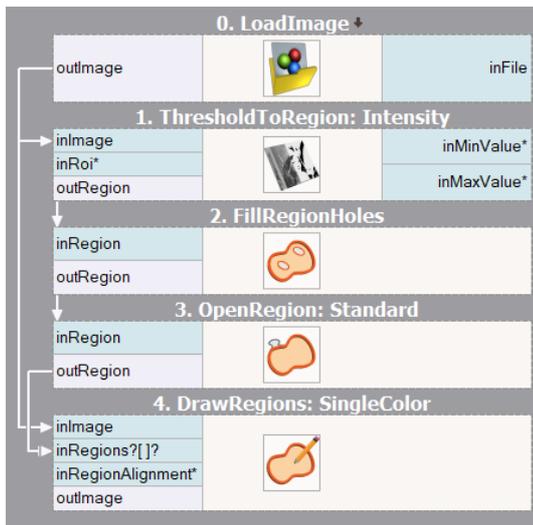


In this example we inspect a set of washing machine capsules on a conveyor line. Our aim is to identify the deformed capsules.

We will proceed in two steps: we will commence by designing a simple program that, given picture of the conveyor line, will be able to identify the region corresponding to the capsule(s) in the picture. In the second step we will use this program as a building block of the complete solution.

### FindRegion Routine

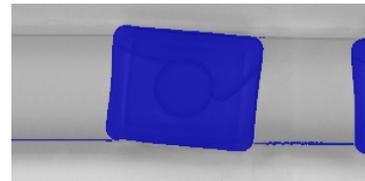
In this section we will develop a program that will be responsible for the **Extraction** and **Refinement** phases of the final solution. For brevity of presentation in this part we will limit the input image to its initial segment.



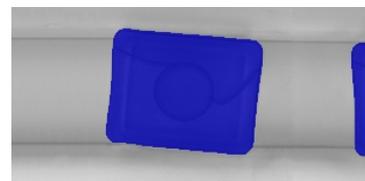
Initial image



ThresholdToRegion



FillRegionHoles



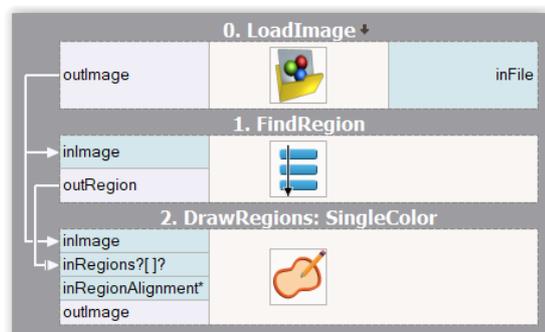
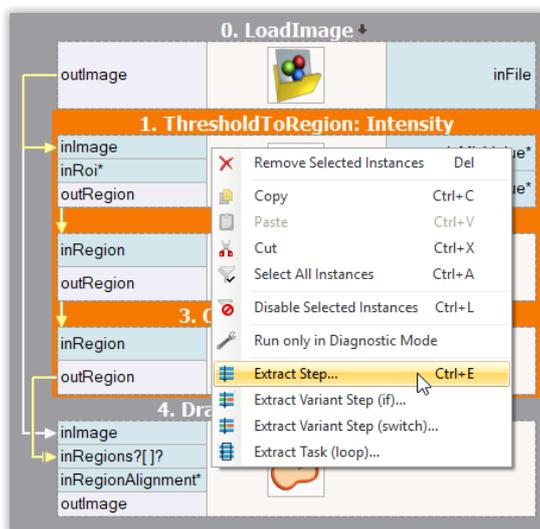
OpenRegion

After a brief inspection of the input image we may note that the task at hand will not be trivial - the average brightness of the capsule body is similar to the intensity of the background. On the other hand the border of the capsule is consistently darker than the background. As it is the border of the object that bears significant information about its shape we may use the basic **ThresholdToRegion** filter to extract the darkest pixels of the image with the intention of filling the extracted capsule border during further refinement.

The extracted region certainly requires such refinement - actually, there are two issues that need to be addressed. We need to fill the shape of the capsule and eliminate the thin horizontal stripes corresponding to the elements of the conveyor line setup. Fortunately, there are fairly straightforward solutions for both of these problems.

**FillRegionHoles** will extend the region to include all pixels enclosed by present region pixels. After the region is filled all that remains is the removal of the thin conveyor lines using the classic **OpenRegion** filter.

Our routine for **Extraction** and **Refinement** of the region is ready. As it constitutes a continuous block of filters performing a well defined task, it is advisable to encapsulate the routine inside a macrofilter to enhance the readability of the soon-to-be-growing program.



### Complete Solution

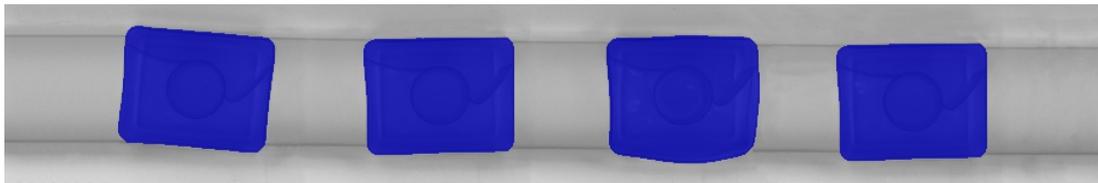
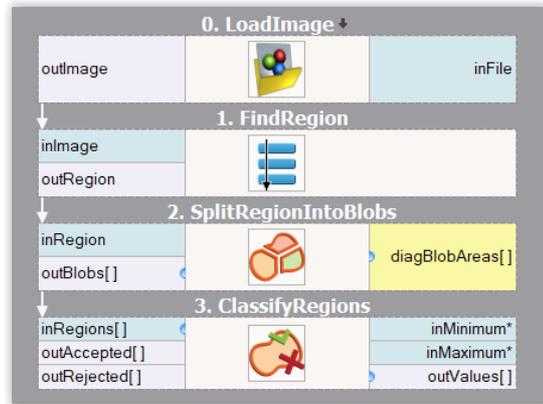
Our program right now is capable of extracting the region that directly corresponds to the capsules visible in the image. What remains is to inspect each capsule and classify it as a correct or deformed one.

As we want to analyze each capsule separately, we should start with decomposition of the extracted region into an array of connected components (blobs). This common operation can be performed using the straightforward **SplitRegionIntoBlobs** filter.

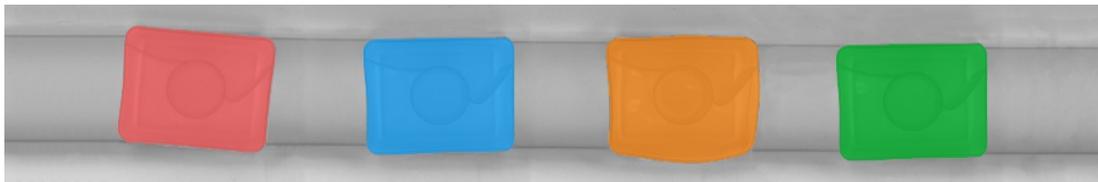
We are approaching the crucial part of our solution - how are we going to distinguish correct capsules from deformed ones? At this stage it is advisable to have a look at the summary of numeric region features provided in **Analysis** section. If we could find a numeric region property that is correlated with the nature of the problem at hand (e.g. it takes low values for a correct capsules and high values for a deformed one, or conversely), we would be nearly done.

Rectangularity of a shape is defined as the ratio between its area and area of its smallest enclosing rectangle - the higher the value, the more the shape of the object resembles a rectangle. As the shape of a correct capsule is almost rectangular (it is a rectangle with rounded corners) and clearly *more* rectangular than the shape of deformed capsule, we may consider using rectangularity feature to classify the capsules.

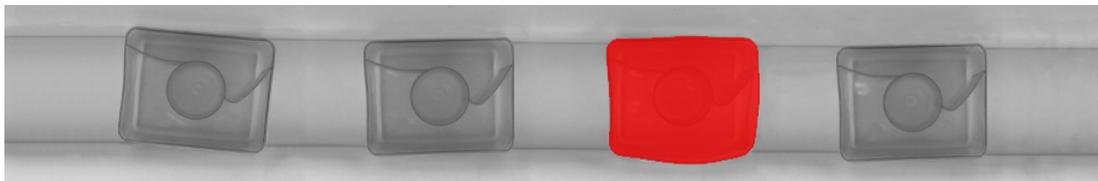
Having selected the numeric feature that will be used for the classification, we are ready to add the **ClassifyRegions** filter to our program and feed it with data. We pass the array of capsule blobs on its **inRegions** input and we select Rectangularity on the **inFeature** input. After brief interactive experimentation with the *inMinimum* threshold we may observe that setting the minimum rectangularity to 0.95 allows proper discrimination of correct (available at **outAccepted**) and deformed (**outRejected**) capsule blobs.



*Region extracted by the FindRegion routine.*



*Decomposition of the region into individual blobs.*



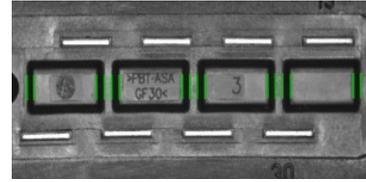
*Blobs of low rectangularity selected by **ClassifyRegions** filter.*

# 1D Edge Detection

## Introduction

*1D Edge Detection* (also called *1D Measurement*) is a classic technique of machine vision where the information about image is extracted from one-dimensional profiles of image brightness. As we will see, it can be used for measurements as well as for positioning of the inspected objects.

Main advantages of this technique include sub-pixel precision and high performance.



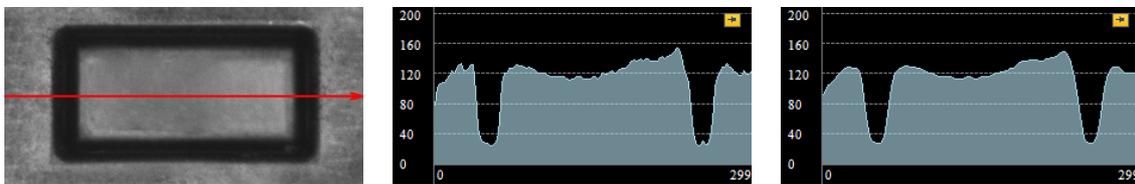
## Concept

The 1D Edge Detection technique is based on an observation that any edge in the image corresponds to a rapid brightness change in the direction perpendicular to that edge. Therefore, to detect the image edges we can scan the image along a path and look for the places of significant change of intensity in the extracted brightness profile.

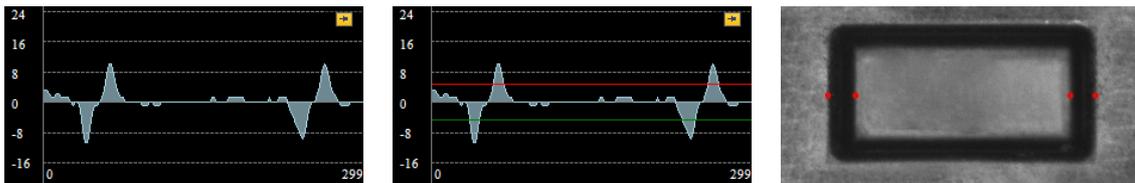
The computation proceeds in the following steps:

1. **Profile extraction** – firstly the profile of brightness along the given path is extracted. Usually the profile is smoothed to remove the noise.
2. **Edge extraction** – the points of significant change of profile brightness are identified as *edge points* – points where perpendicular edges intersect the scan line.
3. **Post-processing** – the final results are computed using one of the available methods. For instance **ScanSingleEdge** filter will select and return the strongest of the extracted edges, while **ScanMultipleEdges** filter will return all of them.

## Example



The image is scanned along the path and the brightness profile is extracted and smoothed.



Brightness profile is differentiated. Notice four peaks of the profile derivative which correspond to four prominent image edges intersecting the scan line. Finally the peaks stronger than some selected value (here minimal strength is set to 5) are identified as edge points.

## Filter Toolset

Basic toolset for the 1D Edge Detection-based techniques scanning for edges consists of 9 filters each of which runs a single scan along the given path (**inScanPath**). The filters differ on the structure of interest (edges / ridges / stripes (edge pairs)) and its cardinality (one / any fixed number / unknown number).

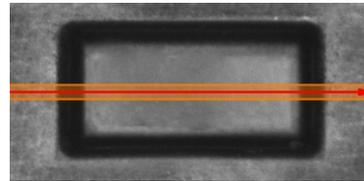
	Edges	Stripes	Ridges
Single Result	<b>ScanSingleEdge</b> inImage inScanPath inScanPath.Alignment out.AlignedScanPath diag.BrightnessProfile diag.ResponseProfile out.Edge.Point out.Edge.Magnitude	<b>ScanSingleStripe</b> inImage inScanPath inScanPath.Alignment out.Stripe out.AlignedScanPath diag.BrightnessProfile diag.ResponseProfile out.Stripe.Width out.Stripe.Magnitude	<b>ScanSingleRidge</b> inImage inScanPath inScanPath.Alignment out.AlignedScanPath diag.BrightnessProfile diag.ResponseProfile
Multiple Results	<b>ScanMultipleEdges</b> inImage inScanPath inScanPath.Alignment out.AlignedScanPath diag.BrightnessProfile diag.ResponseProfile out.Edges.Point out.Edges.Magnitude	<b>ScanMultipleStripes</b> inImage inScanPath inScanPath.Alignment out.Stripes out.AlignedScanPath diag.BrightnessProfile diag.ResponseProfile out.Stripes.Width out.Stripes.Magnitude	<b>ScanMultipleRidges</b> inImage inScanPath inScanPath.Alignment out.AlignedScanPath diag.BrightnessProfile diag.ResponseProfile
Fixed Number of Results	<b>ScanExactlyNEdges</b> inImage inScanPath inScanPath.Alignment out.AlignedScanPath diag.BrightnessProfile diag.ResponseProfile out.Edges.Point out.Edges.Magnitude	<b>ScanExactlyNStripes</b> inImage inScanPath inScanPath.Alignment out.Stripes out.AlignedScanPath diag.BrightnessProfile diag.ResponseProfile out.Stripes.Width out.Stripes.Magnitude	<b>ScanExactlyNRidges</b> inImage inScanPath inScanPath.Alignment out.AlignedScanPath diag.BrightnessProfile diag.ResponseProfile

Note that in Adaptive Vision Library there is the **CreateScanMap** function that has to be used before a usage of any other 1D Edge Detection function. The special function creates a scan map, which is passed as an input to other functions considerably speeding up the computations.

## Parameters

### Profile Extraction

In each of the nine filters the brightness profile is extracted in exactly the same way. The stripe of pixels along **inScanPath** of width **inScanWidth** is traversed and the pixel values across the path are accumulated to form one-dimensional profile. In the picture on the right the stripe of processed pixels is marked in orange, while **inScanPath** is marked in red.

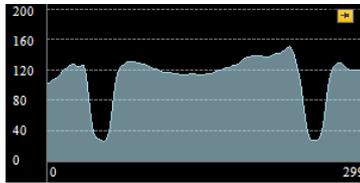


The extracted profile is smoothed using Gaussian smoothing with standard deviation of **inSmoothingStdDev**. This parameter is important for the robustness of the computation - we should pick the value that is high enough to eliminate noise that could introduce false / irrelevant extrema to the profile derivative, but low enough to preserve the actual edges we are to detect.

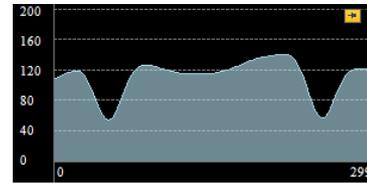
The **inSmoothingStdDev** parameter should be adjusted through interactive experimentation using **diagBrightnessProfile** output, as demonstrated below.



*Too low **inSmoothingStdDev** - too much noise*



*Appropriate **inSmoothingStdDev** - low noise, significant edges are preserved*

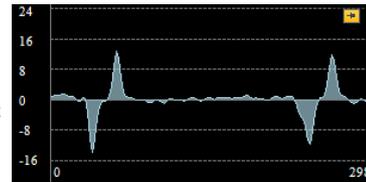


*Too high **inSmoothingStdDev** - significant edges are attenuated*

### Edge Extraction

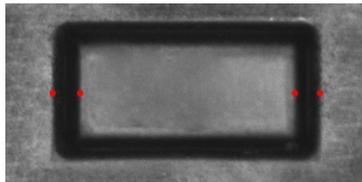
After the brightness profile is extracted and refined, the derivative of the profile is computed and its local extrema of magnitude at least **inMinMagnitude** are identified as edge points. The **inMinMagnitude** parameter should be adjusted using the **diagResponseProfile** output.

The picture on the right depicts an example **diagResponseProfile** profile. In this case the significant extrema vary in magnitude from 11 to 13, while the magnitude of other extrema is lower than 3. Therefore any **inMinMagnitude** value in range (4, 10) would be appropriate.

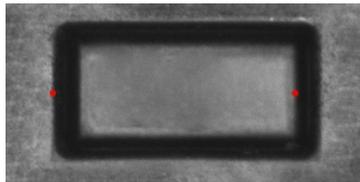


### Edge Transition

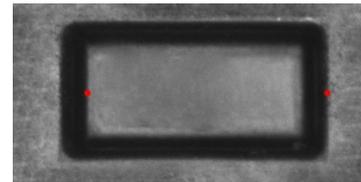
Filters being discussed are capable of filtering the edges depending on the kind of transition they represent - that is, depending on whether the intensity changes from bright to dark, or from dark to bright. The filters detecting individual edges apply the same condition defined using the **inTransition** parameter to each edge (possible choices are *bright-to-dark*, *dark-to-bright* and *any*).



***inTransition** = Any*



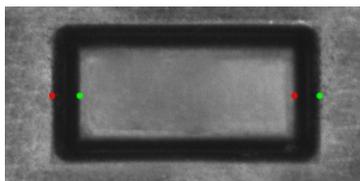
***inTransition** = BrightToDark*



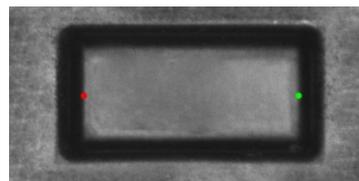
***inTransition** = DarkToBright*

### Stripe Intensity

The filters detecting stripes expect the edges to alternate in their characteristics. The parameter **inIntensity** defines whether each stripe should bound the area that is brighter, or darker than the surrounding space.



***inIntensity** = Dark*



***inIntensity** = Bright*

### Case Study: Blades



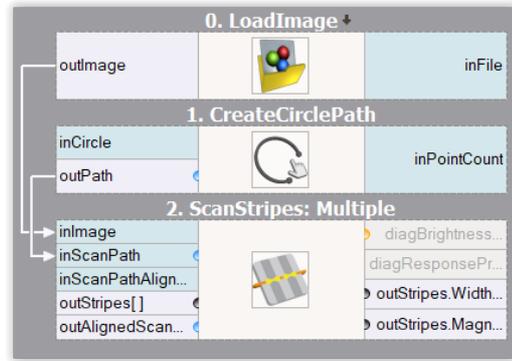
Assume we want to count the blades of a circular saw from the picture.

We will solve this problem running a single 1D Edge Detection scan along a circular path intersecting the blades, and therefore we need to produce appropriate circular path. For that we will use a straightforward **CreateCirclePath** filter. The built-in editor will allow us to point & click the required **inCircle** parameter.

The next step will be to pick a suitable measuring filter. Because the path will alternate between dark blades and white background, we will use a filter that is capable of measuring stripes. As we do not know how many blades there are on the image (that is what we need to compute), the **ScanMultipleStripes** filter will be a perfect choice.

We expect the measuring filter to identify each blade as a single stripe (or each space between blades, depending on our selection of **inIntensity**), therefore all we need to do to compute the number of blades is to read the value of the **outStripes.Count** property output of the measuring filter.

The program solves the problem as expected (perhaps after increasing the **inSmoothingStdDev** from default of 0.6 to bigger value of 1.0 or 2.0) and detects all 30 blades of the saw.



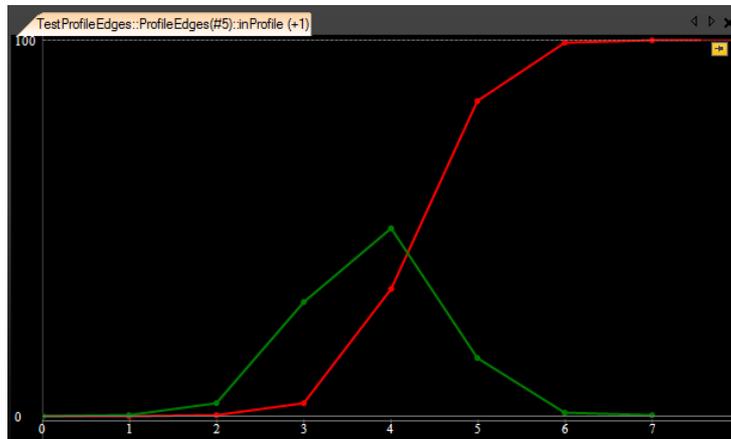
# 1D Edge Detection – Subpixel Precision

## Introduction

One of the key strengths of the 1D Edge Detection tools is their ability to detect edges with precision higher than the pixel grid. This is possible, because the values of the derivative profile (of pixel values) can be interpolated and its maxima can be found analytically.

## Example: Parabola Fitting

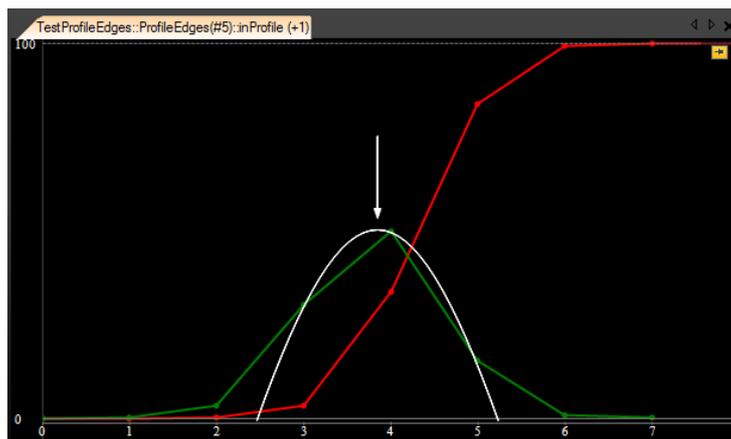
Let us consider a sample profile of pixel values corresponding to an edge (red):



Sample edge profile (red) and its derivative (green). Please note, that the derivative is shifted by 0.5.

The steepest segment is between points 4.0 and 5.0, which corresponds to the maximum of the derivative (green) at 4.5. Without the subpixel precision the edge would be found at this point.

It is, however, possible to consider information about the values of the neighbouring profile points to extract the edge location with higher precision. The simplest method is to fit a parabola to three consecutive points of the derivative profile:



Fitting a parabola to three consecutive points.

Now, the edge point we are looking for can be taken from the maximum of the parabola. In this case it will be 4.363, which is already a subpixel-precise result. This precision is still not very high, however. We know it from an experiment – this particular profile, which we are considering in this example, has been created from a perfect gaussian edge located at the point 430 and downsampled 100 times to simulate a camera looking at an edge at the point 4.3. The error that we got, is 0.063 px. From other experiments we know that in the worst case it can be up to 1/6 px.

## Advanced: Methods Available in Adaptive Vision

More advanced methods can be used that consider not three, but four consecutive points and which employ additional techniques to assure the highest precision in presence of noise and other practical edge distortions. In Adaptive Vision Studio they are available in a form of 3 different profile interpolation methods:

- *Linear* – the simplest method that results in pixel-precise results,
- *Quadratic3* – an improved fitting of parabola to 3 consecutive points,
- *Quadratic4* – an advanced method that fits parabola to 4 consecutive points.

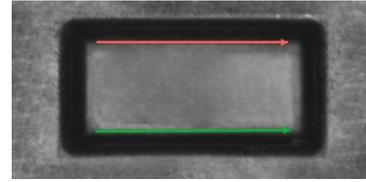
The precision of these methods on perfect gaussian edges is respectively: 1/2 px, 1/6 px and 1/23 px. It has to be added, however, that the *Quadratic4* method differs significantly in its performance on edges which are only slightly blurred – when the image quality is close to perfect, the precision can be even higher than 1/50 px.

# Shape Fitting

## Introduction

*Shape Fitting* is a machine vision technique that allows for precise detection of objects whose shapes and rough positions are known in advance. It is most often used in measurement applications for establishing line segments, circles, arcs and paths defining the shape that is to be measured.

As this technique is derived from **1DEdgeDetection**, its key advantages are similar – including sub-pixel precision and high performance.

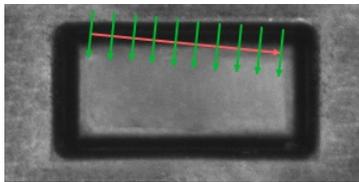


## Concept

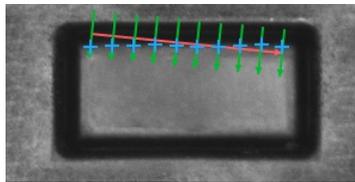
The main idea standing behind **ShapeFitting** is that a continuous object (such as a circle, an arc or a segment) can be determined using a finite set of points belonging to it. These points are computed by means of appropriate **1DEdgeDetection** filters and are then combined together into a single higher-level result.

Thus, a single **ShapeFitting** filter's work consists of the following steps:

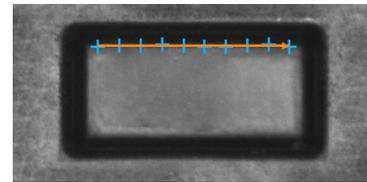
1. **Scan segments preparation** – a series of segments is prepared. The number, length and orientations of the segments are computed from the filter's parameters.
2. **Points extraction** – points that should belong to the object being fitted are extracted using (internally) a proper **1DEdgeDetection** filter (e.g. **ScanSingleEdge** in **FitCircleToEdges**) along each of the scan segments as the scan path.
3. **Object fitting** – the final result is computed with the use of a technique that allows fitting an object to a set of points. In this step, a filter from **Geometry2DFitting** is internally used (e.g. **FitCircleToPoints** in **FitCircleToEdges**). An exception to the rule is path fitting. No **Geometry2DFitting** filter is needed there, because the found points serve themselves as the output path characteristic points.



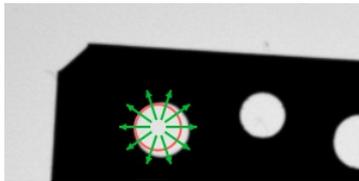
The scan segments are created according to the fitting field and other parameters (e.g. *inScanCount*).



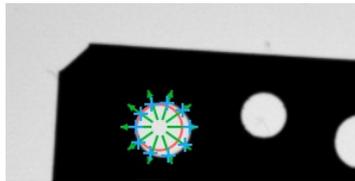
**ScanSingleEdge** (or another proper **1DEdgeDetection** filter) is performed.



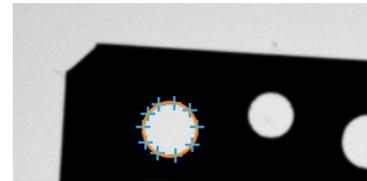
A segment is fitted to the obtained points.



The scan segments are created according to the fitting field and other parameters (e.g. *inScanCount*).



**ScanSingleEdge** (or another proper **1DEdgeDetection** filter) is performed.



A segment is fitted to the obtained points.

## Toolset

### Studio

The whole toolset for **ShapeFitting** consists of several filters. The filters differ on the object being fitted (a circle, an arc, a line segment or a path) and the **1DEdgeDetection** structures extracted along the scan segments (edges, ridges or stripe, all of them clearly discernible on the input image).

FitCircleToStripe	
inImage	
inFittingFieldAlignment	
diagScanSegments	
outCircle	
outInnerCircle	
outOuterCircle	
outStripes	
	diagBrightnessProfiles
	diagResponseProfiles
	outStripePoints

FitPathToEdges	
inImage	diagBrightnessProfiles
inFittingFieldAlignment	diagResponseProfiles
diagScanSegments	outDeviationProfile
outPath	outEdges.Point

Sample **ShapeFitting** filters.

### Library

The typical usage of the shape fitting method encompasses two distinct functions. One of the **CreateObjectFittingMap** functions (e.g. **CreateCircleFittingMap**) has to be used before any other **ShapeFitting** function. The special functions create a fitting map consisting of the scan segments. The fitting map is then passed as an input to other functions and, because it generally must be created only once for a whole series of fitting, this strategy speeds up the computations considerably. However, the fitting map must be created before every fitting when **inFittingFieldAlignment** parameter of the **CreateObjectFittingMap** function is not *Nil*.

A sample program is shown below:

```

avl::CreateCircleFittingMap(
  sampleImage,
  CircleFittingField(expectedCircle, 20.0f),
  NIL,
  10,
  1,
  InterpolationMethod::Bilinear,
  circleFittingMap
);

while (true)
{
  Image image;
  atl::Conditional<avl::Circle2D> outCircle;
  GetImageFromCamera(image); // Get images here

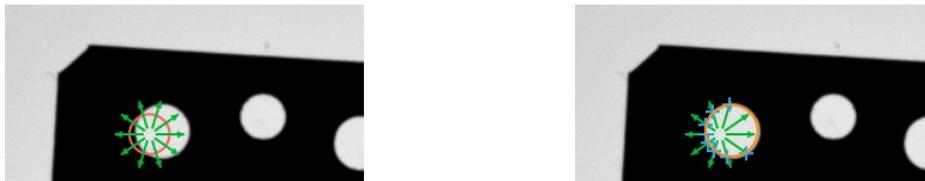
  avl::FitCircleToEdges(
    image,
    circleFittingMap,
    EdgeScanParams(),
    Selection::Best,
    NIL,
    0.1f,
    CircleFittingMethod::AlgebraicPratt,
    NIL,
    outCircle
  );
}

```

## Parameters

Because of the internal use of **1DEdgeDetection** filters and **Geometry2DFitting** filters, all parameters known from them are also present in **ShapeFitting** filters interfaces.

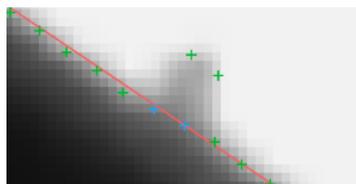
Beside these, there are also a few parameters specific to the subject of shape fitting. The **inScanCount** parameter controls the number of the scan segments. However, not all of the scans have to succeed in order to regard the whole fitting process as being successful. The **inMaxIncompleteness** parameter determines what fraction of the scans may fail.



*FitCircleToEdges* performed on the sample image with **inMaxIncompleteness** = 0.25. Although two scans have ended in failure, the circle has been fitted successfully.

The path fitting functions have some additional parameters, which help to control the output path shape. These parameters are:

- **inMaxDeviationDelta** – it defines the maximal allowed difference between deviations of consecutive points of the output path from the corresponding input path points; if the difference between deviations is greater, the point is considered to be not found at all.
- **inMaxInterpolationLength** – if some of the scans fail or if some of found points are classified to be wrong according to another control parameters (e.g. **inMaxDeviationDelta**), output path points corresponding to them are interpolated depending on points in their nearest vicinity. No more than **inMaxInterpolationLength** consecutive points can be interpolated, and if there exists a longer series of points that would have to be interpolated, the fitting is considered to be unsuccessful. The exception to this behavior are points which were not found on both ends of the input path. Those are not part of the result at all.



*FitPathToEdges* performed on the sample image with **inMaxDeviationDelta** = 2 and **inMaxInterpolationLength** = 3. Blue points are the points that were interpolated. If **inMaxInterpolationLength** value was less than 2, the fitting would have failed.

# Template Matching

## Introduction

*Template Matching* is a high-level machine vision technique that identifies the parts on an image that match a predefined template. Advanced template matching algorithms allow to find occurrences of the template regardless of their orientation and local brightness.



Template Matching techniques are flexible and relatively straightforward to use, which makes them one of the most popular methods of object localization. Their applicability is limited mostly by the available computational power, as identification of big and complex templates can be time-consuming.

## Concept

Template Matching techniques are expected to address the following need: provided a reference image of an object (the *template image*) and an image to be inspected (the *input image*) we want to identify all *input image* locations at which the object from the *template image* is present. Depending on the specific problem at hand, we may (or may not) want to identify the rotated or scaled occurrences.

We will start with a demonstration of a naive Template Matching method, which is insufficient for real-life applications, but illustrates the core concept from which the actual Template Matching algorithms stem from. After that we will explain how this method is enhanced and extended in advanced **Grayscale-based Matching** and **Edge-based Matching** routines.

## Naive Template Matching

Imagine that we are going to inspect an image of a plug and our goal is to find its pins. We are provided with a *template image* representing the reference object we are looking for and the *input image* to be inspected.



Template image



Input image

We will perform the actual search in a rather straightforward way – we will position the *template* over the image at every possible location, and each time we will compute some numeric measure of similarity between the template and the image segment it currently overlaps with. Finally we will identify the positions that yield the best similarity measures as the probable template occurrences.

## Image Correlation

One of the subproblems that occur in the specification above is calculating the *similarity measure* of the aligned template image and the overlapped segment of the input image, which is equivalent to calculating a similarity measure of two images of equal dimensions. This is a classical task, and a numeric measure of image similarity is usually called *image correlation*.

### Cross-Correlation

The fundamental method of calculating the image correlation is so called *cross-correlation*, which essentially is a simple sum of pairwise multiplications of corresponding pixel values of the images.

Though we may notice that the correlation value indeed seems to reflect the similarity of the images being compared, cross-correlation method is far from being robust. Its main drawback is that it is biased by changes in global brightness of the images - brightening of an image may sky-rocket its cross-correlation with another image, even if the second image is not at all similar.

Image1	Image2	Cross-Correlation
		19404780
		23316890
		24715810

$$\text{Cross-Correlation}(\text{Image1}, \text{Image2}) = \sum_{x,y} \text{Image1}(x, y) \times \text{Image2}(x, y)$$

### Normalized Cross-Correlation

*Normalized cross-correlation* is an enhanced version of the classic *cross-correlation* method that introduces two improvements over the original one:

- The results are invariant to the global brightness changes, i.e. consistent brightening or darkening of either image has no effect on the result (this is accomplished by subtracting the mean image brightness from each pixel value).
- The final correlation value is scaled to [-1, 1] range, so that NCC of two identical images equals 1.0, while NCC of an image and its negation equals -1.0.

Image1	Image2	NCC
		-0.417
		0.553
		0.844

$$\text{NCC}(\text{Image1}, \text{Image2}) = \frac{1}{N\sigma_1\sigma_2} \sum_{x,y} (\text{Image1}(x, y) - \overline{\text{Image1}}) \times (\text{Image2}(x, y) - \overline{\text{Image2}})$$

## Template Correlation Image

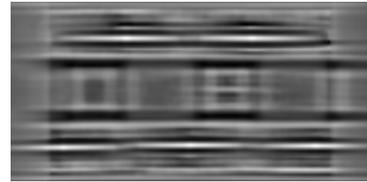
Let us get back to the problem at hand. Having introduced the Normalized Cross-Correlation - robust measure of image similarity - we are now able to determine how well the template fits in each of the possible positions. We may represent the results in a form of an image, where brightness of each pixels represents the NCC value of the template positioned over this pixel (black color representing the minimal correlation of -1.0, white color representing the maximal correlation of 1.0).



Template image



Input image



Template correlation image

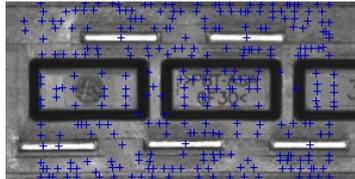
### Identification of Matches

All that needs to be done at this point is to decide which points of the *template correlation image* are good enough to be considered actual matches. Usually we identify as matches the positions that (simultaneously) represent the template correlation:

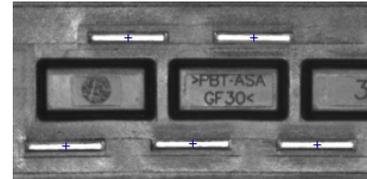
- stronger than some predefined threshold value (i.e stronger than 0.5)
- locally maximal (stronger than the template correlation in the neighboring pixels)



Areas of template correlation above 0.75



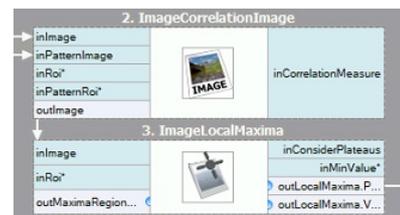
Points of locally maximal template correlation



Points of locally maximal template correlation above 0.75

### Summary

It is quite easy to express the described method in Adaptive Vision Studio - we will need just two built-in filters. We will compute the template correlation image using the **ImageCorrelationImage** filter, and then identify the matches using **ImageLocalMaxima** - we just need to set the **inMinValue** parameter that will cut-off the weak local maxima from the results, as discussed in previous section.



Though the introduced technique was sufficient to solve the problem being considered, we may notice its important drawbacks:

- Template occurrences have to preserve the orientation of the reference template image.
- The method is inefficient, as calculating the template correlation image for medium to large images is time consuming.

In the next sections we will discuss how these issues are being addressed in advanced template matching techniques: **Grayscale-based Matching** and **Edge-based Matching**.

### Grayscale-based Matching, Edge-based Matching

**Grayscale-based Matching** is an advanced Template Matching algorithm that extends the original idea of correlation-based template detection enhancing its efficiency and allowing to search for template occurrences regardless of its orientation. **Edge-based Matching** enhances this method even more by limiting the computation to the object edge-areas.

In this section we will describe the intrinsic details of both algorithms. In the next section (**Filter toolset**) we will explain how to use these techniques in Adaptive Vision Studio.

### Image Pyramid

*Image Pyramid* is a series of images, each image being a result of downsampling (scaling down, by the factor of two in this case) of the previous element.



Level 0 (input image)



Level 1

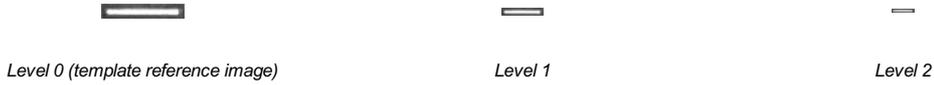


Level 2

### Pyramid Processing

Image pyramids can be applied to enhance the efficiency of the correlation-based template detection. The important observation is that the template depicted in the reference image usually is still discernible after significant downsampling of the image (though, naturally, fine details are lost in the process). Therefore we can identify match candidates in the downsampled (and therefore much faster to process) image on the highest level of our pyramid, and then repeat the search on the lower levels of the pyramid, each time considering only the template positions that scored high on the previous level.

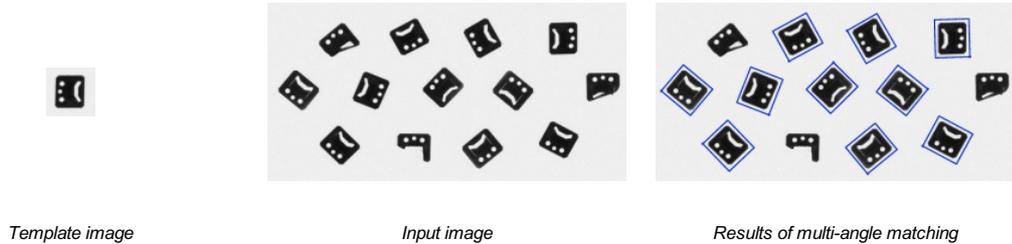
At each level of the pyramid we will need appropriately downsampled picture of the reference template, i.e. both input image pyramid and template image pyramid should be computed.



### Grayscale-based Matching

Although in some of the applications the orientation of the objects is uniform and fixed (as we have seen in the plug example), it is often the case that the objects that are to be detected appear rotated. In Template Matching algorithms the classic pyramid search is adapted to allow *multi-angle* matching, i.e. identification of rotated instances of the template.

This is achieved by computing not just one *template image* pyramid, but a set of pyramids - one for each possible rotation of the template. During the pyramid search on the input image the algorithm identifies the pairs (*template position, template orientation*) rather than sole template positions. Similarly to the original schema, on each level of the search the algorithm verifies only those (*position, orientation*) pairs that scored well on the previous level (i.e. seemed to match the template in the image of lower resolution).

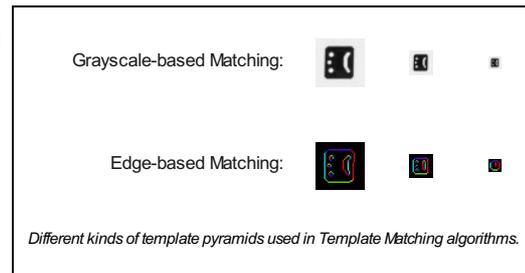


The technique of pyramid matching together with *multi-angle* search constitute the **Grayscale-based Template Matching** method.

### Edge-based Matching

Edge-based Matching enhances the previously discussed Grayscale-based Matching using one crucial observation - that the shape of any object is defined mainly by the shape of its edges. Therefore, instead of matching of the whole template, we could extract its edges and match only the nearby pixels, thus avoiding some unnecessary computations. In common applications the achieved speed-up is usually significant.

Matching object edges instead of an object as a whole requires slight modification of the original pyramid matching method: imagine we are matching an object of uniform color positioned over uniform background. All of object edge pixels would have the same intensity and the original algorithm would match the object anywhere wherever there is large enough blob of the appropriate color, and this is clearly not what we want to achieve. To resolve this problem, in Edge-based Matching it is the *gradient direction* (represented as a color in HSV space for the illustrative purposes) of the edge pixels, not their intensity, that is matched.



### Filter Toolset

Adaptive Vision Studio provides a set of filters implementing both **Grayscale-based Matching** and **Edge-based Matching**. For the list of the filters see **TemplateMatching** filters.

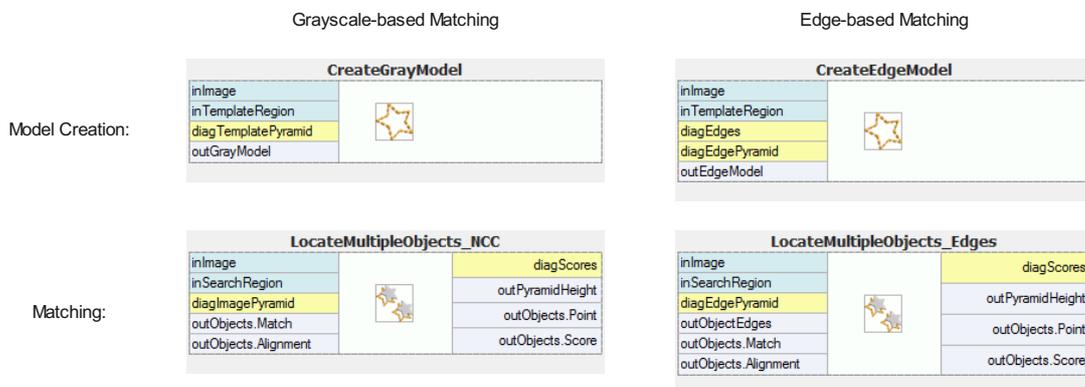
As the template image has to be preprocessed before the pyramid matching (we need to calculate the template image pyramids for all possible rotations), the algorithms are split into two parts:

- **Model Creation** - in this step the *template image* pyramids are calculated and the results are stored in a *model* - atomic object representing all the data needed to run the pyramid matching.
- **Matching** - in this step the *template model* is used to match the template in the *input image*.

Such an organization of the processing makes it possible to compute the *model* once and reuse it multiple times.

### Available Filters

For both Template Matching methods two filters are provided, one for each step of the algorithm.



Please note that the use of **CreateGrayModel** and **CreateEdgeModel** filters will only be necessary in more advanced applications. Otherwise it is enough to use a single filter of the **Matching** step and create the *model* by setting the *inGrayModel* or *inEdgeModel* parameter of the filter. For more information see [Creating Models for Template Matching](#).

The main challenge of applying the Template Matching technique lies in careful adjustment of filter parameters, rather than designing the program structure.

## Advanced Application Schema

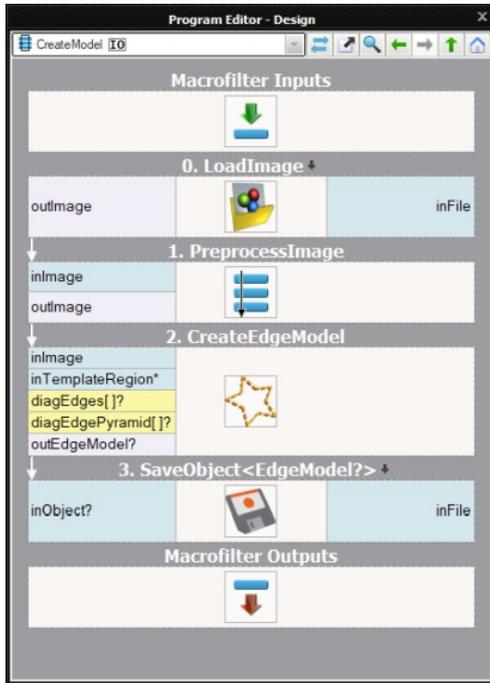
There are several kinds of advanced applications, for which the interactive GUI for Template Matching is not enough and the user needs to use the **CreateGrayModel** or **CreateEdgeModel** filter directly. For example:

1. When creating the model requires non-trivial image preprocessing.
2. When we need an entire array of models created automatically from a set of images.
3. When the end user should be able to define his own templates in the runtime application (e.g. by making a selection on an input image).

### Schema 1: Model Creation in a Separate Program

For the cases 1 and 2 it is advisable to implement model creation in a separate *Task* macrofilter, save the model to an AVDATA file and then link that file to the input of the matching filter in the main program:

Model Creation:



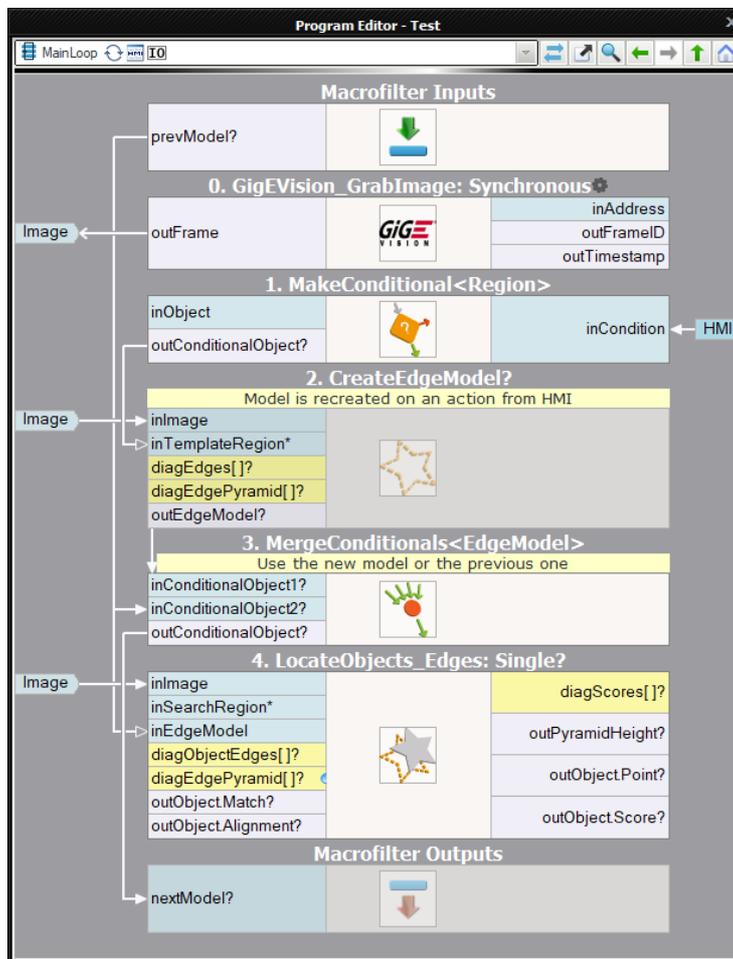
Main Program:



When this program is ready, you can run the "CreateModel" task as a program at any time you want to recreate the model. The link to the data file on the input of the matching filter does not need any modifications then, because this is just a link and what is being changed is only the file on disk.

### Schema 2: Dynamic Model Creation

For the case 3, when the model has to be created dynamically, both the model creating filter and the matching filter have to be in the same task. The former, however, should be executed conditionally, when a respective HMI event is raised (e.g. the user clicks an *ImpulseButton* or makes some mouse action in a *VideoBox*). For representing the model, a register of *EdgeModel?* type should be used, that will store the latest model (another option is to use **LastNotNil** filter). Here is an example realization with the model being created from a predefined box on an input image when a button is clicked in the HMI:

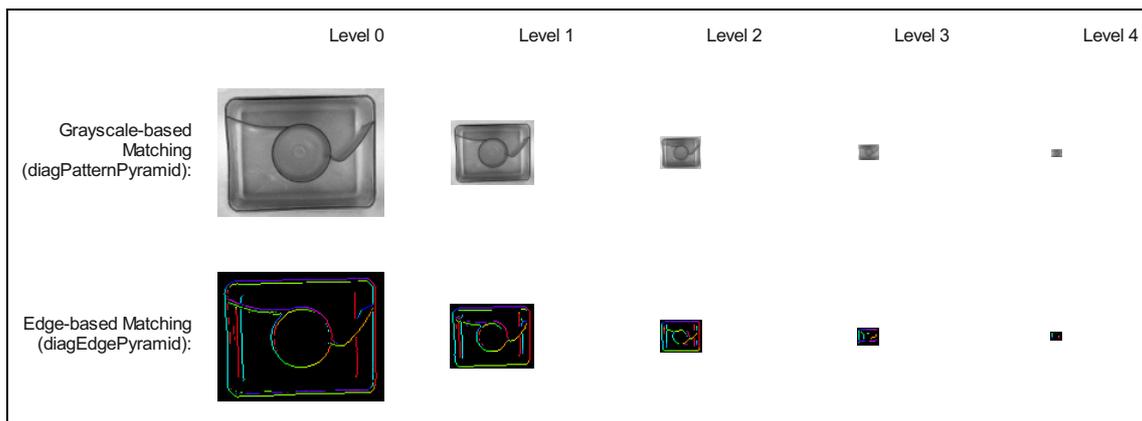


## Model Creation

### Height of the Pyramid

The **inPyramidHeight** parameter determines the number of levels of the pyramid matching and should be set to the largest number for which the template is still recognizable on the highest pyramid level. This value should be selected through interactive experimentation using the diagnostic output **diagPatternPyramid** (Grayscale-based Matching) or **diagEdgePyramid** (Edge-based Matching).

In the following example the **inPyramidHeight** value of 4 would be too high (for both methods), as the structure of the template is entirely lost on this level of the pyramid. Also the value of 3 seem a bit excessive (especially in case of Edge-based Matching) while the value of 2 would definitely be a safe choice.



### Angle Range

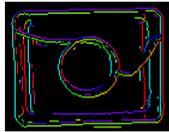
The **inMinAngle**, **inMaxAngle** parameters determine the range of template orientations that will be considered in the matching process. For instance (values in brackets represent the pairs of **inMinAngle**, **inMaxAngle** values):

- (0.0, 360.0): all rotations are considered (default value)
- (-15.0, 15.0): the template occurrences are allowed to deviate from the reference template orientation at most by 15.0 degrees (in each direction)
- (0.0, 0.0): the template occurrences are expected to preserve the reference template orientation

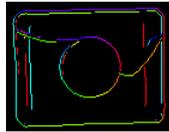
Wide range of possible orientations introduces significant amount of overhead (both in memory usage and computing time), so it is advisable to limit the range whenever possible.

### Edge Detection Settings (only Edge-based Matching)

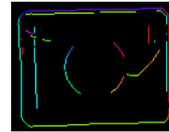
The **inEdgeMagnitudeThreshold**, **inEdgeHysteresis** parameters of **CreateEdgeModel** filter determine the settings of the hysteresis thresholding used to detect edges in the template image. The lower the **inEdgeMagnitudeThreshold** value, the more edges will be detected in the template image. These parameters should be set so that all the significant edges of the template are detected and the amount of redundant edges (noise) in the result is as limited as possible. Similarly to the pyramid height, edge detection thresholds should be selected through interactive experimentation using the diagnostic output **diagEdgePyramid** - this time we need to look only at the picture at the lowest level.



(15.0, 30.0) - excessive amount of noise



(40.0, 60.0) - OK



(60.0, 70.0) - significant edges lost

The **CreateEdgeModel** filter will not allow to create a model in which no edges were detected at the top of the pyramid (which means not only *some* significant edges were lost, but all of them), yielding an error in such case. Whenever that happens, the height of the pyramid, or the edge thresholds, or both, should be reduced.

### Matching

The **inMinScore** parameter determines how permissive the algorithm will be in verification of the match candidates - the higher the value the less results will be returned. This parameter should be set through interactive experimentation to a value low enough to assure that all correct matches will be returned, but not much lower, as too low value slows the algorithm down and may cause false matches to appear in the results.

### Tips and Best Practices

#### How to Select a Method?

For vast majority of applications the **Edge-based Matching** method will be both more robust and more efficient than **Grayscale-based Matching**. The latter should be considered only if the template being considered has smooth color transition areas that are not defined by discernible edges, but still should be matched.

# Using Local Coordinate Systems

## Introduction

Local coordinate systems provide a convenient means for inspecting objects that may appear at different positions on the input image. Instead of denoting coordinates of geometrical primitives in the absolute coordinate system of the image, local coordinate systems make it possible to use coordinates local to the object being inspected. In an initial step of the program the object is located and a local coordinate system is set accordingly. Other tools can then be configured to work within this coordinate system, and this makes them independent of the object translation, rotation and scale.

Two most important notions here are:

- **CoordinateSystem2D** – a structure consisting of *Origin* (Point2D), *Angle* (real number) and *Scale* (real number), defining a relative Cartesian coordinate system with its point (0, 0) located at the *Origin* point of the parent coordinate system (usually an image).
- **Alignment** – the process of transforming geometrical primitives from a local coordinate system to the coordinates of an image (absolute), or data defining such transformation. An alignment is usually represented with the **CoordinateSystem2D** data type.

## Creating a Local Coordinate System

There are two standard ways of setting a local coordinate system:

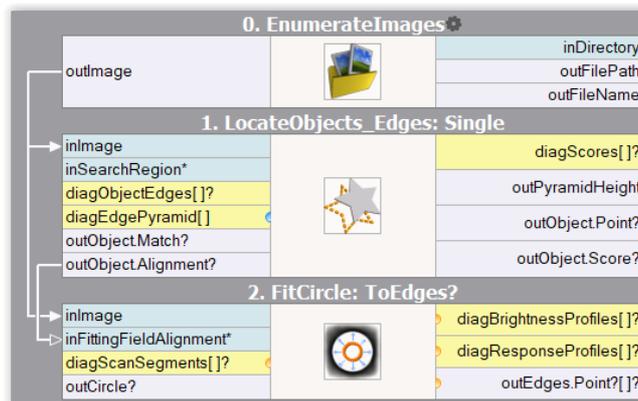
1. With **TemplateMatching** filters it is straightforward as the filters have **outObjectAlignment(s)** outputs, which provide local coordinate systems of the detected objects.
2. With one of the **CreateCoordinateSystem** filters, which allow for creating local coordinate systems manually at any location, and with any rotation and scale. In most typical scenarios of this kind, the objects are located with **1D Edge Detection**, **Shape Fitting** or **Blob Analysis** tools.

## Using a Local Coordinate System

After a local coordinate system is created it can be used in the subsequent image analysis tools. The high level tools available in Adaptive Vision Studio have an **inAlignment** (or similar) input, which just needs to be connected to the port of the created local coordinate system. At this point, you should first run the program at least to the point where the coordinate system is computed, and then the geometrical primitives you will be defining on other inputs, will be automatically aligned with the position of the inspected object.

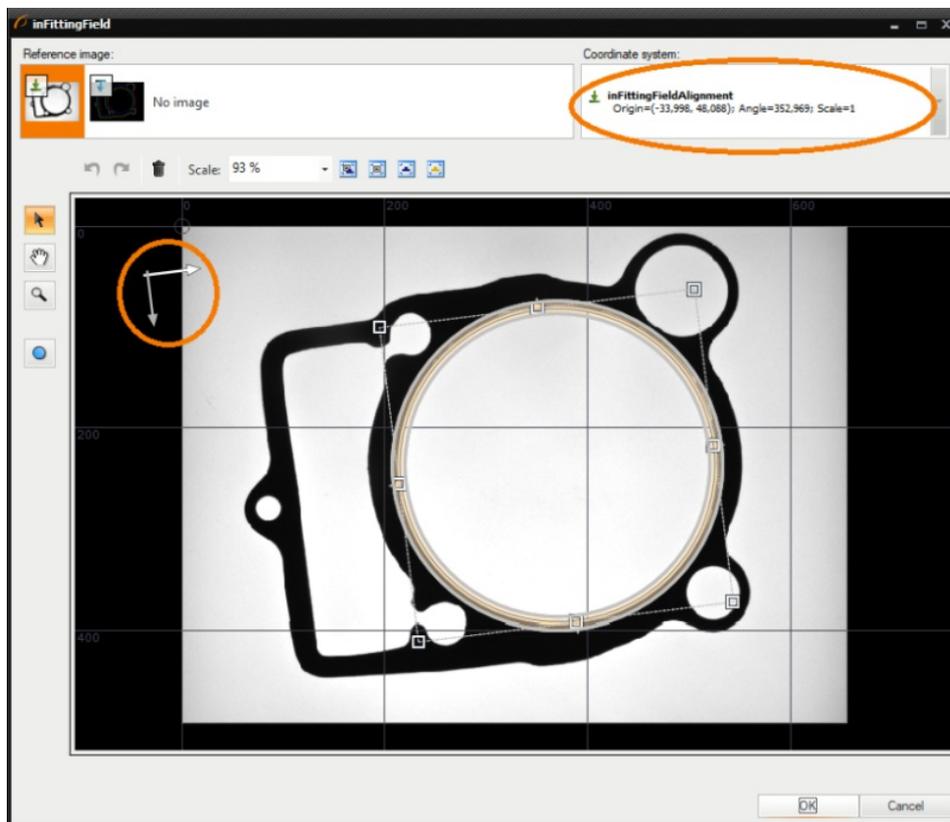
### Example 1: Alignment from Template Matching

To use object alignment from a **TemplateMatching** filter, you need to connect the **Alignment** ports:



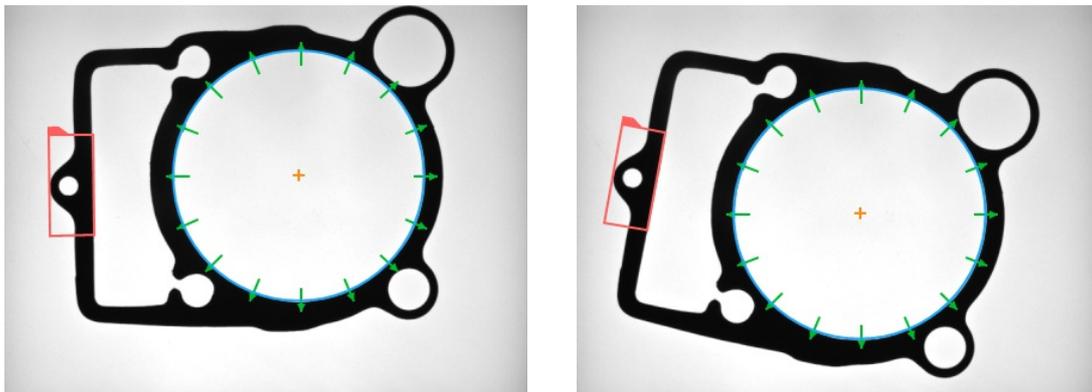
Template Matching and an aligned circle fitting.

When you execute the template matching filter and enter the editor of the **inFittingField** input of the **FitCircleToEdges** filter, you will have the local coordinate system already selected (you can also select it manually) and the primitive you create will have relative coordinates:



*Editing an expected circle in a local coordinate system.*

During program execution this geometrical primitive will be automatically aligned with the object position. Moreover, you will be able to adjust the input primitive in the context of any input image, because they will be always displayed aligned. Here are example results:



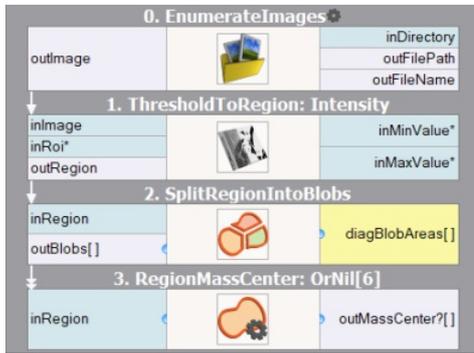
### Example 2: Alignment from Blob Analysis

In many applications objects can be located with methods simpler and faster than Template Matching – like [1D Edge Detection](#), [Shape Fitting](#) or [Blob Analysis](#). In the following example we will show how to create a local coordinate system from two blobs:

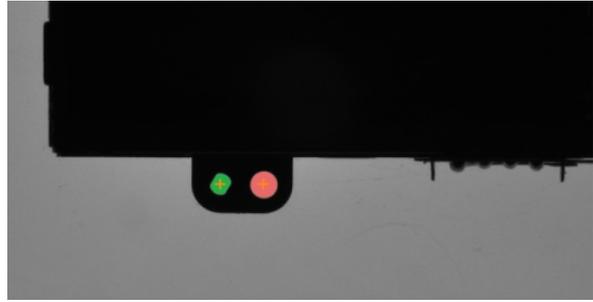


*Two holes clearly define the object location.*

In the first step we detect the blobs (see also: [Blob Analysis](#)) and their centers:

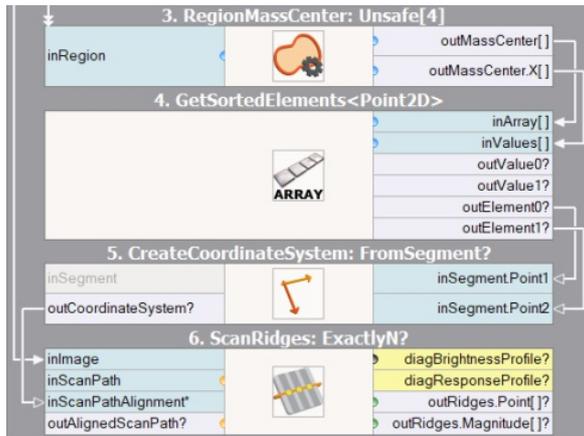


Filters detecting blobs and their centers.

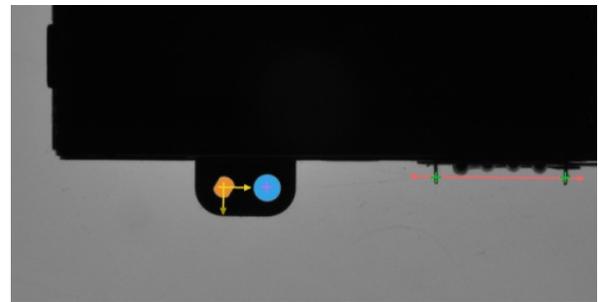


The result of blob detection.

In the second step we sort the centers by the X coordinate and create a coordinate system "from segment" defined by the two points (**CreateCoordinateSystemFromSegment**). The segment defines both the origin and the orientation. Having this coordinate system ready, we connect it to the **inScanPathAlignment** input of **ScanExactlyNRidges**, which will measure the distance between two insets. The measurement will work correctly irrespective of the object position (mind the expanded structure inputs and outputs):



Filters creating a coordinate systems and performing an aligned measurement.



Created local coordinate system and an aligned measurement.

## Manual Alignment

In some cases the filter you will need to use with a local coordinate system will have no appropriate **inAlignment** input. In such cases the solution is to transform the primitive manually with filters like **AlignPoint**, **AlignCircle**, **AlignRectangle**. These filters accept a geometrical primitive defined in a local coordinate system, and the coordinate system itself, and return the same primitive, but with absolute coordinates, i.e. aligned to the coordinate system of an image.

A very common case is with ports of type **Region**, which is pixel-precise and, while allowing for creation of arbitrary shapes, cannot be directly transformed. In such cases it is advisable to use the **CreateRectangleRegion** filter and define the region-of-interest at **inRectangle**. The filter, having also the **inRectangleAlignment** input connected, will return a region properly aligned with the related object position. Some ready-made tools, e.g. **CheckPresence\_Intensity**, use this approach internally.

## Not Mixing Local Coordinate Systems

It is important to keep in mind that geometrical primitives that appear in different places of a program may belong to different coordinate systems. When such different objects are combined together (e.g. with a filter like **SegmentSegmentIntersection**) or placed on a single data preview, the results will be meaningless or at least confusing. Thus, only objects belonging to the same coordinate system should be combined. In particular, when placing primitives on a preview on top of an image, only aligned primitives (with absolute coordinates) should be used.

As a general rule, image analysis filters of Adaptive Vision Studio accept primitives in local coordinate systems on inputs, but outputs are always aligned (i.e. in the absolute coordinate system). In particular, many filters that align input primitives internally also have outputs that contain the input primitive transformed to the absolute coordinate system. For example, the **ScanSingleEdge** filter has a **inScanPath** input defined in a local coordinate system and a corresponding **outAlignedScanPath** output defined in the absolute coordinate system:



The **ScanSingleEdge** filter with a pair of ports: **inScanPath** and **outAlignedScanPath**, belonging to different coordinate systems.

# Optical Character Recognition

## Introduction

Optical Character Recognition (OCR) is a machine vision task consisting in extracting textual information from images.

State of the art techniques for OCR offer high accuracy of text recognition and invulnerability to medium grain graphical noises. They are also applicable for recognition of characters made using dot matrix printers. This technology gives satisfactory results for partially occluded or deformed characters.

Efficiency of the recognition process mostly depends on the quality of text segmentation results. Most of the recognition cases can be done using a provided set of recognition models. In other cases a new recognition model can be easily prepared.



Result of data extraction using OCR.

## Concept

OCR technology is widely used for automatic data reading from various sources. It is especially used to gather data from documents and printed labels.

In the first part of this manual usage of high level filters will be described.

The second part of this manual shows how to use standard OCR models provided with Adaptive Vision Studio. It also shows how to prepare an image to get best possible results of recognition.

The third part describes the process of preparing and training OCR models.

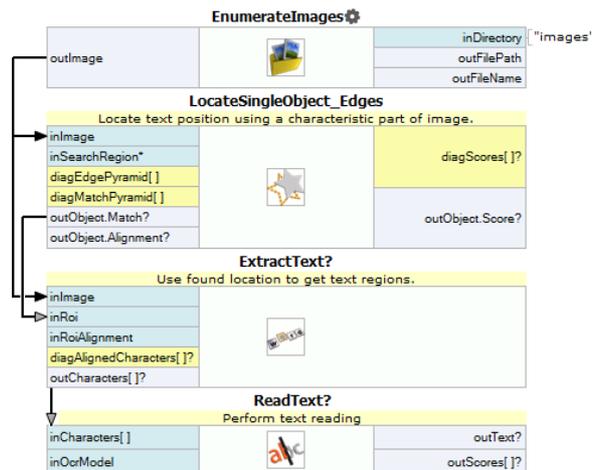
The last part presents an example program that reads text from images.

## Using high level Optical Character Recognition filters

Adaptive Vision Studio offers a convenient way to extract a text region from an image and then read it using a trained OCR classifier.

The typical OCR application consists of the following steps:

1. **Find text position** – locate the text position using template matching,
2. **Extract text** – use the filter **ExtractText** to distinct the text form the background and perform its segmentation,
3. **Read text** – recognizing the extracted characters with the **ReadText** filter.



Example OCR application using high level filters.

Adaptive Vision Studio provides convenient graphical tools for configuring the OCR process. More informations can be found in the following articles:

1. [Creating Text Segmentation Models](#)
2. [Creating Text Recognition Models](#)

## Details on Optical Character Recognition technique

### Reading text from images

In order to achieve the most accurate recognition it is necessary to perform careful text extraction and segmentation. The overall process of acquiring text from images consists of the following steps:

1. [Getting text location](#),
2. [Extracting text from the background](#),
3. [Segmenting text](#),
4. [Using prepared OCR models](#),
5. [Character recognition](#),
6. [Interpreting results](#),
7. [Verifying results](#).

The following sections will introduce methods used to detect and recognize text from images. For better understanding of this guide the reader should be familiar with [basic blob analysis techniques](#).

### Getting text location

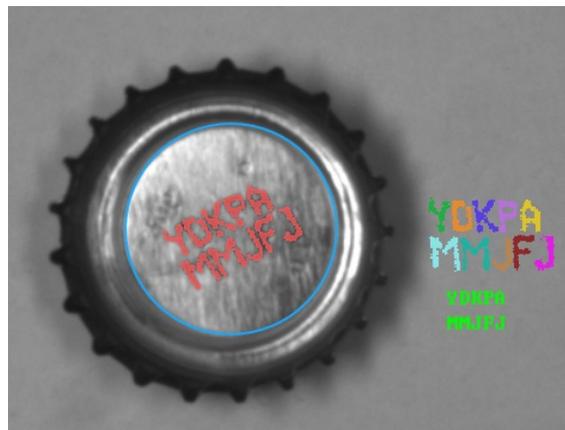
In general, text localization tasks can be divided into three cases:

1. The location of text is fixed and it is described by boxes called masks. For example, the personal identification card is produced according to the formal specification. The location of each data field is known. A well calibrated vision system can take images in which the location of the text is almost constant.



*An example image with text masks.*

2. Text location is not fixed, but it is related to a characteristic element on the input images or to a special marker (an optical mark). To get the location of the text the optical mark has to be found. This can be done with template matching, 1D edge detection or other technique.
3. The location of text is not specified, but characters can be easily separated from the background with image thresholding. The correct characters can then be found with blob analysis techniques.

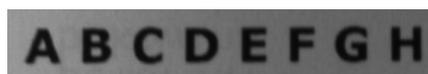


*Getting text from a bottle cap.*

When the text location is specified, the image under analysis must be transformed to make text lines parallel to the X-axis. This can be done with **RotateImage**, **CropImageToRectangle** or **ImageAlongPath** filters.

#### Extracting text from the background

A major complication during the process of text extraction may be uneven light. Some techniques like light normalization or edge sharpening can help in finding characters. The example of light normalization can be found in the example project *Examples\Tablets*. The presentation of image sharpening using the Fourier transform can be found in the *Examples\Fourier* example.



*Original image.*



*Image after light normalization.*



*Image after low-frequency image damping using the Fourier transform.*

Text extraction is based on image binarization techniques. To extract characters, filters like **ThresholdToRegion** and **ThresholdToRegion\_Dynamic** can be used. In order to avoid recognizing regions which do not include characters, it is advisable to use filters based on blob area.



*Sample images with uneven light.*



Results of *ThresholdToRegion* and *ThresholdToRegion\_Dynamic* on images with uneven light.

At this point the extracted text region is prepared for segmentation.

### Segmenting text

Text region segmentation is a process of splitting a region into lines and individual characters. The recognition step is only possible if each region contains a single character.

Firstly, if there are multiple lines of text, separation into lines must be performed. If the text orientation is horizontal, simple region dilation can be used followed by splitting the region into blobs. In other cases the text must be transformed, so that the lines become horizontal.

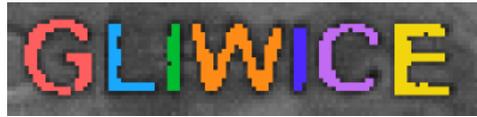


The process of splitting text into lines using region morphology filters.

When text lines are separated, each line must be split into individual characters. In a case when characters are not made of diacritic marks and characters can be separated well, the filter **SplitRegionIntoBlobs** can be used. In other cases the filter **SplitRegionIntoExactlyNCharacters** or **SplitRegionIntoMultipleCharacters** must be used.



Character segmentation using *SplitRegionIntoBlobs*.



Character segmentation using *SplitRegionIntoMultipleCharacters*.

Next, the extracted characters will be translated from graphical representation to textual representation.

### Using prepared OCR models

Standard OCR models are typically located in the disk directory *C:\ProgramData\Adaptive Vision\{Adaptive Vision Product Name}\PretrainedFonts*.

The table below shows the list of available font models:

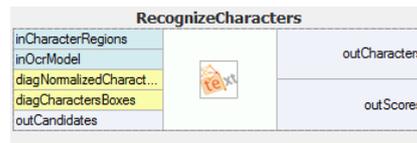
Font name	Font typeface	Set name	Characters
OCRA	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ-./
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+
OCRB	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ-./
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+
MCR	monospaced	ABC09	ABC0123456789
Computer	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ-./
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+
DotMatrix	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ+.-/
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ+.-0123456789./
		09	0123456789.+.-/
Regular	proportional	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ-./
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+

### Character recognition

Adaptive Vision Library offers two types of character classifiers:

1. Classifier based on multi-layer perceptron (MLP).
2. Classifier based on support vector machines (SVM).

Both of the classifiers are stored in the **OcrModel** type. To get a text from character regions use the **RecognizeCharacters** filter, shown on the image below:



The first and the most important step is to choose the appropriate character normalization size. The internal classifier recognizes characters using their normalized form. More information about character normalization process will be provided in the section describing the process of classifier training.

The character normalization allows to classify characters with different sizes. The parameter **inCharacterSize** defines the size of a character before the normalization. When the value is not provided, the size is calculated automatically using the character bounding box.

Character presentation

Characters after normalization

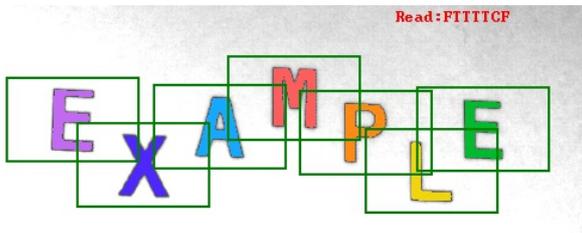
Description



The appropriate character size is chosen.



The size of character is too small.



Too much information about a character is lost because of too large size has been selected .

Next, character sorting order must be chosen. The default order is from left to right.

If the input text contains spaced characters, the value of `inMinSpaceWidth` input must be set. This value indicates the minimal distance between two characters between which a space will be inserted.

Character recognition provides the following information:

1. the read text as a string (`outCharacters`),
2. an array of character recognition scores (`outScores`),
3. an array of recognition candidates for each character (`outCandidates`).

### Interpreting results

The table below shows recognition results for characters extracted from the example image. An unrecognized character is colored in red.

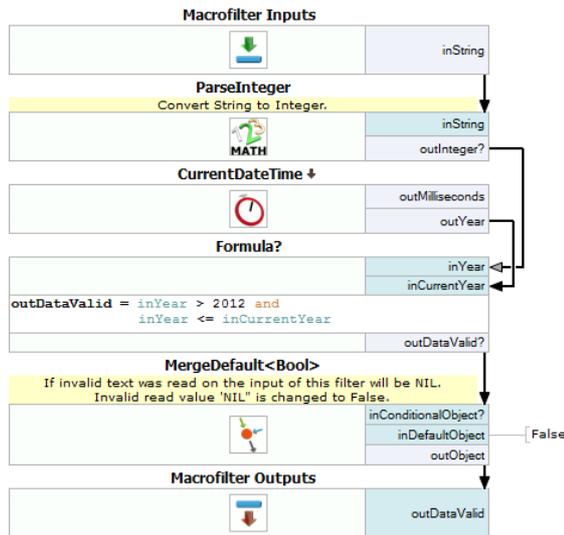
Original character	Recognized character	Score	Candidates (character and accuracy)	
	( <code>outCharacters</code> )	( <code>outScores</code> )	(outCandidates)	
E	E	1.00	E: 1.00	
X	X	1.00	X: 1.00	
A	A	1.00	A: 1.00	
M	M	1.00	M: 1.00	
P	R	0.50	R: 0.90	B: 0.40
L	L	1.00	L: 1.00	
E	E	1.00	E: 1.00	

In this example the letter *P* was not included in the training set. In effect, the OCR model was unable to recognize the representation of the *P* letter. The internal classifier was trying to select most similar known character.

### Verifying results

The results of the character recognition process can be validated using basic string manipulation filters.

The example below shows how to check if a read text contains a valid year value. The year value should be greater than 2012 (e.g. production started in this year) and must not be greater than the current year.



The short year value validation.

This example consists of two parts. In the first part an input string is converted to an **Integer** value. The second one checks if year number is valid.

The filter **ParseInteger** was used to convert a text into a number of the **Integer** type. **ParseInteger** can return Nil value so **MergeDefault** filter should be used to decide what is a validation result when invalid year number was provided.

Next, the year number is checked using a formula. To get the current year the filter **CurrentDateTime** is used. If all conditions are met, the validation passes.

For more complex validation a user defined filter is recommended. For further information read the [documentation on creating user filters](#).

## Preparation of the OCR models

An OCR model consists of an internal statistical tool called a classifier and a set of character data. There are two kinds of classifiers used to recognize characters. The first classifier type is based on the multilayer perceptron classifier (MLP) and the second one uses support vector machines (SVM). For further details please refer to the documentation of the **MLP\_Init** and the **SVM\_Init** filters. Each model must be trained before it can be used.

The process of OCR model training consists of the following steps:

1. preparation of the training data set,
2. selection of the normalization size and character features,
3. setup of the OCR model,
4. training of the OCR model,
5. saving results to a file.

When these steps are performed, the model is ready to use.

### Preparation of the training data set

Each classifier needs character samples in order to begin the training process. To get the best recognition accuracy, the training character samples should be as similar as possible to those which will be provided for recognition. There are two possible ways to obtain sample characters: (1) extraction of characters from real images or (2) generation of artificial characters using computer fonts.



Synthetic characters generated by means of a computer font.

In the perfect world the model should be trained using numerous real samples. However, sometimes it can be difficult to gather enough real character samples. In this case character samples should be generated by deforming the available samples. A classifier which was trained on a not big enough data set can focus only on familiar character samples at the same time failing to recognize slightly modified characters.



Character samples acquired from a real usage.

Example operations which are used to create new character samples:

1. region rotation (using the **RotateRegion** filter),
2. shearing (**ShearRegion**),
3. dilatation and erosion (**DilateRegion**, **ErodeRegion**),
4. addition of a noise.



The set of character samples deformed by: the region rotation, morphological transforms, shearing and noises.

Note: Adding too many deformed characters to a training set will increase the training time of a model.

Note: Excessive deformation of character shape can result in classifier inability to recognize the learnt character base. For example: if the training set contains a C character with too many noises, it can be mistaken for O character. In this case the classifier will be unable to determine the base of a newly provided character.

Each character sample must be stored in a structure of type **CharacterSample**. This structure consists of a character region and its textual representation. To create an array of character samples use the **MakeCharacterSamples** filter.

### Selection of normalization size and character features

The character normalization allows for reduction of the amount of data used in the character classification. The other aim of normalization is to enable the classification process to recognize characters of various sizes.

During normalization each character is resized into a size which was provided during the model initialization. All further classifier operations will be performed on the resized (normalized) characters.



Various size characters before and after the normalization process.

Selection of too large normalization size will increase training time of the OCR classifier. On the other hand, too low size will result in loss of important character details. The selected normalization size should be a compromise between classification time and the accuracy of recognition. For the best results, a character size after normalization should be similar to its size before normalization.

During the normalization process some character details will be lost, e.g. the aspect ratio of a character. In the training process, some additional information can be added, which can compensate for the information loss in the normalization process. For further information please refer to the documentation of the **TrainOcr\_MLP** filter.

### Training of the OCR model

There are two filters used to train each type of an OCR classifier. These filters require parameters which describe the classifier training process.



Training of MLP classifier using **TrainOcr\_MLP**.



Training of SVM classifier using **TrainOcr\_SVM**.

### Saving the training results

After successful classifier training the results should be saved for future use. The filter **SaveObject** should be used.

### Example: Reading numbers from masks

The purpose of this example is to read fields from ID cards. The input data is extracted from real cases. The example consists of two parts:

1. Creation of data set for classifier training is presented in "TrainCharacters" macrofilter.
2. This macrofilter is designed to acquire data from provided masks. It is implemented in the "Main" macrofilter.

The example project is available in *Examples\Ocr Read Numbers* directory.

### Training the classifier

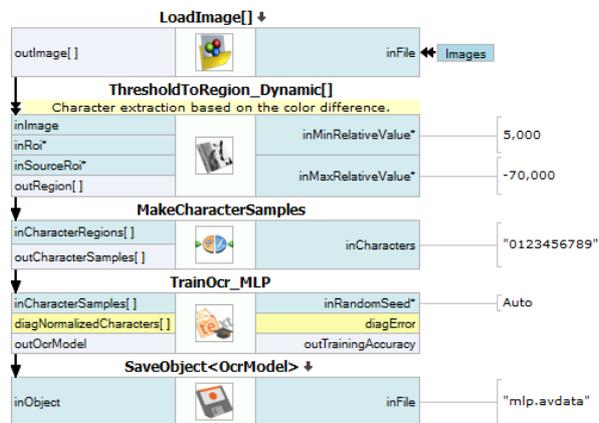
First, this filter **LoadImage** loads the character images from separate files. File names are stored in a **global parameter** called *FileNames*.

Next, each image is thresholded and the character samples are made.

Then an MLP classifier is created.

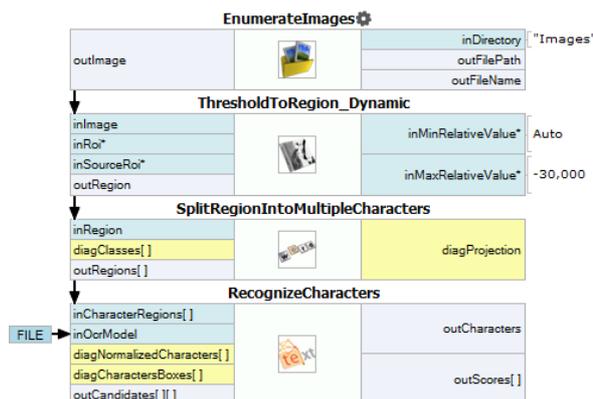
After initialization the classifier is trained using available character samples.

Finally, the created classifier is saved to a file.



Macrofilter definition which performs OCR training.

### Reading data from images



Macrofilter which uses the trained classifier to acquire text from images.

First, the example image is loaded from a file.

Next, the text is extracted from the background by **ThresholdToRegion\_Dynamic**.

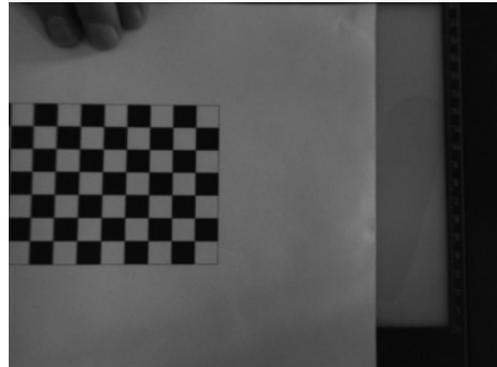
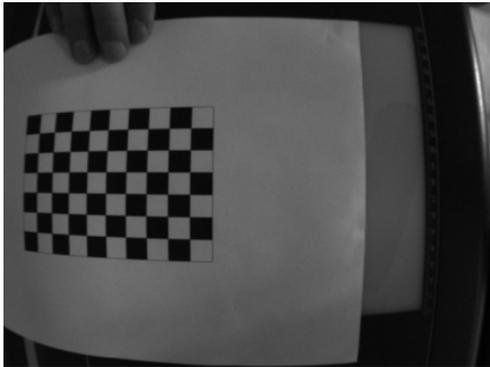
Then, the text is split into separated characters with **SplitRegionIntoMultipleCharacters**.

Finally, the characters are recognized.

# Camera Calibration

## Introduction

Camera calibration is a process of creating an image deformation model, which can be used to recover an undeformed image. Adaptive Vision Studio makes it possible to model lens distortions as well as perspective, cylindrical and spherical transformations. It is often required both in 2D and 3D applications. In the case of 2D vision it is necessary for performing reliable measurements.



*Removing lens distortion using camera calibration filters. Images before and after remapping.*

## Type SpatialMap

Computation of the deformation model is a very complex process and can take significant time. To assure the highest performance this process is performed once and the model is stored in the **SpatialMap** type.

The type **SpatialMap** is a map of image locations and their corresponding positions after removal of distortion. The process of transforming an input image to an image without distortion is called **remapping**. It is very fast, because it only translates pixels by precomputed vectors.

## Deformation Types

The purpose of remapping of an image is to recover the image from before deformation. Some deformations result from physical imperfection of the used optics, other are caused by the physical shape of the object under inspection. Adaptive Vision Studio provides tools for dealing with several different deformation types.

### Lens Distortion

This unwanted distortions are introduced by imperfections of camera lenses. It is common for low-cost wide-angle lenses to introduce radial (barrel) distortion. The effect is that straight lines in reality are not reproduced as such in the recorded image. The central area in the image usually has little distortion, and the effect becomes stronger toward the borders of the image.

Note: The lens imperfections also introduce the *vignette* effect which results in darkening of the image pixels with the distance to the center of the lens. This kind of deformations is not covered in this document, but it can be fixed using basic image processing filters.

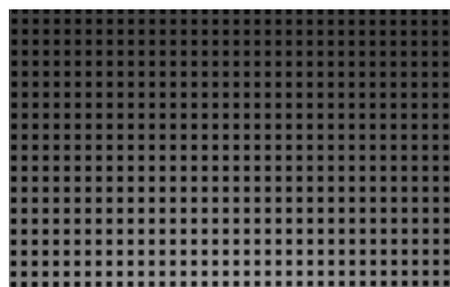
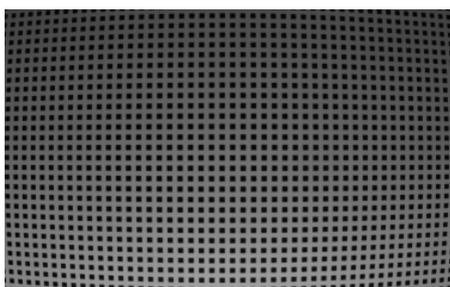
Adaptive Vision Studio offers a convenient way to create lens undistortion map using a [GUI for chessboard-based calibration](#).

Examples which contains more detailed step-by-step computation of a undistortion map (without use of the graphical tool):

- [Camera calibration example using chessboards with Calibration GUI.](#)
- [Camera calibration example using chessboards \(Advanced\).](#)
- [Camera calibration example using a custom grid.](#)

The manual process of computing a lens distortion model consists of the following steps:

1. **Gathering images** – the most important step is to gather images that will be used to determine lens intrinsic parameters. Two most common methods can be used:
  1. **Calibration using chessboards** – this method requires to use a special calibration grid that can be bought or printed. Quality of the calibration depends on the calibration grid. The filter **DetectChessboardGrid** is prepared to extract points from such grids.
  2. **Calibration using custom grids** – to calibrate camera the calibration grid is needed. Filter **AnnotateGridPoints** computes a difference in the position of the grid points between real images and the computed calibration grid.
2. **Finding camera parameters** – calculating camera distortion coefficients (camera matrix) using the found 3D object points and their projections in the 2D space. To find a camera matrix the **CalibrateCamera** filter can be used.
3. **Preparing a spatial map** – the **CreateUndistortionMap** filter uses the camera matrix to compute a spatial map. The result can be saved to a file for later use.
4. **Remapping an image** – use filter the **UndistortImage** filter to remove distortions from an input image.

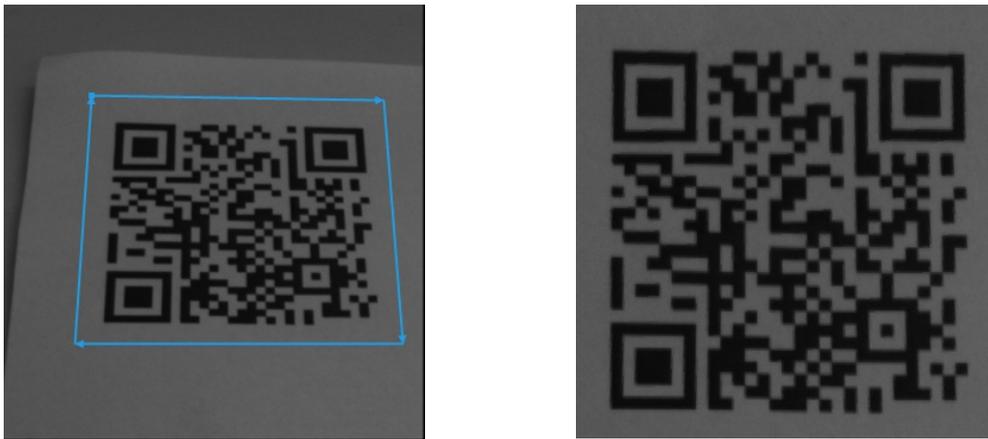


*Remapping an image with a custom dot-based calibration grid.*

## Perspective Removal

In some cases the camera is located at a position in which the objects can be deformed due to perspective transform.

To remove perspective distortion the filter **CreatePerspectiveMap\_Path** can be used. This filter requires a path made of four points. The first point should indicate the top-left corner of the object. Next corner points should be marked in clockwise order: top-right, bottom-right and bottom-left.



The left image shows a tilted QR code. The perspective is removed in the right image.

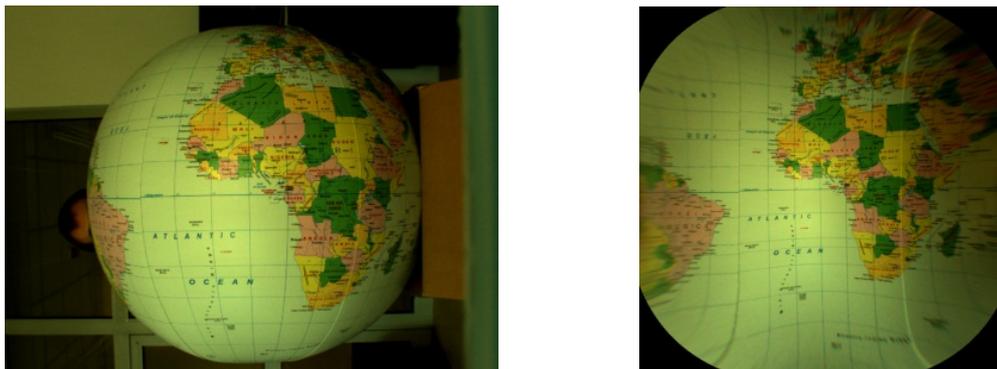
The process of removing perspective is presented in [World Calibration Example](#).

### Geometric Deformation

This kind of distortions are introduced by the shape of the physical object on the image. Calculation of a distortion model for these object is necessary to perform image analysis.

There are several filters for creating spatial maps for an object with a geometric distortion:

1. **Cylinder (CreateCylinderMap)** – e.g. flattening of a bottle label.
2. **Sphere (CreateSphereMap)** – e.g. reading a label from light bulb.
3. **Circular objects (polar transform) (CreateImagePolarTransformMap)** - e.g. reading a label wrapped around a DVD disk center.

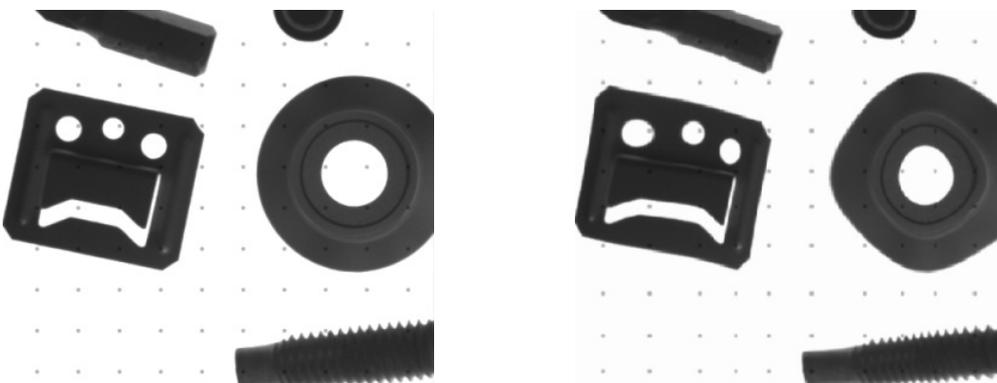


Example of remapping of a spherical object using **CreateSphereMap** and **RemapImage**. Image before and after remapping.

### Other Transforms (Warping)

Image remapping can also be used to perform other geometric transforms, especially affine image transforms. If the transform parameters are fixed (e.g. the rotation angle), then this approach results in higher performance than can be achieved with regular filters (e.g. **RotateImage**). Examples of such operations are: **CreateImageRotationMap**, **CreateImageResizeMap** or **CreateMatrixTransformMap**.

Furthermore custom spatial maps can be created with **ConvertMatrixMapsToSpatialMap**.

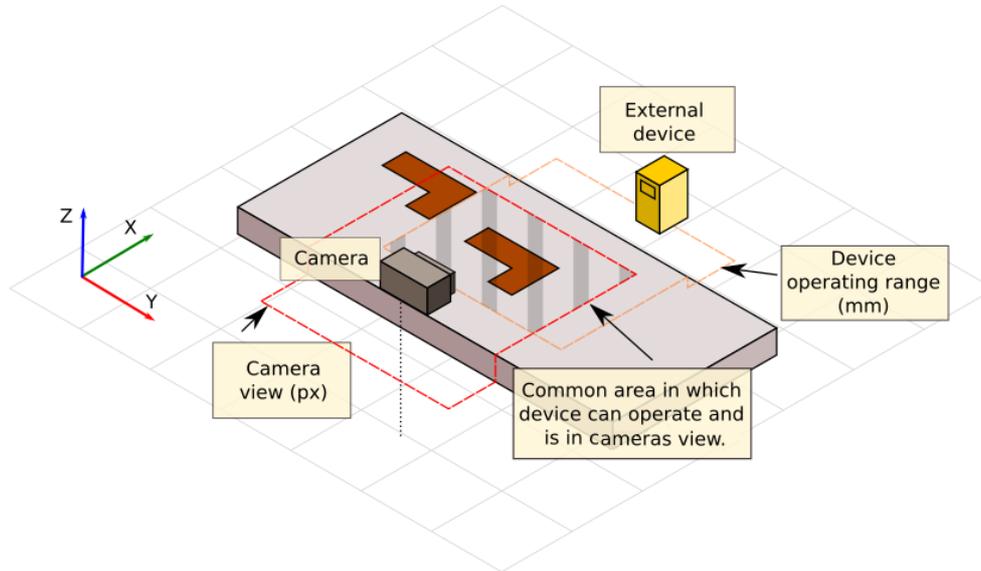


An example of custom image transform created with **ConvertMatrixMapsToSpatialMap**. Image before and after remapping.

# Calibration of World Coordinates

## Introduction

Adaptive Vision Studio provides tools to perform inspections on 2D images. Image analysis results are then passed to external devices like industrial robots or separators. In most cases, however, these devices work in different coordinate systems than the coordinate system of the image and use different measurement units (e.g. millimeters instead of pixels). For these reasons an appropriate calibration is required.

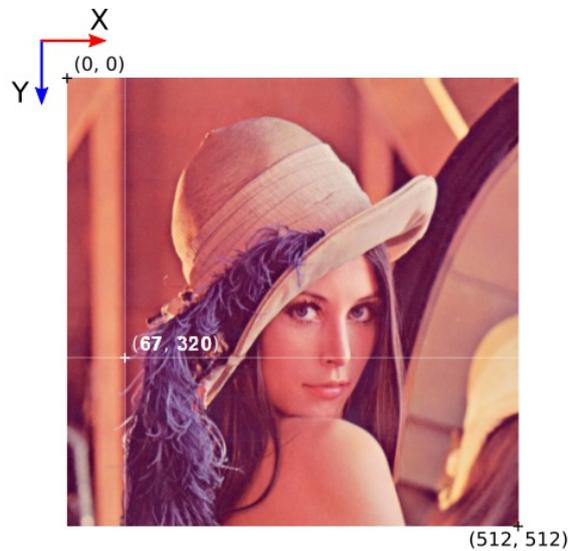


To send correct locations to an external device a calibration must be performed using points in common area marked on the image.

## Image and World Coordinates

Image coordinates are denoted in pixels, with the origin point (0, 0) corresponding always to the top-left corner of the image. On the other hand world coordinates are denoted in real-world units, like millimeters, with the origin point placed at arbitrary location. Moreover, world coordinates can be three-dimensional and the plane of interest can be visible with perspective distortion. The task is to find the correspondence between the image and the world coordinates, and translate between ones and another.

The image below shows the basic coordinate values in image coordinates. The X axis starts at the left edge of an image and has maximal value at the right edge. The Y axis starts at the top of the image and ends at the bottom edge. This view is always shown in pixel units.



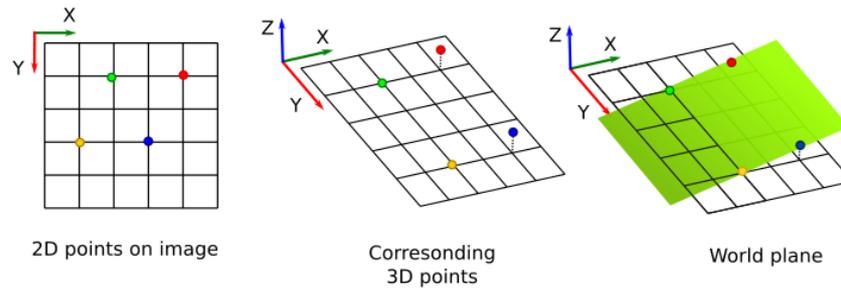
Directions and pixel positions in image coordinates.

## Using Calibration

Image analysis tools of Adaptive Vision Studio operate in the image coordinate system. Before the results are passed to an external devices, they usually have to be translated to world coordinates. A typical example is when an object is found on an image and then its location should be passed to an industrial robot to pick it up.

Adaptive Vision Studio offers two ways of world coordinate calibration:

- **Point-To-Point Calibration** in which the model is created from known image points (2D) and the corresponding world points (3D). The filter **CalibrateWorldCoordinates** computes the world plane which can be later used to compute world points. Also world calibration resolution (inverse of scale) is found, which is necessary to calculate world points.



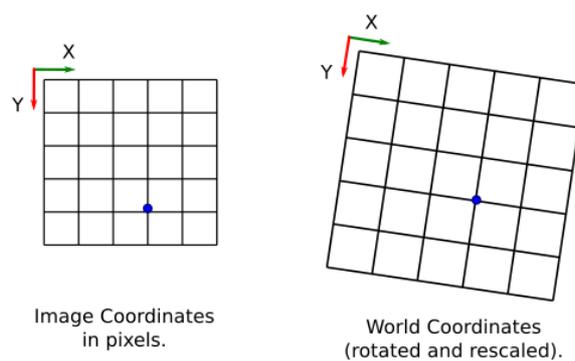
*Finding a world plane using N pairs of points (at least four).*

Image's points can be marked using an image as a background, but world points must be entered into the Properties Control manually.

	X	Y	Z	X	Y
0	150,000	110,000	0,000	465,690	373,149
1	150,000	0,000	0,000	463,981	121,944
2	0,000	0,000	0,000	113,056	118,241
3	0,000	110,000	0,000	117,685	376,574

*Configured pairs of image points (2D) and world points (3D).*

- **Calibration with explicit parameters** in which only 2D translations, rotations and scaling is possible. This calibration can be done with the **CoordinateSystem2DtoPosition3D** filter.



*Creating a world calibration plane from the **CoordinateSystem2D** type.*

Coordinate calibration model is represented by the **Position3D** type, which denotes the position of the plane-of-interest in the camera field of view. Calibration can be done once before the main program loop or in a separate program which saves it to a file for later use. Also resolution must be saved for the further usage.

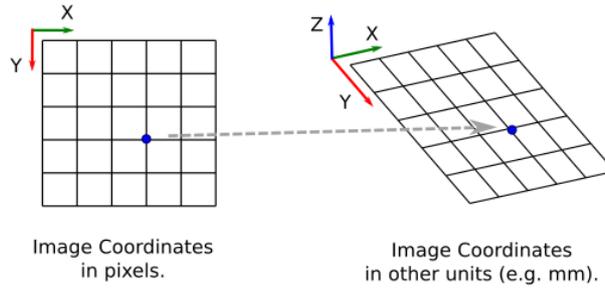
**Remarks:**

- Coordinate calibration can take some time when many point pairs are used.
- If the calibration points are not consistent, then the world plane is approximated. Please always check if the calibration error lies in an expected range.
- In many cases the calibration resolution (inverse of scale) is different from 1.0 so it is necessary to save this parameter too.

**Getting World Points**

When the world plane is known basic geometric primitives can be converted to world coordinates. Adaptive Vision Studio offers the following conversion filters:

- **PointToWorldCoordinates** – converts a single **Point2D** to a single **Point3D** value,
- **SegmentToWorldCoordinates** – converts a **Segment2D** to an array of two 3D points (**Point3DArray**),
- **PathToWorldCoordinates** – converts a **Path** to an array of 3D points (**Point3DArray**).



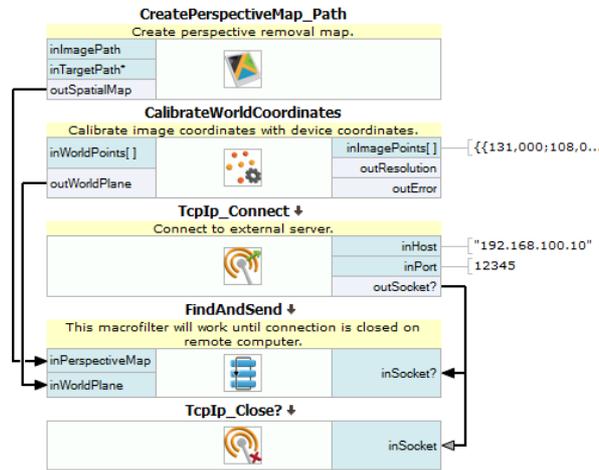
Converting a point from image to world coordinates using the **PointToWorldCoordinates** filter.

There is also the opposite filter, **PointToImageCoordinates**, which converts a 3D point to a point on an image.

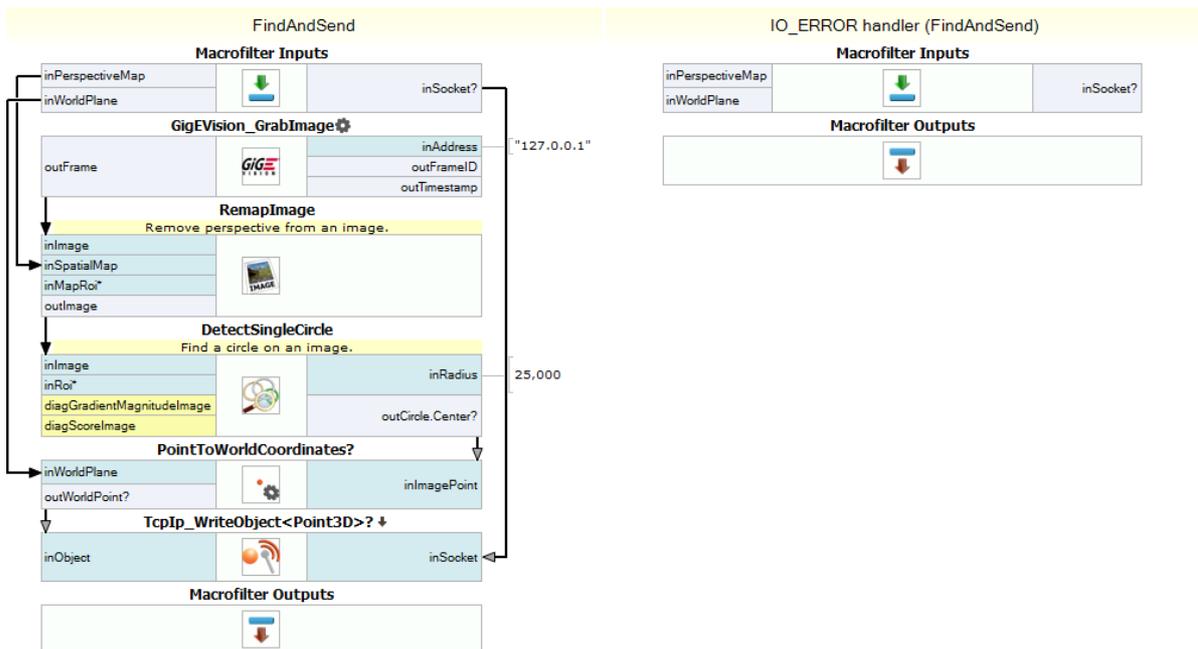
### Example

The program below finds a circle on an image, then converts image to world coordinates and sends the results to an external device using TCP/IP filters.

**Macrofilter Main** – initialization of a TCP/IP connection, creation of a perspective map, calibration of world coordinates.



**Macrofilter FindAndSend** – macrofilter finds a circle on an image and then converts its coordinates to world coordinates. At the end the results are sent to the external device.



## Golden Template

Golden Template technique performs a pixel-to-pixel comparison of two images. This technique is especially useful when the object's surface or object's shape is very complex.

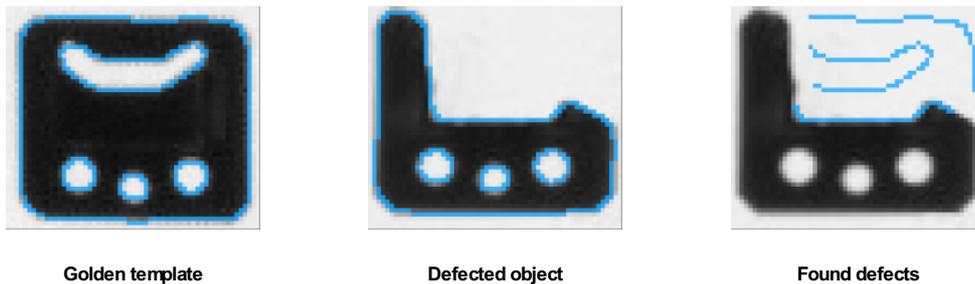
Adaptive Vision Studio offers two way of performing the golden template comparison.

- **Comparison based on pixels intensity** - it can be achieved using the **CompareGoldenTemplate\_Intensity**. In this method two images are compared pixel-by-pixel and the defect is classified based on a difference between pixels intensity. This technique is especially useful in finding defects like smudges, scratches etc.



*Example usage of Golden Template technique using the pixels intensity based comparison.*

- **Comparison based on objects edges** - this method is very useful when defects may occur on the edge of the object and pixel comparison may fail due to different light reflections or the checking the object surface is not necessary. For matching object's edges use the **CompareGoldenTemplate\_Edges** filter.



*Example usage of Golden Template technique using the edges comparison.*

## How To Use

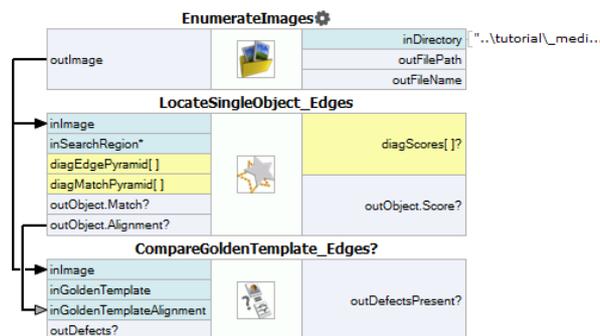
Golden template is a previously prepared image which is used to compare image from the camera. This robust technique allows us to perform quick comparison inspection but some conditions must be met:

- stable light conditions,
- position of the camera and the object must be still,
- precise object positioning,
- the **camera calibration** must be performed if non-telecentric lens is used.

Most applications use the **Template Matching** technique for finding objects and then matched rectangle is compared. Golden template image and image to compare must have this same dimensions. To get best results filter **CropImageToRectangle** should be used. Please notice that filter **CropImageToRectangle** performs cropping using a real values and it has sub-pixel precision.

## Example

Example below shows how to use a basic golden template matching.



Example application performs following operations

1. Finding object location using the **Template Matching** technique.
2. Comparing input image with previously prepared golden template.

# Deep Learning

Table of contents:

1. Overview
2. Anomalies detection
3. Features detection (segmentation)
4. Object classification
5. Instance segmentation
6. Troubleshooting

## 1. Overview

Deep Learning is a milestone machine learning technique in computer vision. It automatically learns from the training images provided and can effectively generate the solutions for a wide range of applications with minimum effort. Main advantages of this technique are: simple configuration, short development time, versatility of possible applications, robustness to noise and high performance.

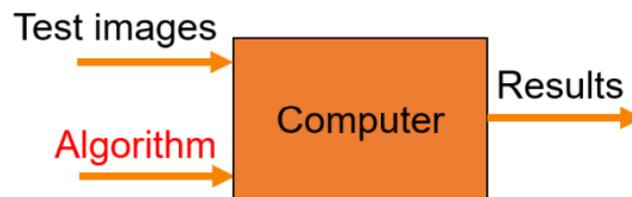
Common applications:

- detection of surface and shape defects (e.g. cracks, deformations, discoloration),
- detecting anomalous samples (e.g. missing, broken or low-quality parts),
- identification of objects/images with respect to predefined classes (i.e. sorting machines),
- localization, segmentation and classification of multiple objects within an image (i.e. bin picking),
- quality analysis in variable environments.

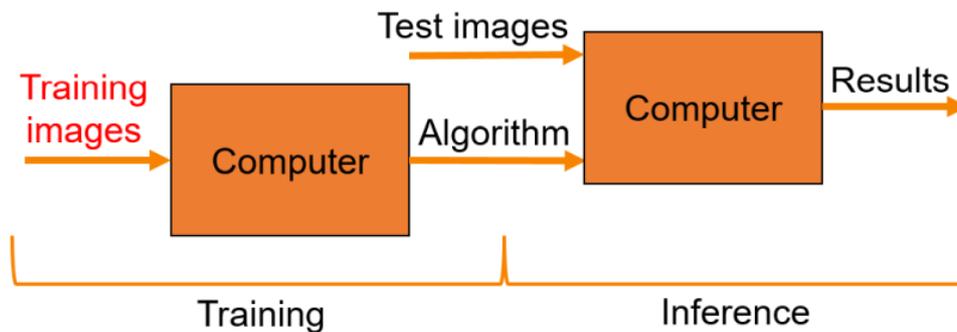
Using deep learning functionality includes two stages:

1. Training - generating a model based on features learned from training samples (using **Deep Learning Editor**),
2. Inference - applying the model on new images in order to perform certain machine vision tasks.

The difference from a classic image processing approach is presented in diagrams below:



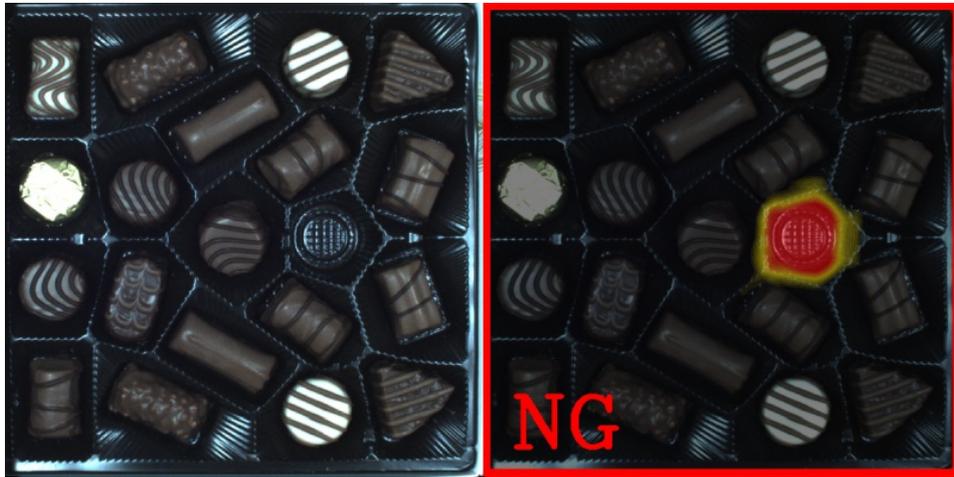
*Classic approach: the algorithm is the missing element that needs to be designed by a human specialist.*



*Machine learning approach: only the manually labeled training images need to be provided.*

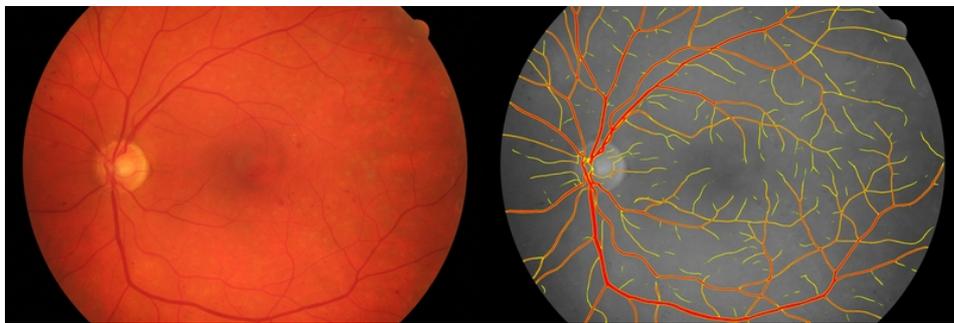
## Available Deep Learning tools

1. **Anomalies detection** - This technique is used to detect anomalous samples. It only needs a set of fault-free samples to learn the model of normal appearance. Optionally, several faulty ones can be useful to define the level of tolerable variations. This tool is useful especially in cases where defects are unknown, too difficult to define upfront or highly variable. The output of this tool are a classification result (normal or faulty), an abnormality score and a heatmap of defects in the image.



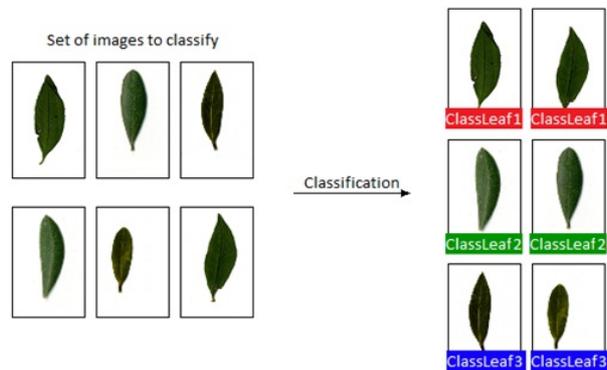
An example of missing part detection using **DeepLearning\_DetectAnomalies2** tool. Left: The original image with a missing element. Right: The classification result with a heatmap of defects overlay.

- Features detection (segmentation)** - This technique is used to precisely segment one or more classes of features within an image. The pixels belonging to each class must be marked by the user in the training step. The result of this technique is an array of probability maps for every class.



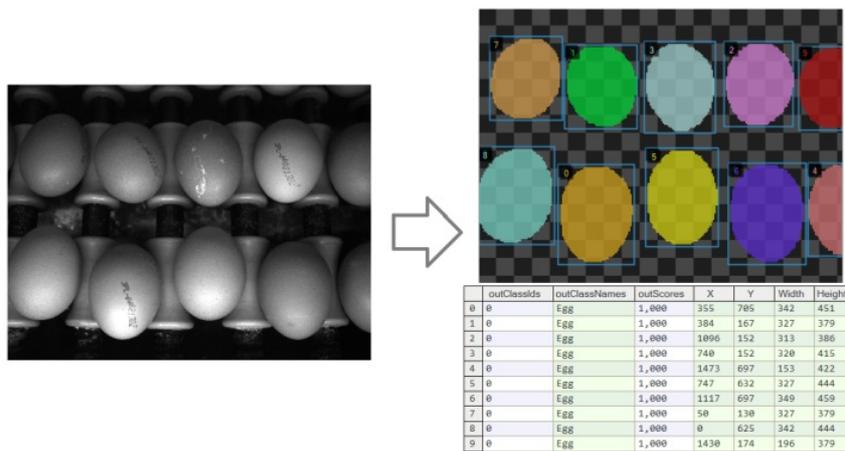
An example of image segmentation using **DeepLearning\_DetectFeatures** tool. Left: The original image of the fundus. Right: The segmentation of blood vessels.

- Object classification** - This technique is used to mark objects/images with one of the predefined classes. First, it is necessary to provide a training set of labeled images. The result of this technique are a name of a class and a classification confidence level for a given image.



An example of object classification using **DeepLearning\_ClassifyObject** tool.

- Instance segmentation** - This technique is used to locate, segment and classify single or multiple objects within an image. The training requires a user to draw regions corresponding to objects on an image and assign them to classes. The result of this technique are lists with elements describing detected objects - their bounding boxes, masks (segmented regions), class IDs, names and membership probabilities.



An example of instance segmentation using **DeepLearning\_SegmentInstances** tool. Left: The original image. Right: The resulting list of detected objects.

## Basic terminology

The users do not need to be equipped with the specialistic scientific knowledge to design their own deep learning solutions. However, it may be very useful to understand the basic terminology and principles behind the process.

### Deep neural networks

Adaptive Vision gives access to deep convolutional neural networks architectures created, adjusted and tested to solve industrial-grade machine vision tasks. Each network is a set of trainable convolutional filters and neural connections which can model complex transformations of the image to extract relevant features and use them in order to solve particular problem. However, they are useless without proper amount of good quality data provided for training process (adjusting weights of filters and connections). This documentation gives necessary practical hints on preparing an effective deep learning model.

Due to various levels of tasks complexity and different expected execution times, the users can choose one of five available network depths. The **Network depth** parameter is an abstract value defining the memory capacity of the network (i.e., the number of layers and filters of a network) and ability to solve more complex problems. The list below gives hints about selecting the proper depth for a task characteristics and conditions.

1. Low depth (value 1-2)
  - o a problem is simple to define,
  - o a problem could be easily solved by a human inspector,
  - o a short time of execution is required,
  - o background and lightning do not change across images,
  - o well-positioned objects and good quality of images.
2. Standard depth (default, value 3)
  - o suitable for a majority of applications without any special conditions,
  - o a modern CUDA-enabled GPU is available.
3. High depth (value 4-5)
  - o **a big amount of training data is available,**
  - o a problem is hard or very complex to define and solve,
  - o complicated irregular patterns across images,
  - o long training and execution times are not a problem,
  - o a large amount of GPU RAM ( $\geq 4\text{GB}$ ) is available,
  - o varying background, lightning and/or positioning of objects.

Tip: test your solution with a lower depth first, and then increase it if needed.

Note: a higher network depth will lead to a significant increase in memory and computational complexity of training and execution.

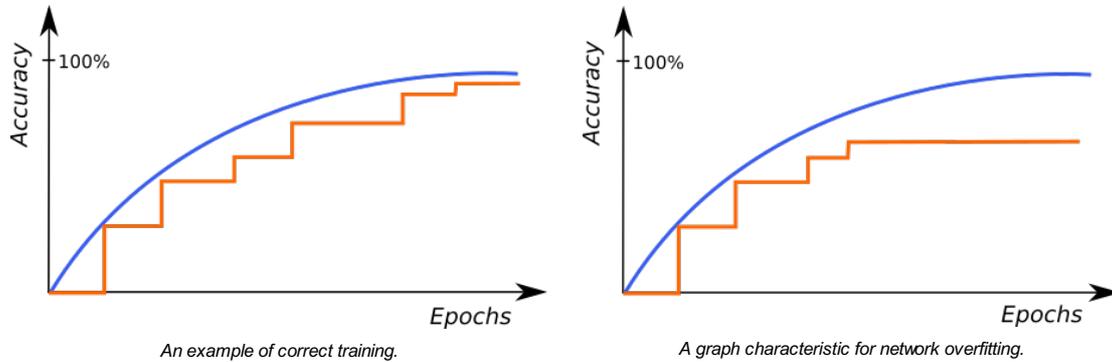
### Training

Model training is an iterative process of updating neural network weights based on the training data. One **iteration** involves some number of steps (determined automatically), each step consists of the following operations:

1. selection of a small subset (**batch**) of training samples,
2. calculation of network error for these samples,
3. updating weights to achieve lower error for these samples.

At the end of each iteration, the current model is evaluated on a separate set of validation samples selected before the training process. Validation set is automatically chosen from the training samples. It is used to simulate how neural network would work with real images not used during training. **Only a set of network weights corresponding with the best validation score at the end of training is saved as the final solution.** Monitoring the training and validation score (blue and orange lines in the figures below) in consecutive iterations gives fundamental information about the progress:

1. both training and validation scores are improving - keep training, model can still improve,
2. both training and validation scores has stopped improving - keep training for a few iterations more and stop if still no change,
3. training score is improving, but validation score has stopped or is going worse - you can stop training, model has probably started **overfitting** your training data (remembering exact samples rather than learning rules about features). It may be caused by too small amount of diverse samples or too low complexity of problem for a network selected.



Above graphs represent training progress in our Deep Learning Editor, the blue line indicates the performance on training samples, and the orange line represents the performance on validation samples. Please note the blue line is plotted more frequently than the orange line, as validation performance is verified only at the end of each iteration.

## Preprocessing

To adjust the performance on your particular task, the user can apply some additional transformations to input images before training starts:

1. **Downsample** — reduction of image size to accelerate training and execution times at the expense of lower level of details possible to detect. Increasing this parameter by 1 will result in downsampling by the factor of 2 over both image dimension.
2. **Convert to grayscale** — while working with problems where color does not matter, you can choose to work with monochrome versions of images.

## Augmentation

In case when the number of training images can be too small to represent all possible variations of samples, it is recommended to use data augmentation that adds artificially modified samples during training. This option can also help avoiding overfitting.

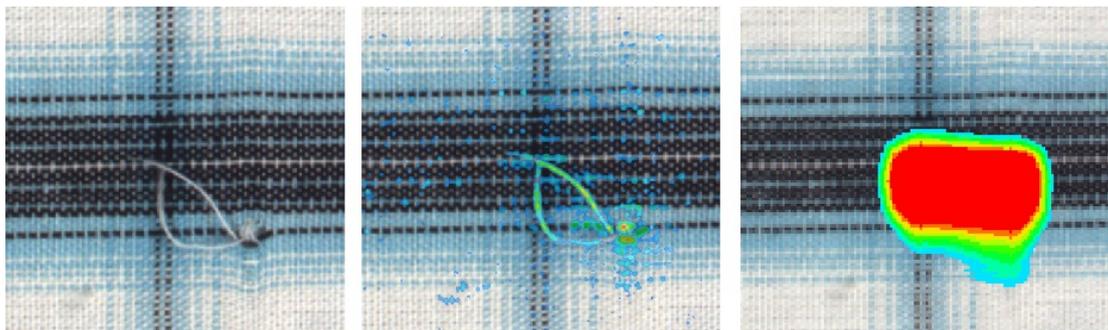
Available augmentations are:

1. **Luminance** — if greater than 0% samples with random image brightness changes will be added. The value of the parameter defines the percentage range of maximum brightness changes (both darker and brighter) relatively to the full image range.
2. **Noise** — if greater than 0% samples with random uniform noise will be added. Each channel and pixel is modified separately in the range defined by percentage of the full image range.
3. **Gaussian Blur** — if greater than 0, samples after Gaussian blurring will be added. Size of the kernel is randomly selected between 0 and provided maximum kernel size. Standard deviation is calculated automatically on the basis of obtained kernel size.
4. **Rotation** — if greater than 0° randomly rotated (by ± parameter value degrees) samples will be added.
5. **Flip Up-Down** — if enabled samples reflected along the X axis will be added.
6. **Flip Left-Right** — if enabled samples reflected along the Y axis will be added.
7. **Relative Translation** — if greater than 0%, shifted samples will be added. Horizontal shift is independent of vertical shift. Both shifts can be negative. Maximum translation is equal to given percent of tile or image (depending on mode) size.
8. **Scale** — if minimum or maximum scale is different than 100%, scaled (by value between minimum scale and maximum scale) samples will be added.
9. **Horizontal Shear** — if greater than 0° horizontally sheared (by angle in ± parameter value range) samples will be added. After this transformation, height of tile (or image) does not change. Angle is measured between vertical line and the same line after transformation.
10. **VerticalShear** — analogous to Horizontal Shear.

Warning: the choice of augmentation options depends only on the task we want to solve, sometimes they might be harmful for quality of a solution. For simple example, enabling the Rotation should not be used if the rotations are not expected in a production environment. Enabling augmentations also increases the network training time (but does not affect execution time!)

## 2. Anomalies detection

Deep Learning Add-on provides two variants of the DetectAnomalies tool: **DeepLearning\_DetectAnomalies** and **DeepLearning\_DetectAnomalies2**. In most cases, the second variant is recommended. The only exception is when very precise defect heatmaps are needed, even at the expense of higher computational time. The difference between them is presented in the example below.



Left: The original image with a defect. Middle: An example of precise but much slower defect detection using **DeepLearning\_DetectAnomalies**. Right: An example of fast defect detection using **DeepLearning\_DetectAnomalies2**.

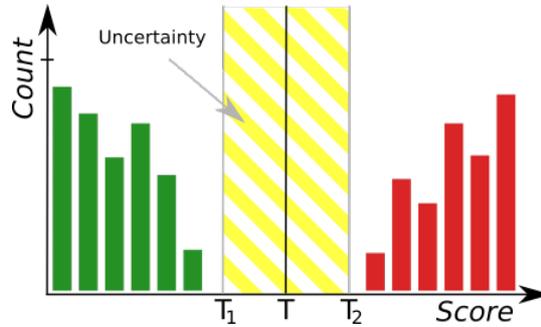
## Interactive histogram tool

DetectAnomalies filters measure deviation of samples from normal image appearance learned during training phase. If the deviation exceeds a given threshold, the image is marked as defected. The suggested threshold is automatically calculated after the training phase but can be adjusted by user in the Deep Learning Editor using the interactive histogram tool described below.

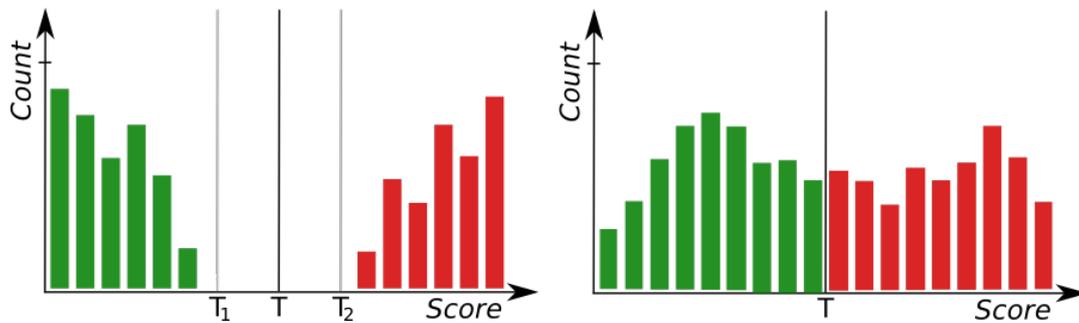
After the training phase, scores are calculated for every training sample and are presented in the form of histogram; good samples are marked with green, bad samples with red bars. In the perfect case, the scores for good samples should be all lower than for bad samples and the threshold should be automatically calculated to give the optimal accuracy of the model. However, the groups may sometimes overlap because of:

1. incorrectly labeled samples,
2. bad **Feature size**,
3. ambiguous definition of the expected defects,
4. high variability of the samples appearance or environmental conditions.

In order to achieve more robust threshold, it is recommended to perform training with a large number of samples from both groups. If the number of samples is limited, our software allows to manually set the uncertainty area with additional thresholds (the information about the confidence of the model can be than obtained from the hidden outIsConfident filter port for each sample).



The histogram tool where green bars represent correct samples and red bars represent defected samples.  $T$  marks the main threshold and  $T_1$ ,  $T_2$  define the area of uncertainty.



Left: a histogram presenting well-separated groups indicating a good accuracy of the model. Right: a poor accuracy of the model.

## Feature size

This parameter defines the expected defect size and is the most significant one in terms of both quality and speed of inspection. It is represented by a gray square in the Image window of the Editor. It need to be adjusted to a specific application using hints below:

- the Feature size should be large enough to contain common defects with some margin,
- too small Feature size may often lead to increased complexity of the model and longer processing time,
- it is better to try first with a larger Feature size (small defects should be still detected) or using the Downsample option,
- too large Feature size may lead to less precise results and lower resolution of the inspection,
- the Feature size is examined to work best in the range of 48-164.

## Sampling density

This parameter controls the spatial resolution of both training and inspection. The larger the density the more precise results but longer computational time. It is recommended to use the Low density only for well positioned and simple objects. The High density is useful when working with complex textures and highly variable objects.

## Global and Local network types

The **DeepLearning\_DetectAnomalies** filter gives an option to choose between Local and Global types of processing. The Local processing is the default one and is used to analyze images in **fragments** of size determined by the Feature size parameter. The Global processing is used to analyze images holistically and is recommended for simple applications where objects are well-positioned and have large defects, like in the image below. The Global-like processing can be achieved in **DeepLearning\_DetectAnomalies2** by setting Feature size close to the image width or height.



An example of cookie shape defects detection using the Global processing of the **DeepLearning\_DetectAnomalies** filter.

## Model usage

Use **DeepLearning\_DetectAnomalies** or **DeepLearning\_DetectAnomalies2** filters.

DeepLearning_DetectAnomalies +			
inImage		inDeepModel	
outHeatmap		outIsValid	outScore

DeepLearning_DetectAnomalies2 +			
inImage		inDeepModel	
outHeatmap		outIsValid	outScore

## 3. Feature detection (segmentation)

This technique is used to precisely mark pixels belonging to one or more classes called features of the image. They should be repeatable across the dataset and easy to define because they need to be marked by the user first. A few common examples of features to detect using this filter are:

- characteristic edges, lines or points,
- characteristic patterns and small objects,
- repeatable defects.

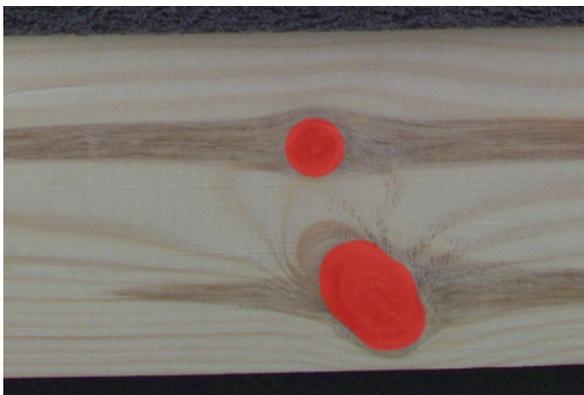
### Preparing the data

Images loaded to the Editor of **DeepLearning\_DetectFeatures** can be of different sizes and can have different ROIs defined. However, it is important to ensure scale and appearance of the features to be consistent with the production environment.

The features can be marked using [intuitive interface](#) in the Editor or can be imported as masks from a file.

Each and every feature should be marked on all training images, or ROI should be limited to include only marked defects. Inconsistently marked features are one of the main reasons of poor accuracy.

The marking precision should be adjusted to the application requirements. The more precise marking the better accuracy in the production environment. While marking with low precision it is better to mark features with some excess margin.



An example of knots marked with low precision.



An example of cracks marked with high precision.

### Multiple classes of features

It is possible to detect many classes of features separately using the same model. For example, road and building like in the image below. Features may overlap but it is usually not recommended. Also, it is not recommended to define more than a few different classes in a single model.



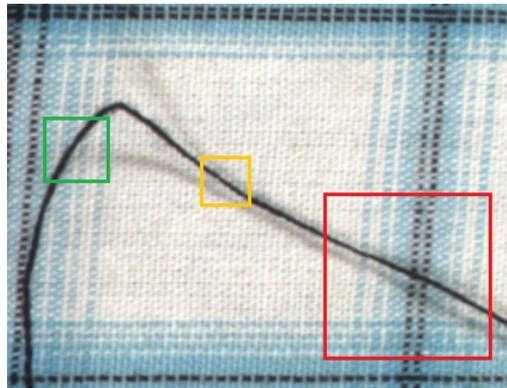
An example of marking two different classes (red roads and yellow buildings) in the one image.

## Feature size

Rules are similar to the ones from [the previous section](#). Especially, it is important to avoid too small values, they are one of the main reasons of poor accuracy and long processing time. The best results should be achieved when Feature size allows to include a defect together with a part of its surroundings (context). A trained network examines the input image only by looking at it through windows defined by Feature size. For this reason, it is recommended to follow the rule that an operator, using the same context, should be able to properly decide whether a given sample contains a defect or not.

Performance tip: a larger Feature size increases the training time and needs more GPU memory and training samples to operate effectively. When feature size exceeds 128 pixels and still looks too small, it is worth considering the **Downsample** option.

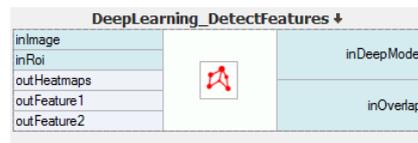
Performance tip: if the execution time is not satisfying you can set the `inOverlap` filter input to `False`. It should speed up the inspection by 10-30% at the expense of less precise results.



Examples of feature size: too large (red), optimal (green) and acceptable (orange). Remember that this is just a heuristic and can vary in some cases.

## Model usage

Use **DeepLearning\_DetectFeatures** filter to perform segmentation of features.



Parameters:

- To limit the area of image analysis you can use **inRoi** input.
- To increase segmentation precision you can use **inOverlap** option.
- Feature segmentation results are passed in a form of bit maps to **outHeatmaps** output as an array and **outFeature1**, **outFeature2**, **outFeature3** and **outFeature4** as separate images.

Notice: Enabling **inOverlap** option increases segmentation quality, but also prolongs feature segmentation process.

## 4. Object classification

This technique is used to identify class of object within an image.

### Principle of operation

During the training, object classification learns the representation of user defined classes. Model uses generalized knowledge gained from samples provided for training, and aims to obtain good separation between classes.



Result of classification after training.

After the training is completed, user is presented with confusion matrix. It indicates, how well model separated user defined classes. It simplifies identification of model accuracy, when large number of samples is presented.

Confusion matrix:

		Actual			
		Class1	Class2	Class3	Class4
Predicted	Class1	3	0	0	0
	Class2	0	3	0	0
	Class3	0	0	3	0
	Class4	0	0	0	3

Confusion matrix presents assignment of samples to user defined classes.

## Training parameters

Object classification has been designed to minimize the time needed to perform classification task on images provided by user. There are no additional parameters apart from default ones. [List of parameters available for all Deep Learning algorithms.](#)

## Model usage

Use **DeepLearning\_ClassifyObject** filter to perform classification.

DeepLearning_ClassifyObject +		
inImage		inDeepModel
inRoi		outClassName
		outClassIndex
		outScore

Parameters:

- To limit the area of image analysis you can use **inRoi** input.
- Classification results are passed to **outClassName** and **outClassIndex** outputs.
- Score value **outScore** indicates confidence of classification.

## 5. Instance segmentation

This technique is used to locate, segment and classify single or multiple objects within an image. The result of this technique are lists with elements describing detected objects - their bounding boxes, masks (segmented regions), class IDs, names and membership probabilities.

Note that in contrary to feature detection technique, instance segmentation detects individual objects and may be able to separate them even if they touch or overlap. On the other hand, instance segmentation is not an appropriate tool to detect and segment features, e.g. scratches or cracks.



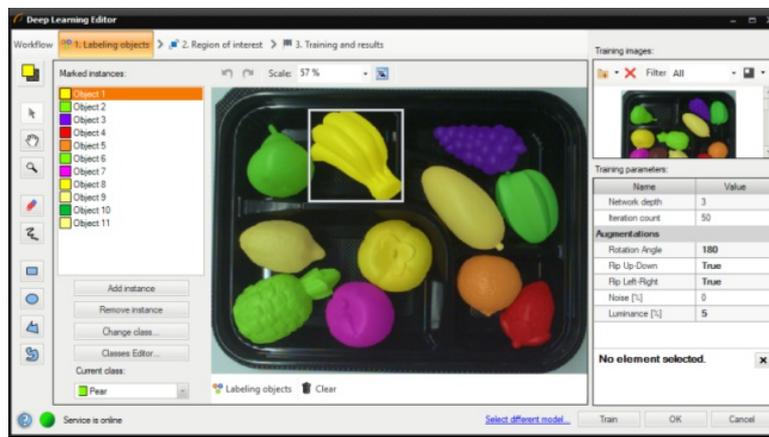
Original image.



Visualized instance segmentation results.

## Preparing the data

The training requires a user to draw regions corresponding to objects on an image and assign them to classes.



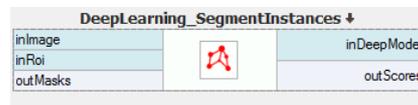
Editor for marking objects.

## Training parameters

Instance segmentation training adapts to the data provided by a user and does not require any additional training parameters besides the default ones.

## Model usage

Use **DeepLearning\_SegmentInstances** filter to perform instance segmentation.



Parameters:

- To limit the area of image analysis you can use **inRoi** input.
- To set minimum detection score **inMinDetectionScore** parameter can be used.
- Maximum number of detected objects on a single image can be set with **inMaxObjectsCount** parameter.
- Results describing detected objects are passed to following outputs:
  - bounding boxes: **outBoundingBoxes**,
  - class IDs: **outClassIds**,
  - class names: **outClassNames**,
  - classification scores: **outScores**,
  - masks: **outMasks**.

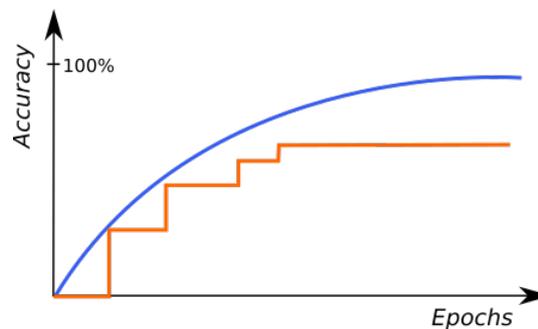
## 6. Troubleshooting

Below you will find a list of most common problems.

### 1. Network overfitting

A situation when network loses its ability to generalize over available problems and focuses only on test data.

Symptoms: during training, the validation graph stops at one level and training graph continues to rise. Defects on training images are marked very precisely, but defects on new images are marked poorly.



A graph characteristic for network overfitting.

Causes:

- The number of test samples is too small,
- Training time is too long.

Solution:

- Provide more real samples,
- Add more samples with possible object rotations.

## 2. Susceptibility to changes in lighting conditions.

Symptoms- network is not able to process images properly when even minor changes in lighting occur.

Causes:

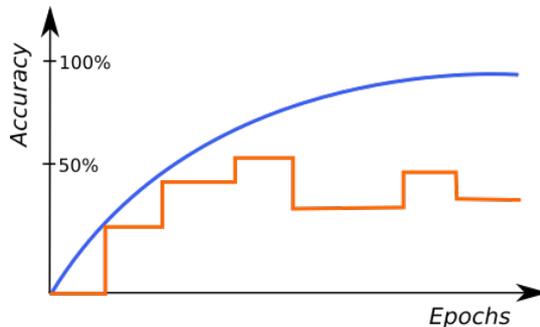
- Samples with variable lighting were not provided.

Solution:

- Provide more samples with variable lighting,
- Enable "Luminance" option for automatic lighting augmentation.

## 3. No progress in network training.

Symptoms — even though the training time is optimal, there is no visible training progress.



*Training progress with contradictory samples.*

Causes:

- The number of samples is too small or the samples are not variable enough,
- image contrast is too small,
- the chosen network architecture is too small,
- contradiction in defect masks.

Solution:

- Modify lighting to expose defects,
- Remove contradictions in defect masks.

Tip: Remember to mark all defects of a given type on the input images or remove images with unmarked defects. Marking only a part of defects of a given type may negatively influence the network learning process.

## 4. Training/sample evaluation is very slow

Symptoms — training or sample evaluation takes a lot of time.

Causes:

- Resolution of the provided input images is too high,
- Fragments that cannot possibly contain defects are also analyzed.

Solution:

- Enable "Downsample" option to reduce the image resolution,
- Limit ROI for sample evaluation.

See also:

- [Deep Learning Service Configuration](#) - installation and configuration of Deep Learning service,
- [Creating Deep Learning Model](#) - how to use Deep Learning Editor.

