Zebra
Aurora™ Vision

Aurora Vision Library 5.3

Machine Vision Guide

Created: 6/8/2023

Product version: 5.3.4.94078

Table of content:

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:

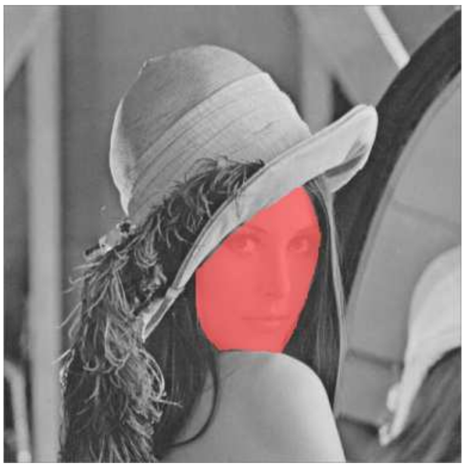
- JPEG compression (creates artifacts not visible by human eye, but disruptive for algorithms)
- CIE Lab and CIE XYZ color spaces (specifically designed for human perception)
- Edge enhancement filters (which improve only the "apparent sharpness")
- Image thresholding performed before edge detection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

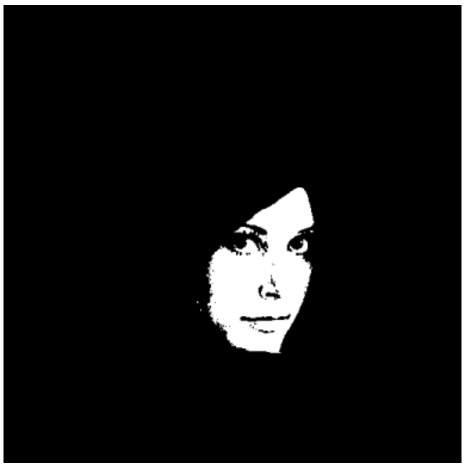
Examples of image processing operations that can really improve information extraction are:

- 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 Aurora 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 Image Local Transforms 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:

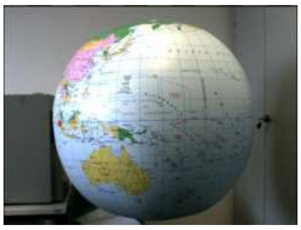
- Never assume any specific value outside of the image, unless specifically defined by the user.
- If only partial information is available, it is better not to detect anything, than detect something that does not exist.

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 Image Combinators 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.

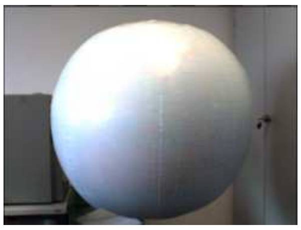


Image of the reflections (calibrating).

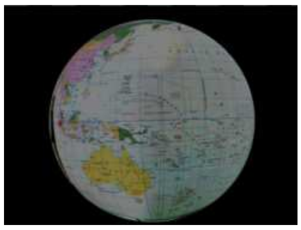
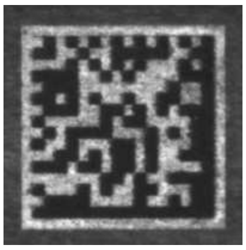
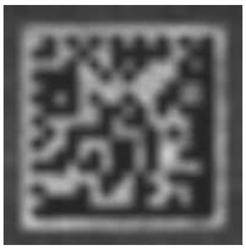

The result of applying DivideImages with inScale = 128 (inRoi was used).

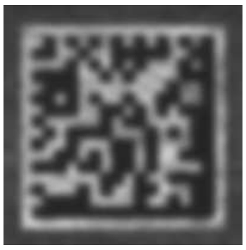
Image Smoothing

The main purpose of the image smoothing filters (located in the Image Local Transforms 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.

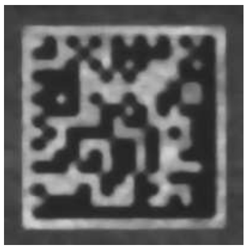
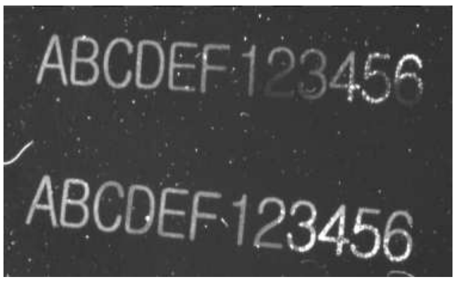
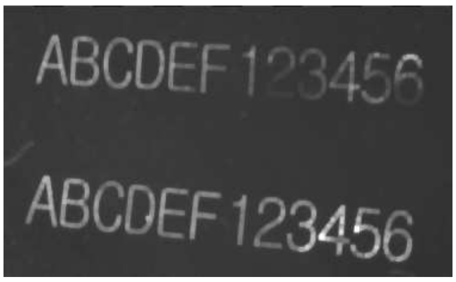

Result of applying SmoothImage_Median.

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.



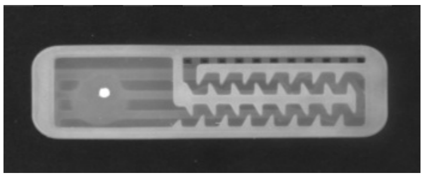
Result of applying OpenImage.

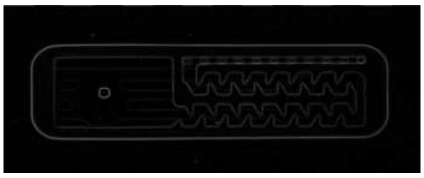
Gradient Analysis

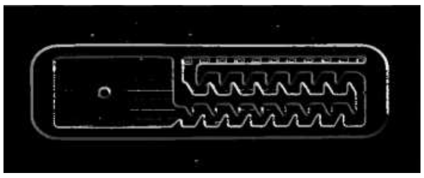
An image gradient is a vector describing direction and magnitude (strength) of local brightness changes. Gradients are used inside of many computer vision tools – for example in object contour detection, edge-based template matching and in barcode and DataMatrix detection.

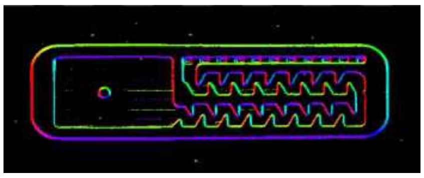
Available filters:

- GradientImage – produces a 2-channel image of signed values; each pixel denotes a gradient vector.
- GradientMagnitudeImage – 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.

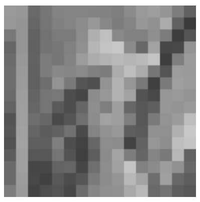
Spatial Transforms

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.

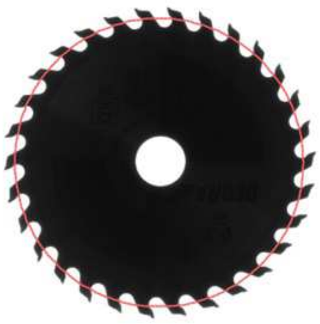
Result of TransposeImage.

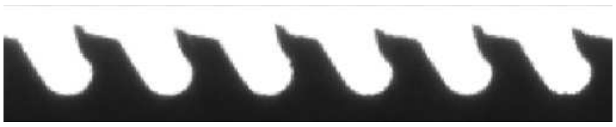
Result of TranslateImage.

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:

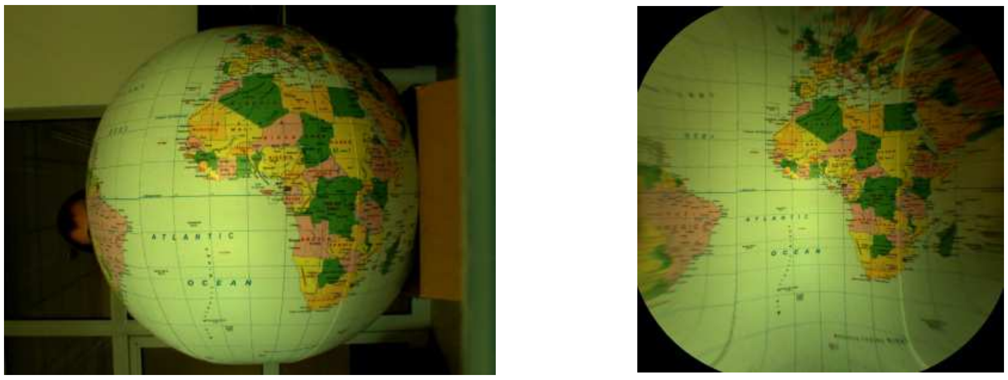
1. compute the destination coordinates (and some coefficients when interpolation is used),
2. copy the pixel value.

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. Aurora Vision provides the Image Spatial Transforms Maps category of filters for exactly that purpose. When you are able to compute the transform beforehand, storing it in the SpatialMap type, in the main loop only the RemapImage filter has to be executed. This approach will be much faster than using standard spatial transform tools.

The SpatialMap type is a map of image locations and their corresponding positions after given geometric transformation has been applied.

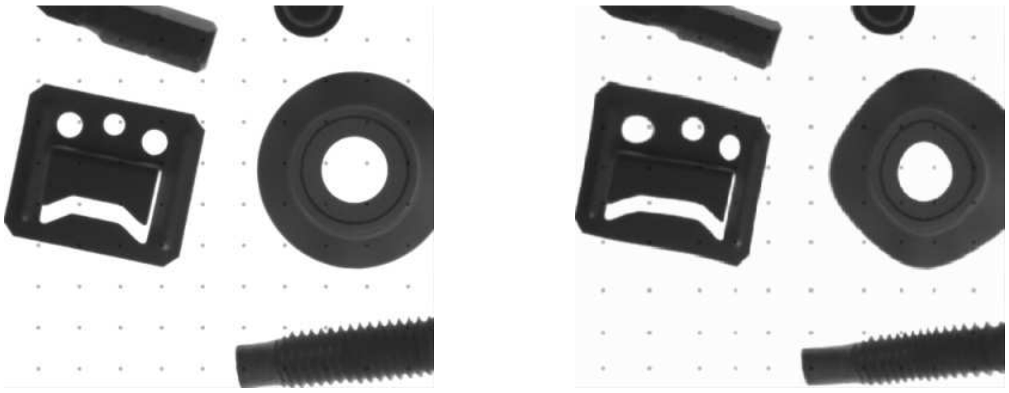
Additionally, the Image Spatial Transforms Maps category provides several filters that can be used to flatten the curvature of a physical object. They can be used for e.g. reading labels glued onto curved surfaces. These filters model basic 3D objects:

1. **Cylinder** (CreateCylinderMap) – e.g. flattening of a bottle label.
2. **Sphere** (CreateSphereMap) – e.g. reading a label from light bulb.
3. **Box** (CreatePerspectiveMap_Points or CreatePerspectiveMap_Path) – e.g. reading a label from a box.
4. **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.

Furthermore custom spatial maps can be created with ConvertMatrixMapsToSpatialMap.

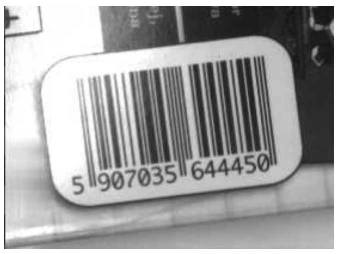


An example of custom image transform created with ConvertMatrixMapsToSpatialMap. Image before and after remapping.

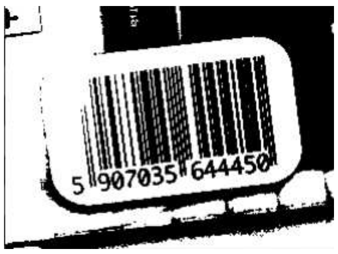
Image Thresholding

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:

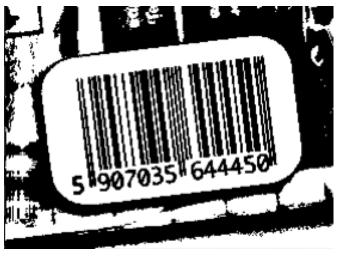
- ThresholdImage_Dynamic and ThresholdToRegion_Dynamic use average local brightness to compensate global illumination variations.
- ThresholdImage_RGB and ThresholdToRegion_RGB select pixel values matching a range defined in the RGB (the standard) color space.
- ThresholdImage_HSx and ThresholdToRegion_HSx 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.

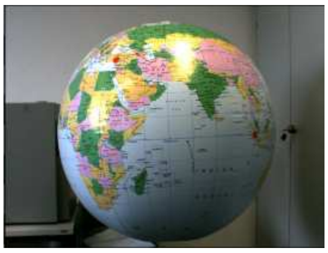


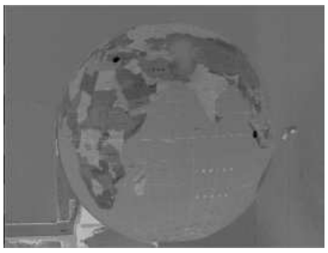
Result of ThresholdImage_Dynamic – the bars are correct.

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.

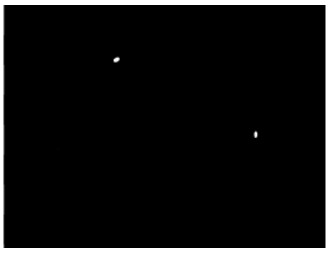
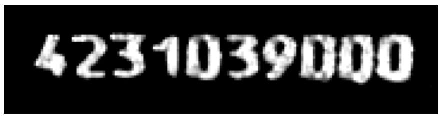
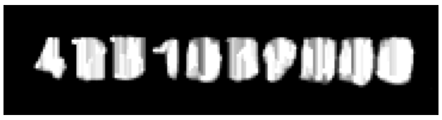
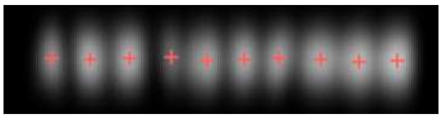

Result of thresholding reveals the location of the red dots on the globe.

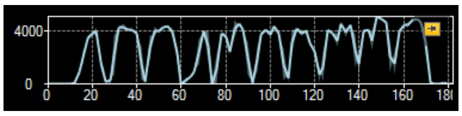
Image Features

Image Features 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.

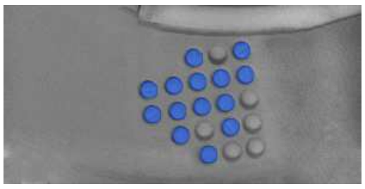

Digit locations extracted by applying SmoothImage_Gauss and ImageLocalMaxima.



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

Blob Analysis

Introduction



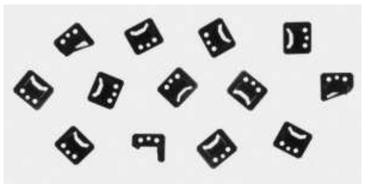
Blob Analysis is a fundamental technique of machine vision based on analysis of consistent image regions. As such it is a tool of choice for applications in which the objects being inspected are clearly discernible from the background. Diverse set of Blob Analysis methods allows to create tailored solutions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

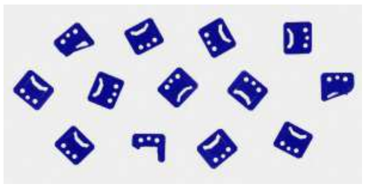
Concept

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

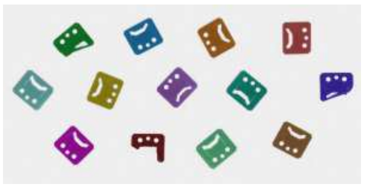
- *Region* is any subset of image pixels. In Aurora Vision Studio regions are represented using Region data type.
- *Blob* is a connected region. In Aurora Vision Studio blobs (being a special case of region) are represented using the same Region data type. They can be obtained from any region using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

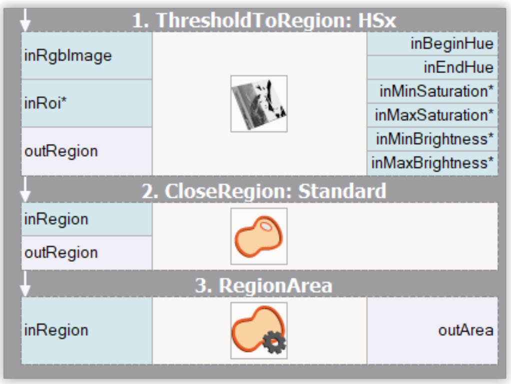
1. **Extraction** - in the initial step one of the Image Thresholding techniques is applied to obtain a region corresponding to the objects (or single object) being inspected.
2. **Refinement** - the extracted region is often flawed by noise of various kind (e.g. due to inconsistent lightning or poor 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).

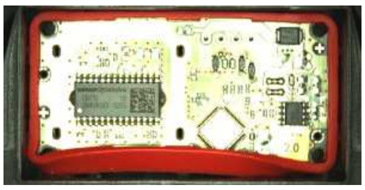


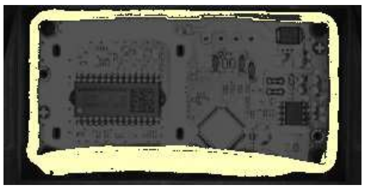
In this case each of the steps: Extraction, Refinement and Analysis is represented by a single filter.

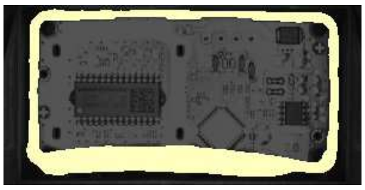
Extraction - to obtain a region corresponding to the red band a Color-based Thresholding technique is applied. The ThresholdToRegion_HSx filter is capable of finding the region of pixels of given color characteristics - in this case it is targeted to detect red pixels.

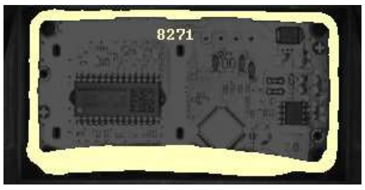
Refinement - the problem of filling the gaps in the extracted region is a standard one. Classic solutions for it are the region morphology techniques. Here, the CloseRegion filter is used to fill the gaps.

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


Initial image


Extraction

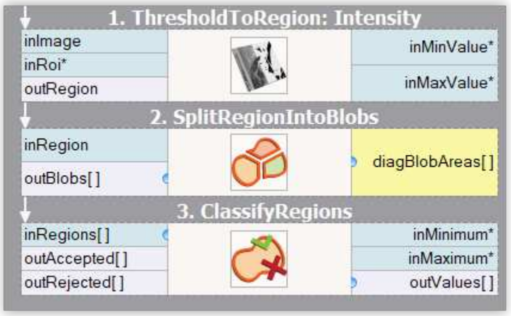

Refinement



Results

Mounts

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

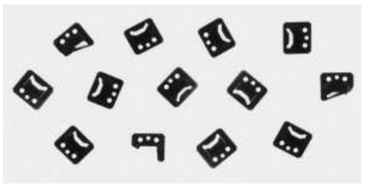


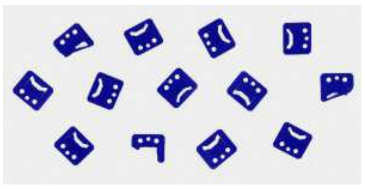
Extraction - as the lightning in the image is uniform, the objects are consistently dark and the background is consistently bright, the extraction of the region corresponding to the objects is a simple task. A basic ThresholdToRegion filter does the job, and does it so well that no **Refinement** phase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

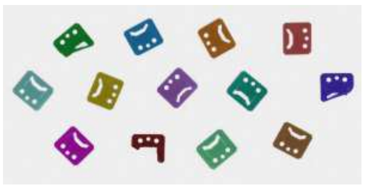
Analysis - as we need to analyze each of the blobs separately, we start by applying the SplitRegionIntoBlobs filter to the extracted region.

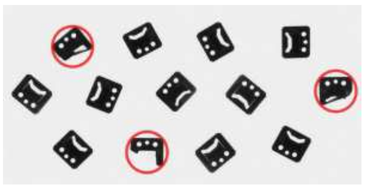
To distinguish the bad parts from the correct parts we need to pick a property of a region (e.g. area, circularity, etc.) that we expect to be high for the good parts and low for the bad parts (or conversely). Here, the area would do, but we will pick a somewhat more sophisticated rectangularity feature,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

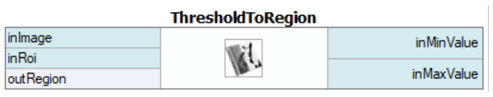
There are two techniques that allow to extract regions from an image:

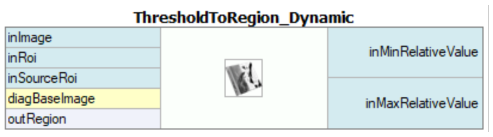
- **Image Thresholding** - commonly used methods that compute a region as a set of pixels that meet certain condition dependent on the specific operator (e.g. region of pixels brighter than given value, or brighter than the average brightness in their neighborhood). Note that the resulting data is always 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

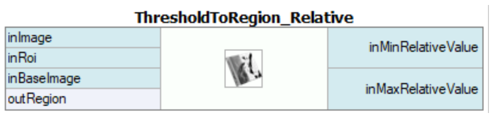
Image Thresholding techniques are preferred for common applications (even those in which a set of objects is inspected rather than a single object) because of their simplicity and excellent performance. In Aurora Vision Studio there are six filters for image-to-region thresholding, each of them implementing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

Brightness-based (basic)

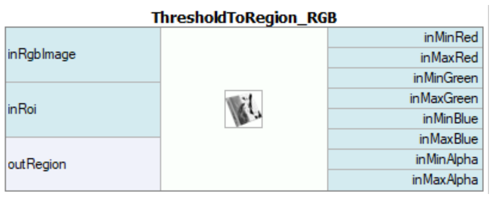


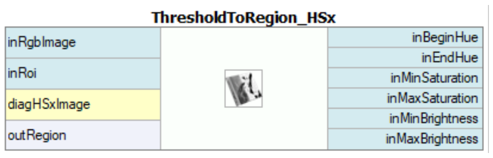


Brightness-based (additional)



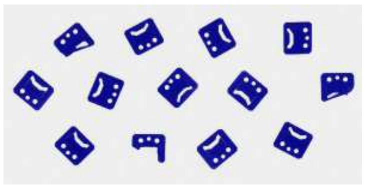
Color-based





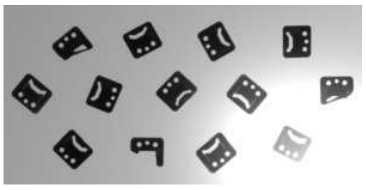
Classic Thresholding

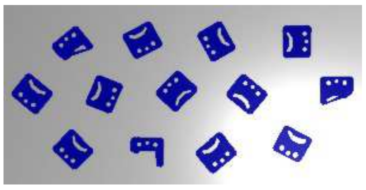
ThresholdToRegion simply selects the image pixels of the specified brightness. It should be considered a basic tool and applied whenever the intensity of the inspected object is constant, consistent and clearly different from the intensity of the background.



Dynamic Thresholding

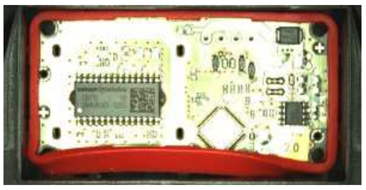
Inconsistent brightness of the objects being inspected is a common problem usually caused by the imperfections of the lightning setup. As we can see in the example below, it is often the case that the objects in one part of the image actually have the same brightness as the background in another part 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.



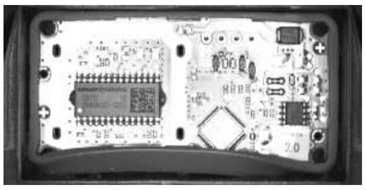


Color-based Thresholding

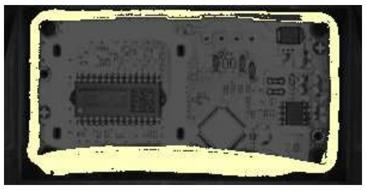
When inspection is conducted on color images it may be the case that despite a significant difference in color, the brightness of the objects is actually the same as the brightness of their neighborhood. In such case it is advisable to use Color-based Thresholding filters: ThresholdToRegion_RGB, ThresholdToRegion_HSx. 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.

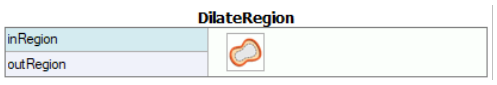
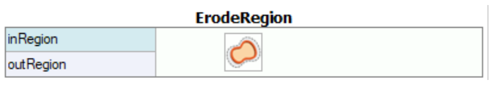
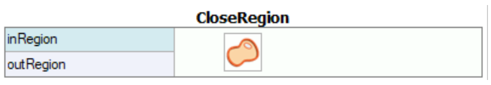
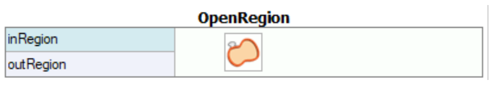


Result of the color-based thresholding targeted at red pixels.

Refinement

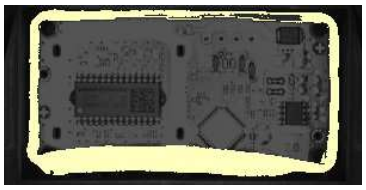
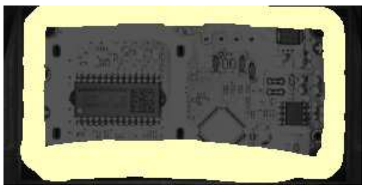
Region Morphology

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.

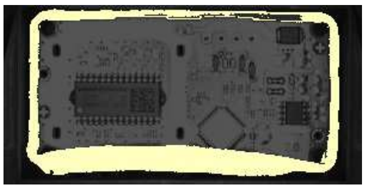
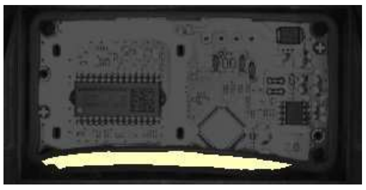
	Expanding	Reducing
Basic		
Composite		

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.

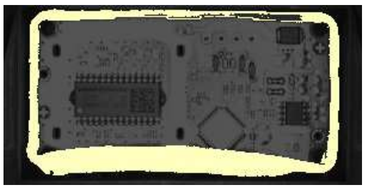
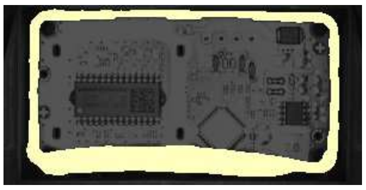



Erosion is a dual operation of **Dilation**. Here, each pixel **P** within the dimensions of the region being transformed is added to the resulting region if and only if the structuring element centered at **P** is *fully contained* in the region pixels. Note that for a circular kernel such transformation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

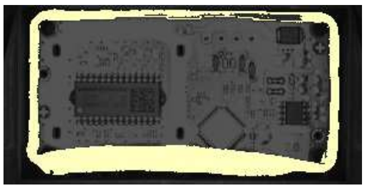
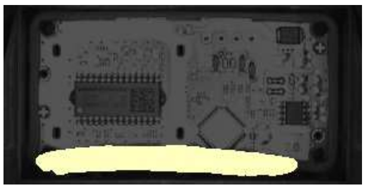



Closing and Opening

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

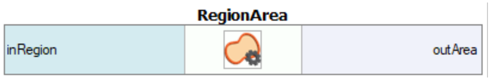
Aurora Vision Studio allows to compute a wide range of numeric (e.g. area) and non-numeric (e.g. bounding circle) region features. Calculation of the measures describing the obtained region is often the very aim of applying the blob analysis in the first place. If we are to check whether the rectangular 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.

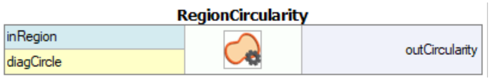
Numeric Features

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

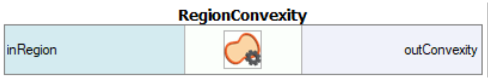
Annotations in brackets indicate the range of the resulting values.



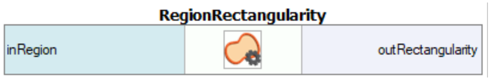
Size of the region (0 - ∞)



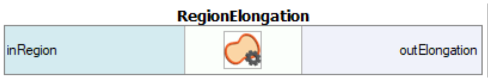
Similarity to a circle (0.0 - 1.0)



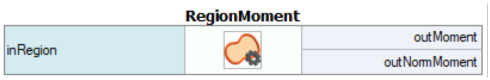
Similarity to own convex hull (0.0 - 1.0)



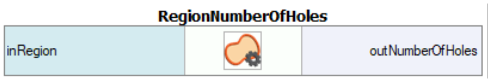
Similarity to a rectangle (0.0 - 1.0)



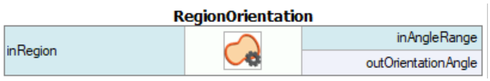
Similarity to a line (0.0 - ∞)



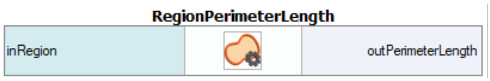
Moments of the region (0.0 - ∞)



Count of the region holes (0 - ∞)



Orientation of the main region axis (0.0 - 180.0)

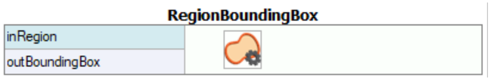


Length of the region contour (0.0 - ∞)

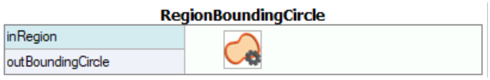
Non-numeric Features

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.

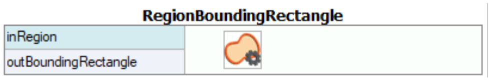
Annotations in brackets indicate Aurora Vision Studio's type of the result.



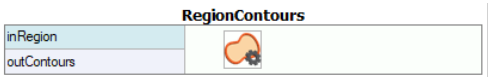
Smallest axis-aligned rectangle containing the region (Box)



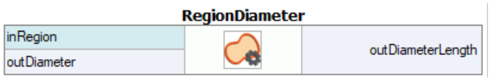
Smallest circle containing the region (Circle2D)



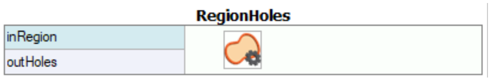
Smallest any-orientation rectangle containing the region (Rectangle2D)



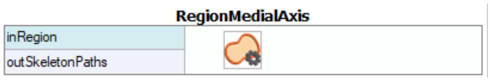
Boundaries of the region (PathArray)



Longest segment connecting two points inside the region (Segment2D)



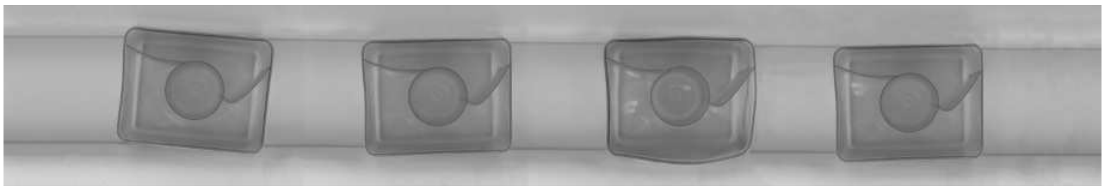
Array of blobs representing gaps in the region (RegionArray)



Skeleton of the region (PathArray)

Case Studies

Capsules

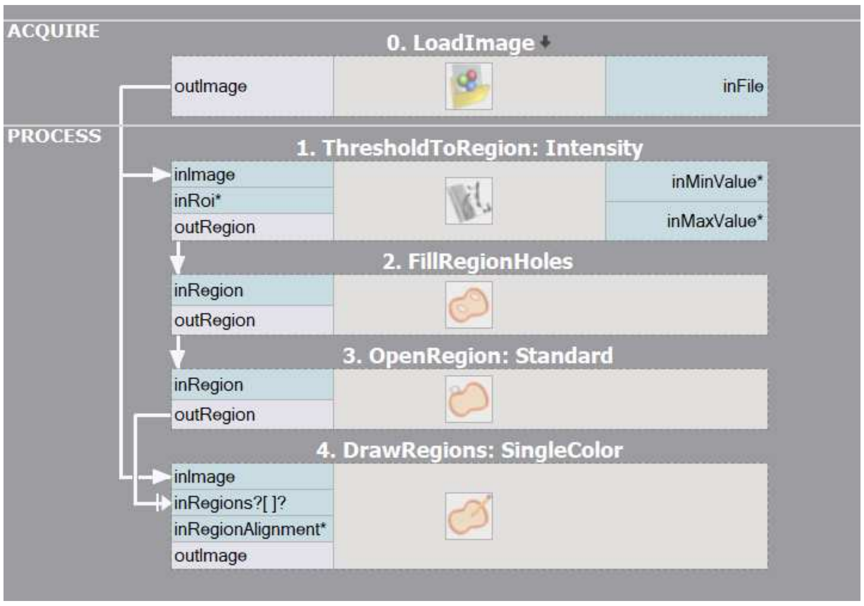


In this example we inspect a set of washing machine capsules on a conveyor line. Our aim is to identify the deformed capsules.

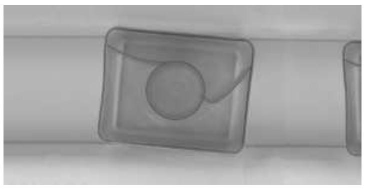
We will proceed in two steps: we will commence by designing a simple program that, given picture of the conveyor line, will be able to identify the region corresponding to the capsule(s) in the picture. In the second step we will use this program as a building block of the complete solution.

FindRegion Routine

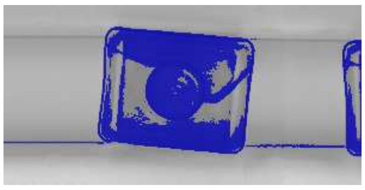
In this section we will develop a program that will be responsible for the **Extraction** and **Refinement** phases of the final solution. For brevity of presentation in this part we will limit the input image to its initial segment.



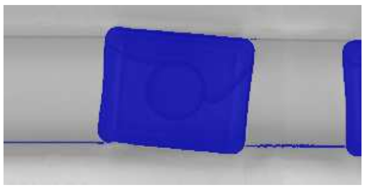
```
ACQUIRE
0. LoadImage
outImage                                  inFile
PROCESS
1. ThresholdToRegion: Intensity
inImage                                   inMinValue*
inRoi*
outRegion                                 inMaxValue*
2. FillRegionHoles
inRegion
outRegion
3. OpenRegion: Standard
inRegion
outRegion
4. DrawRegions: SingleColor
inImage
inRegions?[ ]?
inRegionAlignment*
outImage
```



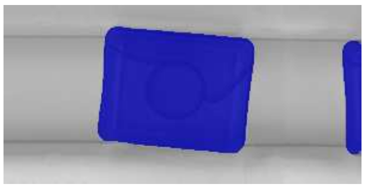
Initial image



ThresholdToRegion



FillRegionHoles



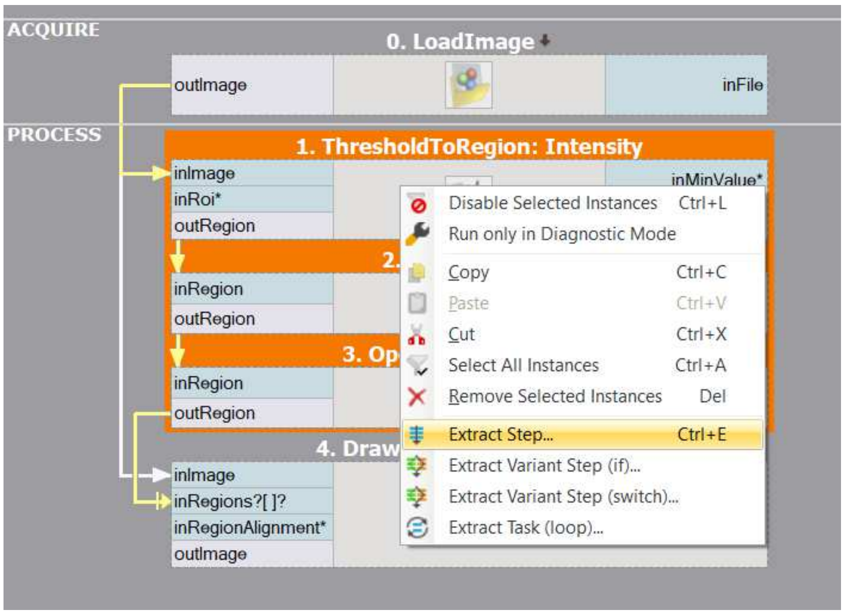
OpenRegion

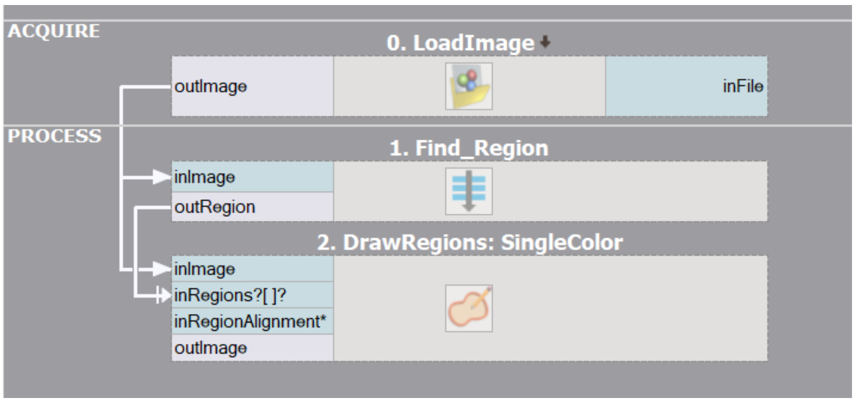
After a brief inspection of the input image we may note that the task at hand will not be trivial - the average brightness of the capsule body is similar to the intensity of the background. On the other hand the border of the capsule is consistently darker than the background. As it is the border of 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.

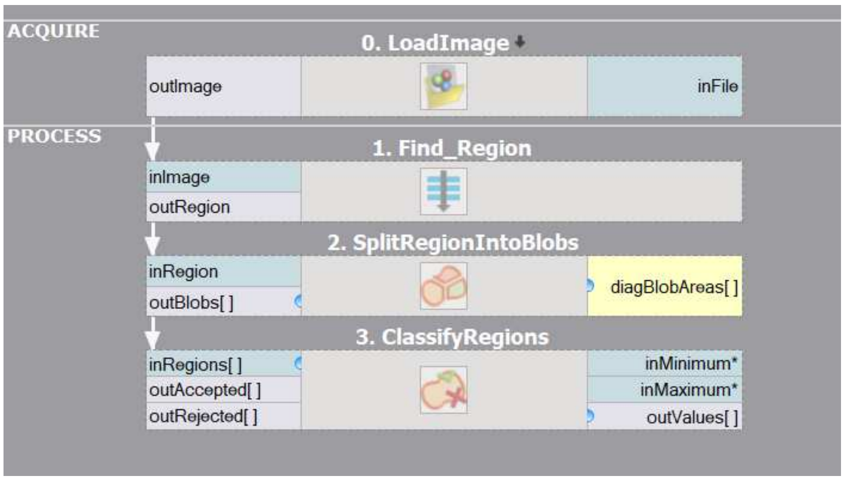
FillRegionHoles will extend the region to include all pixels enclosed by present region pixels. After the region is filled all that remains is the removal of the thin conveyor lines using the classic OpenRegion filter.

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 function to enhance the readability of the soon-to-be-growing program.





Complete Solution



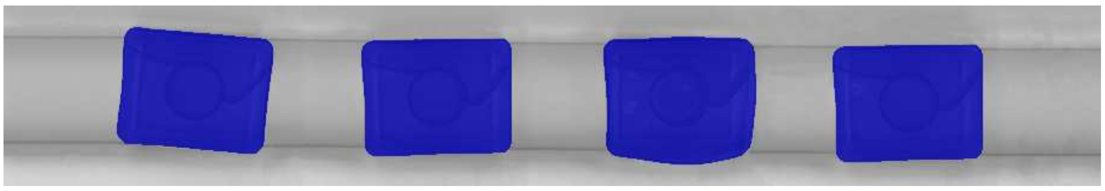
Our program right now is capable of extracting the region that directly corresponds to the capsules visible in the image. What remains is to inspect each capsule and classify it as a correct or deformed one.

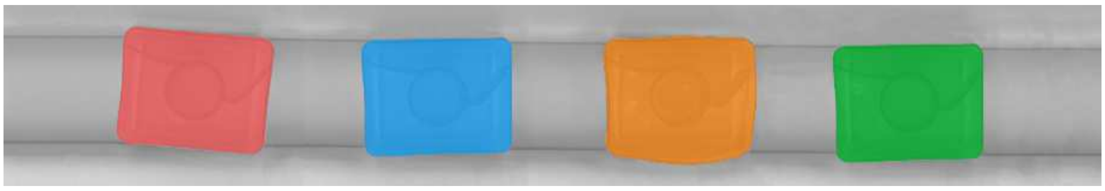
As we want to analyze each capsule separately, we should start with decomposition of the extracted region into an array of connected components (blobs). This common operation can be performed using the straightforward SplitRegionIntoBlobs filter.

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.

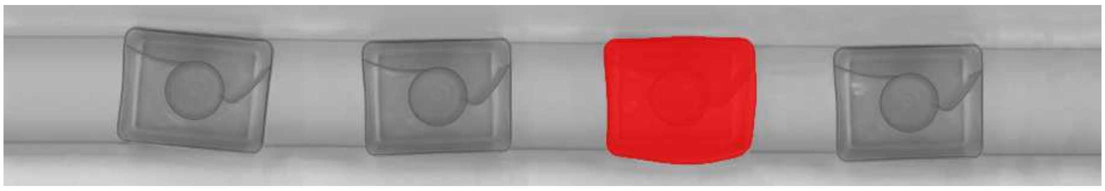
Rectangularity of a shape is defined as the ratio between its area and area of its smallest enclosing rectangle - the higher the value, the more the shape of the object resembles a rectangle. As the shape of a correct capsule is almost rectangular (it is a rectangle with rounded corners) and clearly *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.



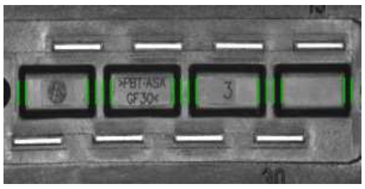
Blobs of low rectangularity selected by ClassifyRegions filter.

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.



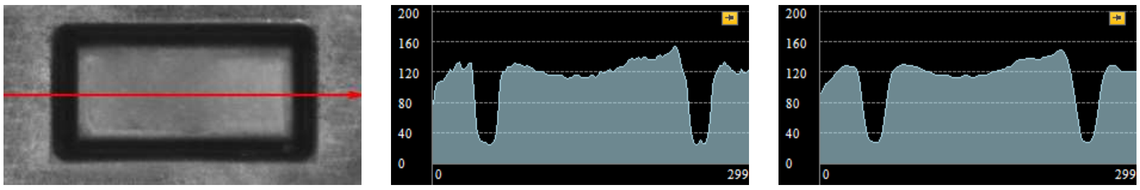
Concept

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.

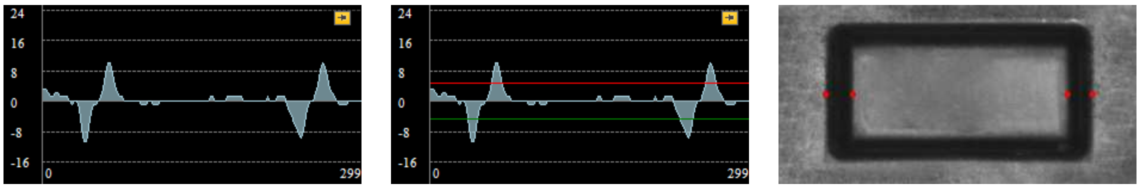
The computation proceeds in the following steps:

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.



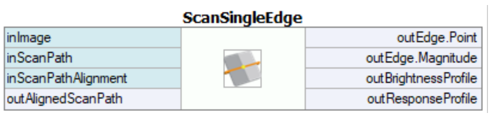
Brightness profile is differentiated. Notice four peaks of the profile derivative which correspond to four prominent image edges intersecting the scan line. Finally the peaks stronger than some selected value (here minimal strength is set to 5) are identified as edge points.

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

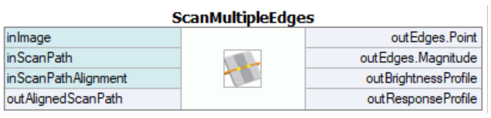
Single Result



ScanSingleEdge

inImage	outEdge.Point
inScanPath	outEdge.Magnitude
inScanPathAlignment	outBrightnessProfile
outAlignedScanPath	outResponseProfile

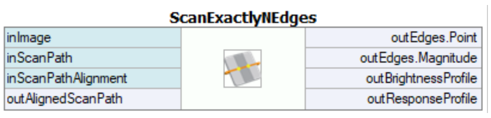
Multiple Results



ScanMultipleEdges

inImage	outEdges.Point
inScanPath	outEdges.Magnitude
inScanPathAlignment	outBrightnessProfile
outAlignedScanPath	outResponseProfile

Fixed Number of Results

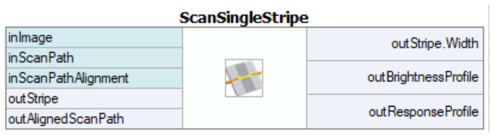


ScanExactlyNEdges

inImage	outEdges.Point
inScanPath	outEdges.Magnitude
inScanPathAlignment	outBrightnessProfile
outAlignedScanPath	outResponseProfile

Stripes

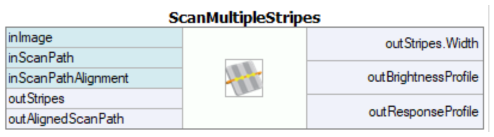
Single Result



ScanSingleStripe

inImage		outStripe.Width
inScanPath		outBrightnessProfile
inScanPathAlignment		outResponseProfile
outStripe		
outAlignedScanPath		

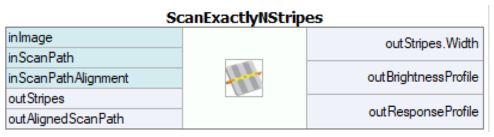
Multiple Results



ScanMultipleStripes

inImage		outStripes.Width
inScanPath		outBrightnessProfile
inScanPathAlignment		outResponseProfile
outStripes		
outAlignedScanPath		

Fixed Number of Results

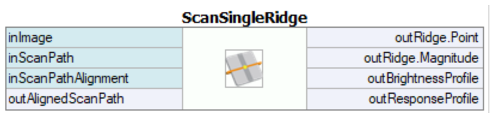


ScanExactlyNStripes

inImage		outStripes.Width
inScanPath		outBrightnessProfile
inScanPathAlignment		outResponseProfile
outStripes		
outAlignedScanPath		

Ridges

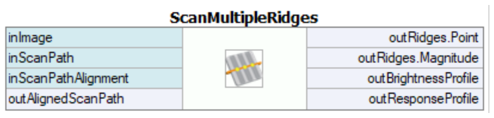
Single Result



ScanSingleRidge

inImage		outRidge.Point
inScanPath		outRidge.Magnitude
inScanPathAlignment		outBrightnessProfile
outAlignedScanPath		outResponseProfile

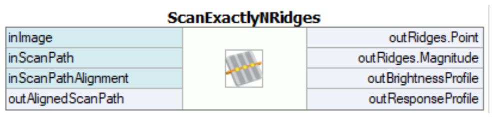
Multiple Results



ScanMultipleRidges

inImage		outRidges.Point
inScanPath		outRidges.Magnitude
inScanPathAlignment		outBrightnessProfile
outAlignedScanPath		outResponseProfile

Fixed Number of Results



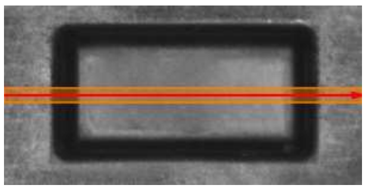
ScanExactlyNRidges

inImage		outRidges.Point
inScanPath		outRidges.Magnitude
inScanPathAlignment		outBrightnessProfile
outAlignedScanPath		outResponseProfile

Note that in Aurora 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

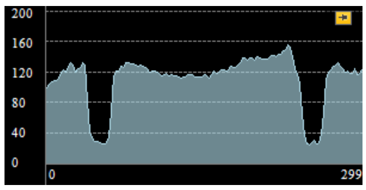
Profile Extraction



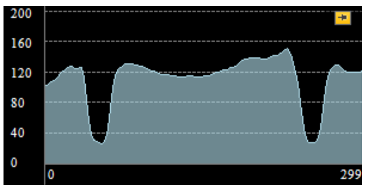
In each of the nine filters the brightness profile is extracted in exactly the same way. The stripe of pixels along **inScanPath** of width **inScanWidth** is traversed and the pixel values across the path are accumulated to form one-dimensional profile. In the picture on the right the stripe of processed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

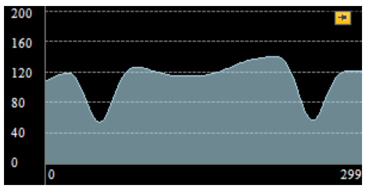
The **inSmoothingStdDev** parameter should be adjusted through interactive experimentation using **outBrightnessProfile** output, as demonstrated below.



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

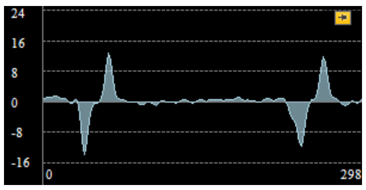


*Appropriate **inSmoothingStdDev** - low noise, significant edges are preserved*



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

Edge Extraction

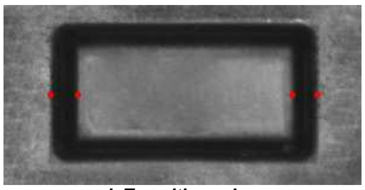


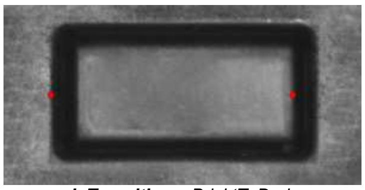
After the brightness profile is extracted and refined, the derivative of the profile is computed and its local extrema of magnitude at least **inMinMagnitude** are identified as edge points. The **inMinMagnitude** parameter should be adjusted using the **outResponseProfile** output.

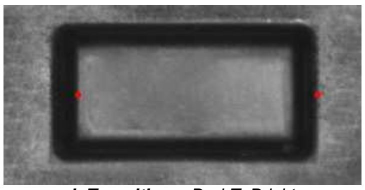
The picture on the right depicts an example **outResponseProfile** profile. In this case the significant extrema vary in magnitude from 11 to 13, while the magnitude of other extrema is lower than 3. Therefore any **inMinMagnitude** value in range (4, 10) would be appropriate.

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*).

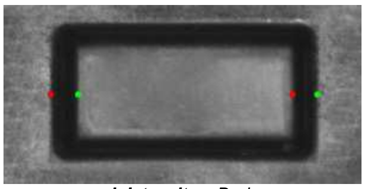

inTransition *= Any*

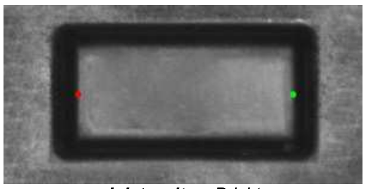

inTransition *= BrightToDark*


inTransition *= DarkToBright*

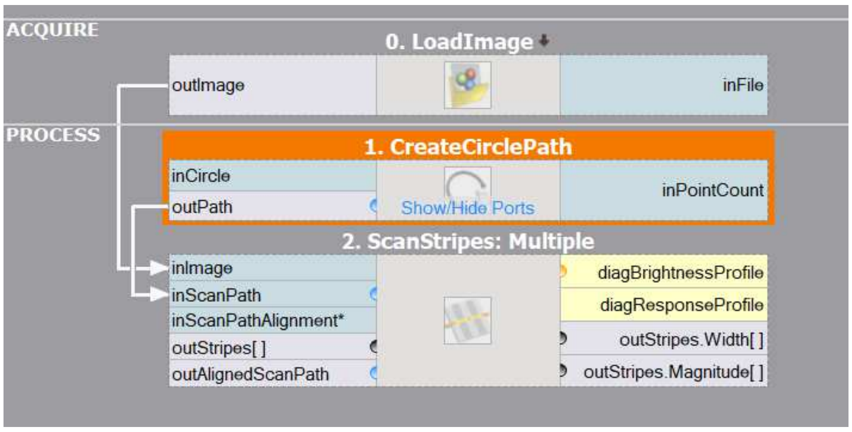
Stripe Intensity

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



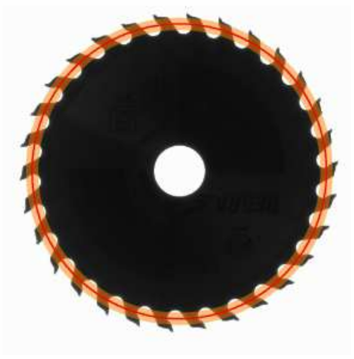
Assume we want to count the blades of a circular saw from the picture.

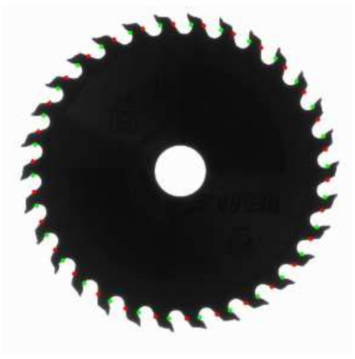
We will solve this problem running a single 1D Edge Detection scan along a circular path intersecting the blades, and therefore we need to produce appropriate circular path. For that we will use a straightforward CreateCirclePath filter. The built-in editor will allow us to point & click the required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





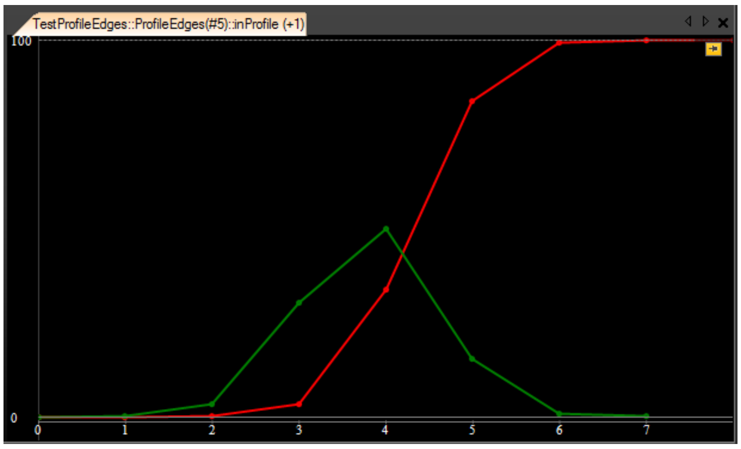
1D Edge Detection – Subpixel Precision

Introduction

One of the key strengths of the 1D Edge Detection tools is their ability do 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

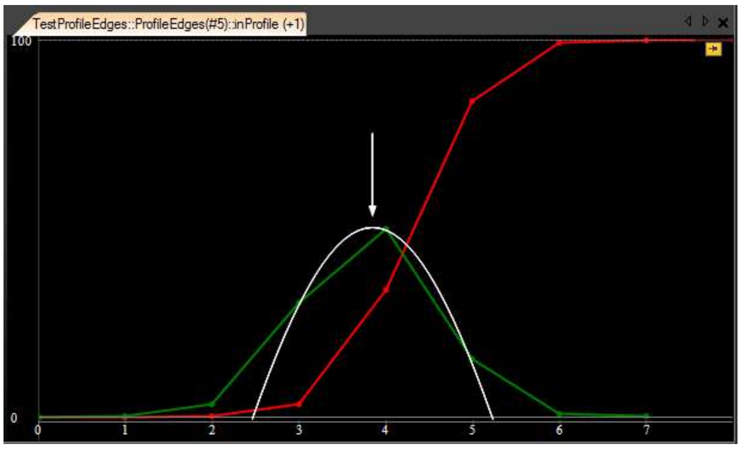
Let us consider a sample profile of pixel values corresponding to an edge (red):



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 Aurora Vision

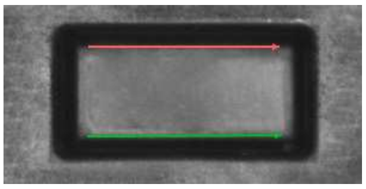
More advanced methods can be used that consider not three, but four consecutive points and which employ additional techniques to assure the highest precision in presence of noise and other practical edge distortions. In Aurora Vision Studio they are available in a form of 3 different profile interpolation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

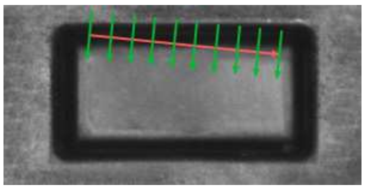
As this technique is derived from 1D Edge Detection, its key advantages are similar – including sub-pixel precision and high performance.

Concept

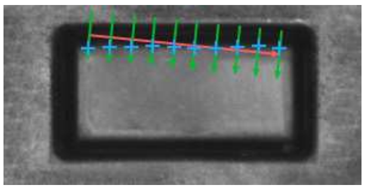
The main idea standing behind Shape Fitting is that a continuous object (such as a circle, an arc or a segment) can be determined using a finite set of points belonging to it. These points are computed by means of appropriate 1D Edge Detection filters and are then combined together into a single higher-level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

Thus, a single Shape Fitting filter's work consists of the following steps:

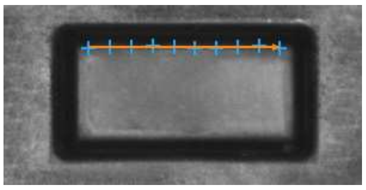
1. **Scan segments preparation** – a series of segments is prepared. The number, length and orientations of the segments are computed from the filter's parameters.
2. **Points extraction** – points that should belong to the object being fitted are extracted using (internally) a proper 1D Edge Detection 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 Geometry 2D Fitting is internally used (e.g. FitCircleToPoints in FitCircleToEdges). An exception to the rule is path fitting. No Geometry 2D Fitting filter is needed there, because the found points serve themselves as the output path characteristic points.



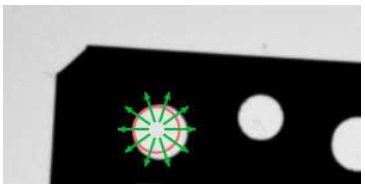
*The scan segments are created according to the fitting field and other parameters (e.g. **inScanCount**).*



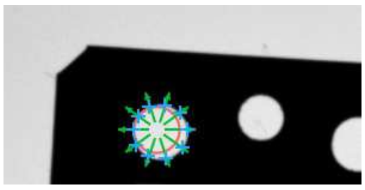
ScanSingleEdge (or another proper 1D Edge Detection filter) is performed.



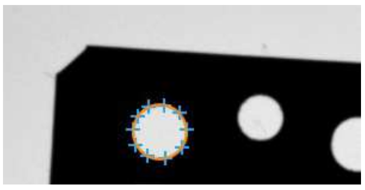
A segment is fitted to the obtained points.



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



ScanSingleEdge (or another proper 1D Edge Detection filter) is performed.



A segment is fitted to the obtained points.

Toolset

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 Shape Fitting 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*.

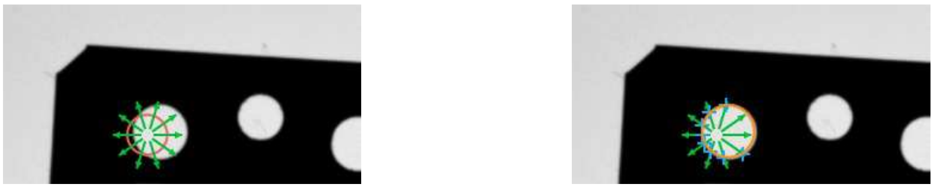
A sample program is shown below:

```
// Precompute CircleFittingMap before loop.
avl::CreateCircleFittingMap
(
 sampleImage,
 CircleFittingField(expectedCircle, 20.0f),
 NIL,
 10,
 1,
 SamplingParams(InterpolationMethod::Bilinear, 1.0f, atl::NIL),
 circleFittingMap
);

while (true)
{
 Image image;
 atl::Conditional<avl::Circle2D> outCircle;

 GetImageFromCamera(image);  // Get images from a camera.

 avl::FitCircleToEdges   // Perform fitting.
 (
  image,
  circleFittingMap,
  EdgeScanParams(),
  Selection::Best,
  NIL,
  0.1f,
  CircleFittingMethod::AlgebraicPratt,
  NIL,
  outCircle
 );

 if (outCircle != NIL)
 {
  // Process results.
 }
}
```

Parameters

Because of the internal use of 1D Edge Detection filters and Geometry 2D Fitting filters, all parameters known from them are also present in Shape Fitting filters interfaces.

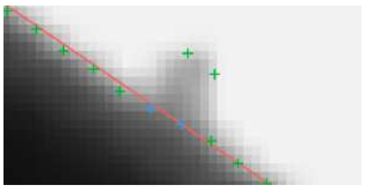
Beside these, there are also a few parameters specific to the subject of shape fitting. The **inScanCount** parameter controls the number of the scan segments. However, not all of the scans have to succeed in order to regard the whole fitting process as being successful. The **inMaxIncompleteness** parameter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

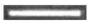
Concept

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.

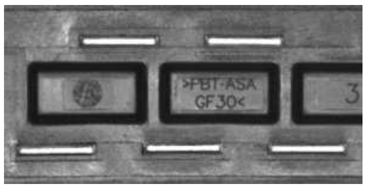
We will start with a demonstration of a naive Template Matching method, which is insufficient for real-life applications, but illustrates the core concept from which the actual Template Matching algorithms stem from. After that we will explain how this method is enhanced and extended in advanced **Grayscale-based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

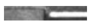
Image Correlation

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

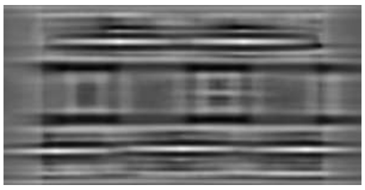
$$\text{NCC}(\text{Image1}, \text{Image2}) = \frac{1}{N\sigma_1\sigma_2} \sum_{x,y} (\text{Image1}(x,y) - \overline{\text{Image1}}) \times (\text{Image2}(x,y) - \overline{\text{Image2}})$$

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

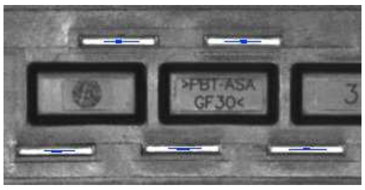
Input image

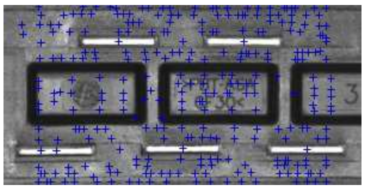

Template correlation image

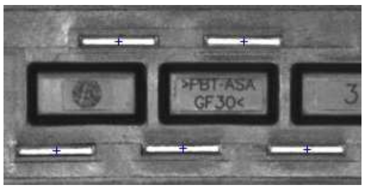
Identification of Matches

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:

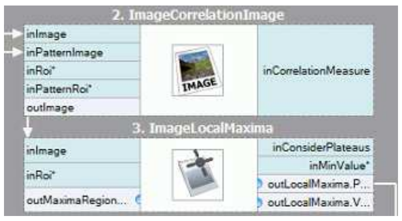
- stronger that some predefined threshold value (i.e stronger that 0.5)
- locally maximal (stronger that the template correlation in the neighboring pixels)


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 Aurora 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 Aurora 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.

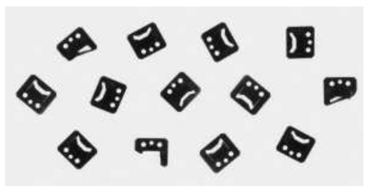
Level 0 (template reference image) *Level 1* *Level 2*

Grayscale-based Matching

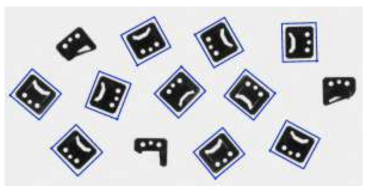
Although in some of the applications the orientation of the objects is uniform and fixed (as we have seen in the plug example), it is often the case that the objects that are to be detected appear rotated. In Template Matching algorithms the classic pyramid search is adapted to allow *multi-angle* matching,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

Template image



Input image



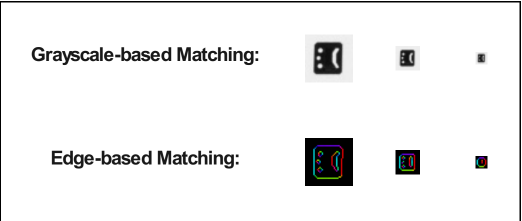
Results of multi-angle matching

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.



Different kinds of template pyramids used in Template Matching algorithms.

Filter Toolset

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

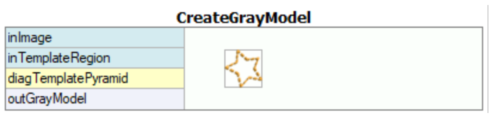
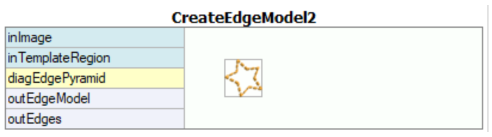
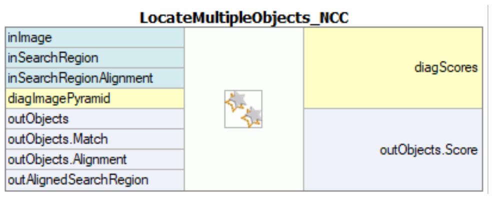
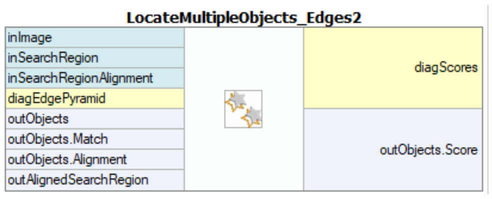
As the template image has to be preprocessed before the pyramid matching (we need to calculate the template image pyramids for all possible rotations and scales), the algorithms are split into two parts:

- **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.

	Grayscale-based Matching	Edge-based Matching
Model Creation:	 **CreateGrayModel** inImage inTemplateRegion diagTemplatePyramid outGrayModel	 **CreateEdgeModel2** inImage inTemplateRegion diagEdgePyramid outEdgeModel outEdges
Matching:	 **LocateMultipleObjects_NCC** inImage inSearchRegion inSearchRegionAlignment diagImagePyramid outObjects outObjects.Match outObjects.Alignment outAlignedSearchRegion diagScores outObjects.Score	 **LocateMultipleObjects_Edges2** inImage inSearchRegion inSearchRegionAlignment diagEdgePyramid outObjects outObjects.Match outObjects.Alignment outAlignedSearchRegion diagScores outObjects.Score

Please note that the use of CreateGrayModel and CreateEdgeModel2 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. The CreateEdgeModel2 and LocateMultipleObjects_Edges2 filters are preferred over CreateEdgeModel1 and LocateMultipleObjects_Edges1 because they are newer, more advanced versions with more capabilities.

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

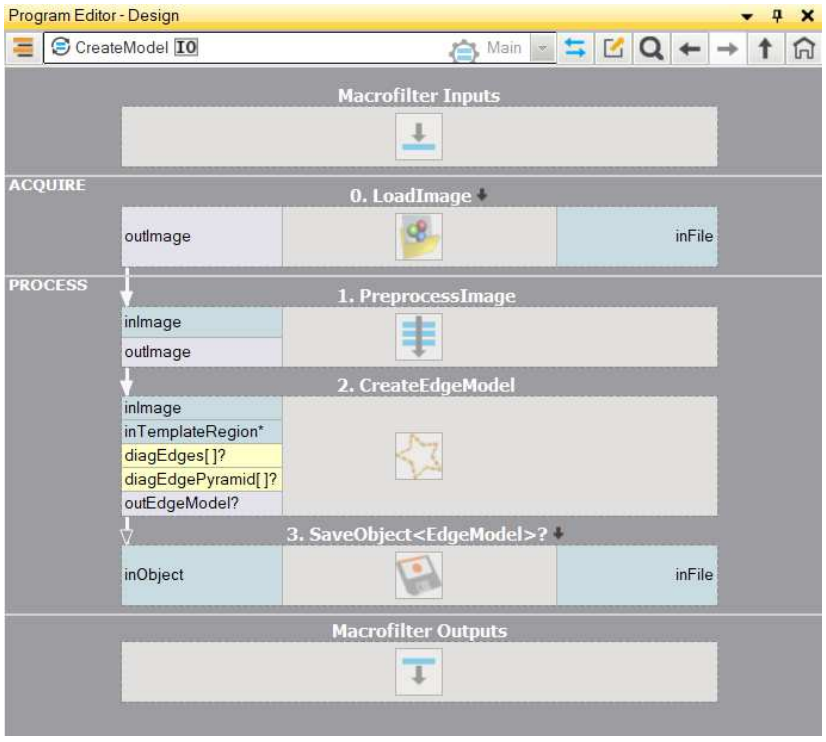
There are several kinds of advanced applications, for which the interactive GUI for Template Matching is not enough and the user needs to use the CreateGrayModel or CreateEdgeModel2 filter directly. For example:

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).

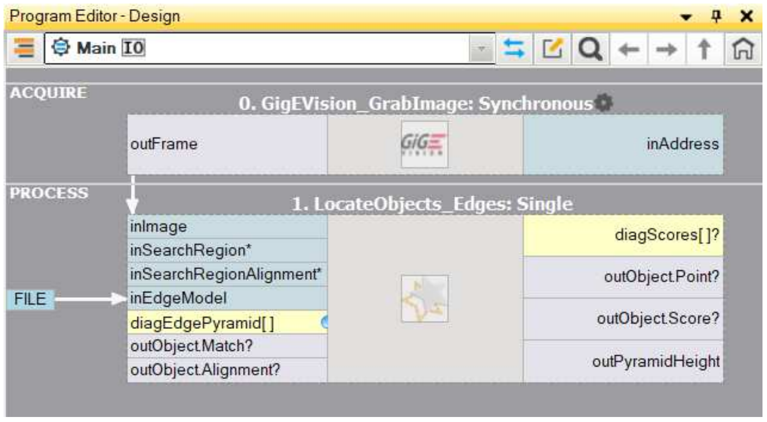
Schema 1: Model Creation in a Separate Program

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:



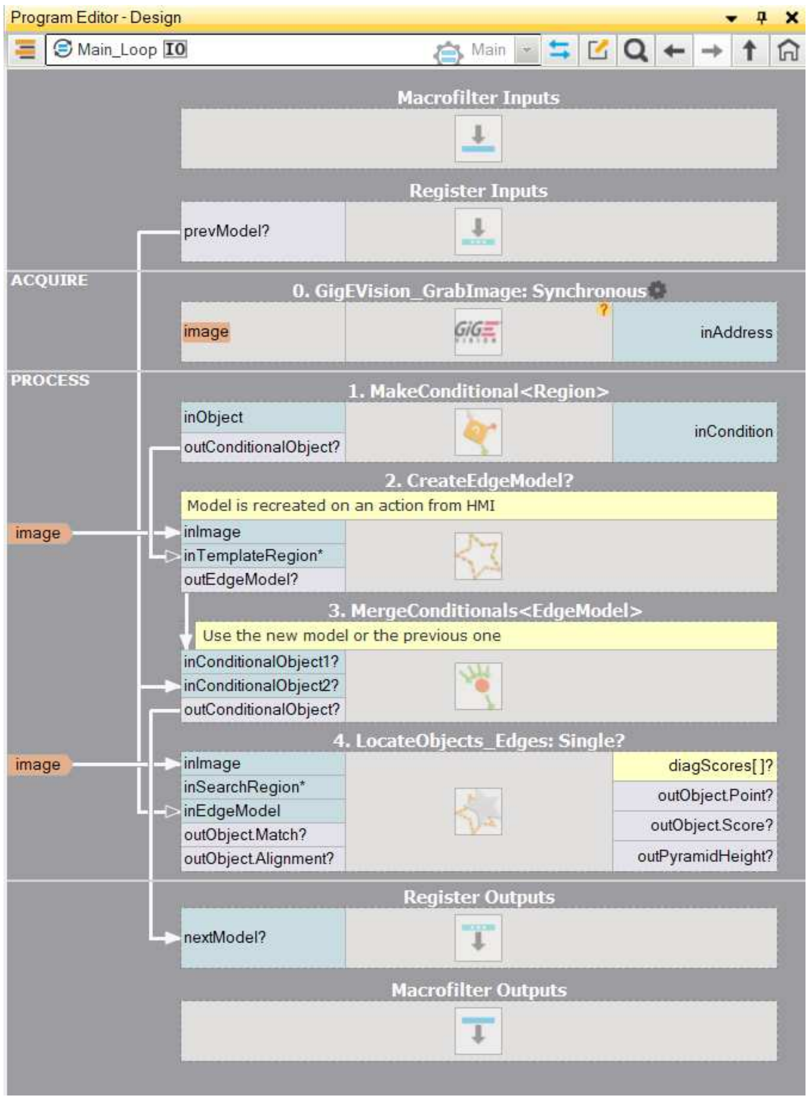
Main Program:



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 EdgeModel2? type should be used, that will store the latest model. 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:



prevModel?

0. GigEVision_GrabImage: Synchronous
image | inAddress

1. MakeConditional<Region>
inObject | inCondition
outConditionalObject?

2. CreateEdgeModel?
Model is recreated on an action from HMI
inImage
inTemplateRegion*
outEdgeModel?

3. MergeConditionals<EdgeModel>
Use the new model or the previous one
inConditionalObject1?
inConditionalObject2?
outConditionalObject?

4. LocateObjects_Edges: Single?
inImage | diagScores[ ]?
inSearchRegion* | outObject.Point?
inEdgeModel | outObject.Score?
outObject.Match? | outPyramidHeight?
outObject.Alignment?

nextModel?

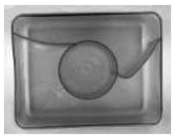
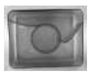
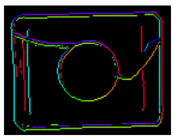
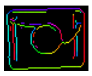
Model Creation

Height of the Pyramid

The **inMaxPyramidLevel** 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 **diagTemplatePyramid** (Grayscale-based Matching) or **diagEdgePyramid** (Edge-based Matching).

The **inMinPyramidLevel** parameter determines the lowest pyramid level that is generated during creation phase and the lowest pyramid level that the occurrences are tracked to during location phase. If the parameter is set to lower value in location than in creation, the missing levels are generated dynamically by the locating filter. This approach leads to much faster creation, but a bit slower location.

In the following example the **inMaxPyramidLevel** 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 seems a bit excessive (especially in case of Edge-based Matching) while the value of 2 would definitely be a safe choice.

	Level 0	**Level 1**	**Level 2**	**Level 3**	**Level 4**
Grayscale-based Matching (diagTemplatePyramid):					
Edge-based Matching (diagEdgePyramid):					

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):

- (-180.0, 180.0): 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, especially if different scales are also involved. The number of rotations created can be further manipulated with **inAnglePrecision** parameter. Decreasing it results in smaller models and smaller execution times, but can also lead to objects that are slightly less accurate.

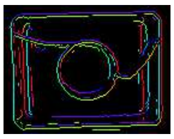
Scale Range

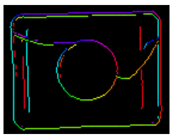
The **inMinScale**, **inMaxScale** parameters determine the range of template scales that will be considered in the matching process. It enables locating objects that are slightly smaller or bigger than the object used during model creation.

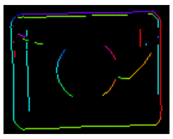
Wide range of possible scales introduces significant amount of overhead (both in memory usage and computing time), so it is advisable to limit the range whenever possible. The number of scales created can be further manipulated with **inScalePrecision** parameter. Decreasing it results in smaller models and smaller execution times, but can also lead to objects that are slightly less accurate.

Edge Detection Settings (only Edge-based Matching)

The **inEdgeThreshold**, **inEdgeHysteresis** parameters of CreateEdgeModel2 filter determine the settings of the hysteresis thresholding used to detect edges in the template image. The lower the **inEdgeThreshold** 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 **outEdges** output and 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 CreateEdgeModel2 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.

How to even further upgrade the results of Edge-based Matching?

You can use EnhanceMultipleObjectMatches filter or EnhanceSingleObjectMatch filter to fine-tune the results. A great example of usage is presented in the CreateGoldenTemplate2 filter.

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:

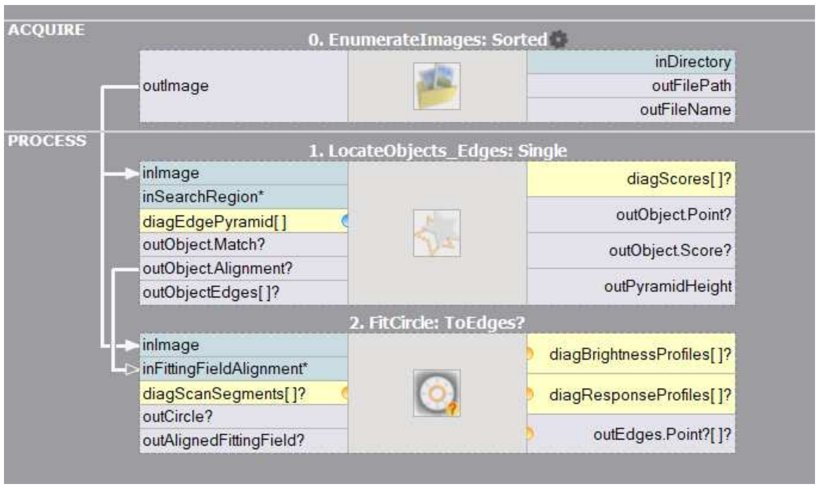
1. With Template Matching filters it is straightforward as the filters have **outObjectAlignment(s)** outputs, which provide local coordinate systems of the detected objects.
2. With one of the CreateCoordinateSystem functions, which allow for creating local coordinate systems manually at any location, 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 Aurora 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.

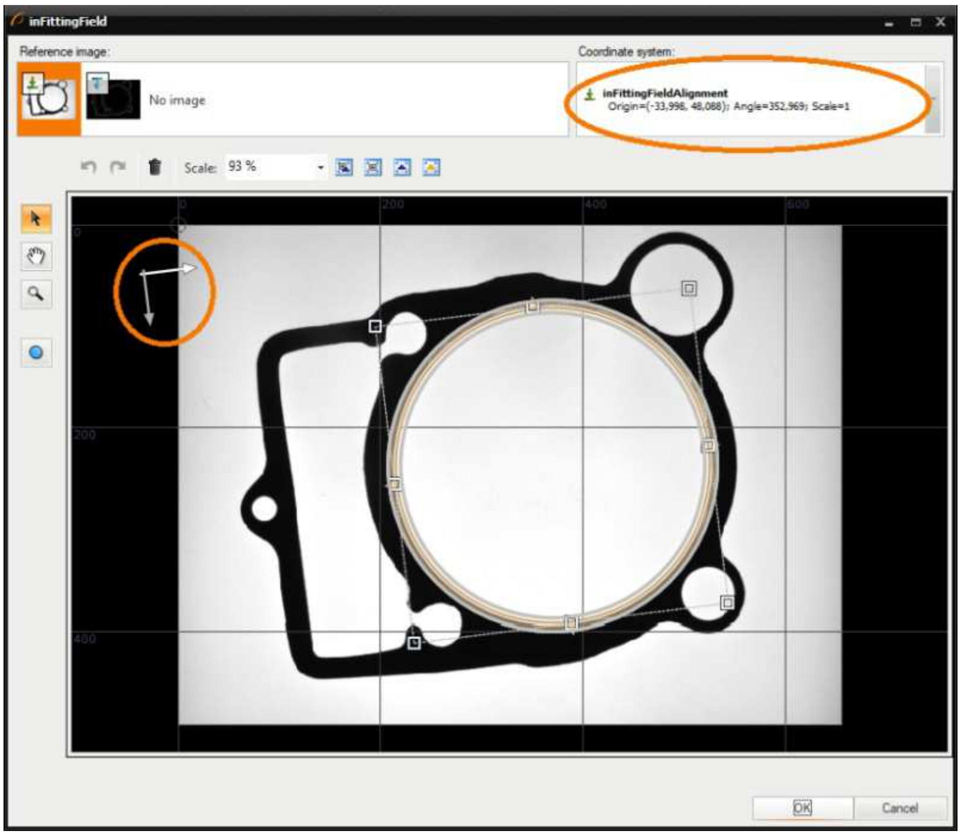
Example 1: Alignment from Template Matching

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



Template Matching and an aligned circle fitting.

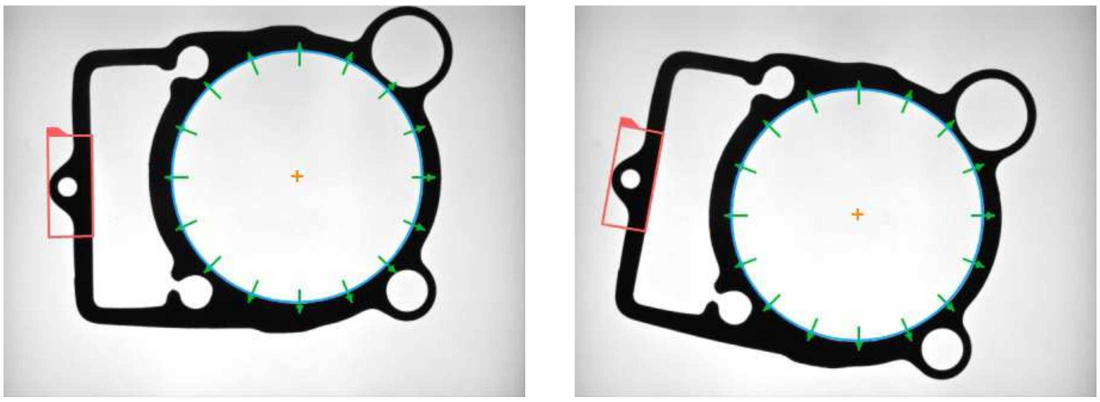
When you execute the template matching filter and enter the editor of the **inFittingField** input of the FitCircleToEdges filter, you will have the local coordinate system already selected (you can also select it manually) and the primitive you create will have relative coordinates:



inFittingFieldAlignment
Origin=(-33,998, 48,088); Angle=352,969; Scale=1

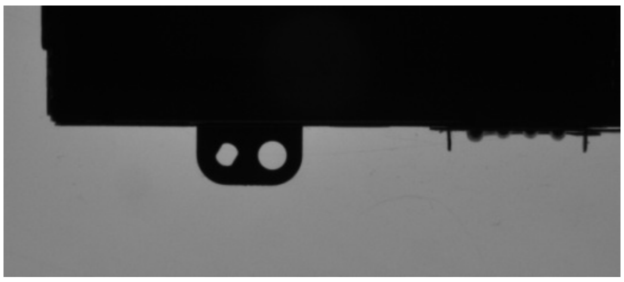
Editing an expected circle in a local coordinate system.

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:



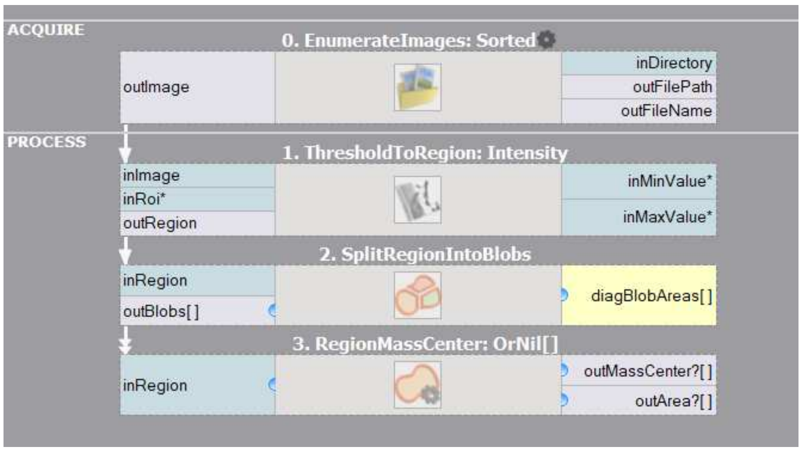
Example 2: Alignment from Blob Analysis

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:

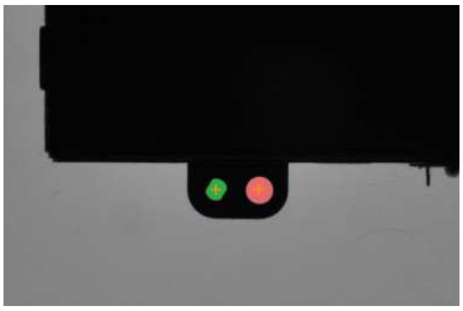


Two holes clearly define the object location.

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

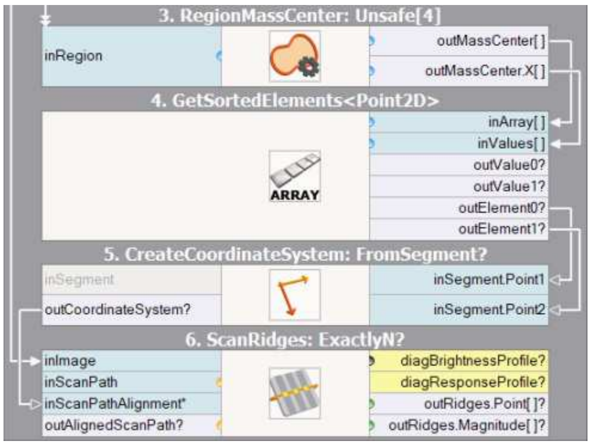


Filters detecting blobs and their centers.

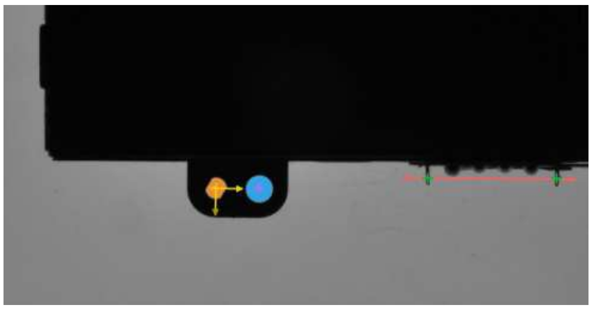


The result of blob detection.

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

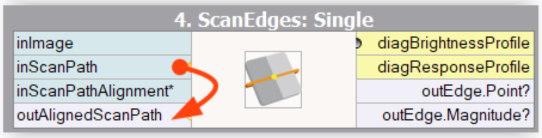
In some cases the filter you will need to use with a local coordinate system will have no appropriate **inAlignment** input. In such cases the solution is to transform the primitive manually with filters like AlignPoint, AlignCircle, AlignRectangle. These filters accept a geometrical primitive defined 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 Aurora 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 filters 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 Aurora 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.

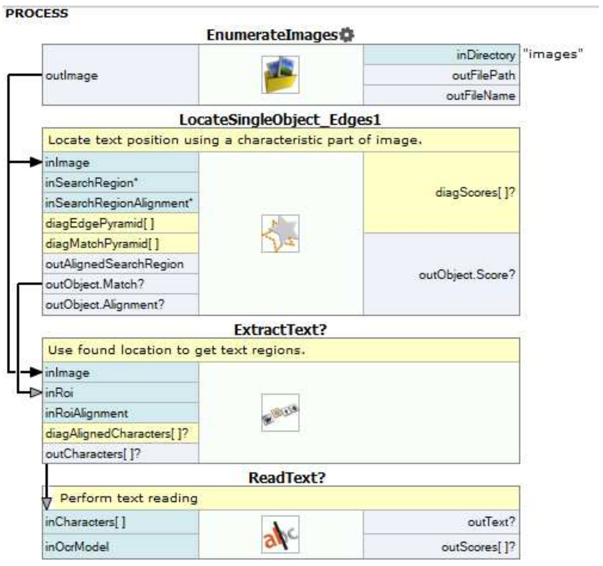
The last part presents an example program that reads text from images.

Using high level Optical Character Recognition filters

Aurora 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.



Locate text position using a characteristic part of image.

Use found location to get text regions.

Perform text reading

Example OCR application using high level filters.

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:

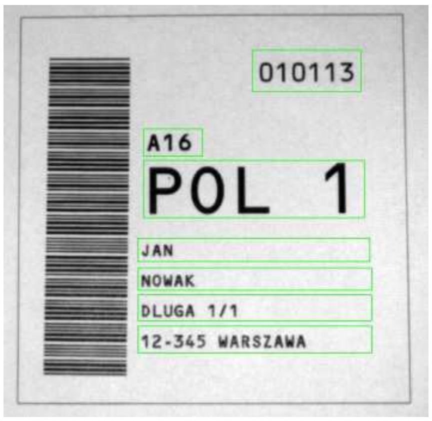
1. Getting text location,
2. Extracting text from the background,
3. Segmenting text,
4. Using prepared OCR models,
5. Character recognition,
6. Interpreting results,
7. Verifying results.

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

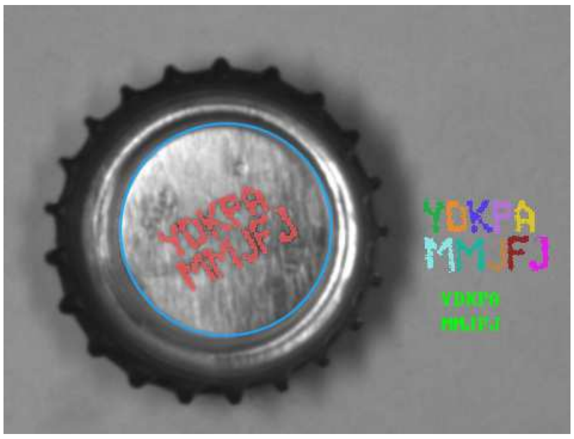
In general, text localization tasks can be divided into three cases:

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.

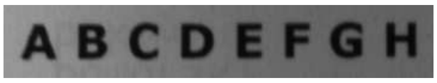


Getting text from a bottle cap.

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.

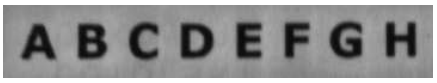
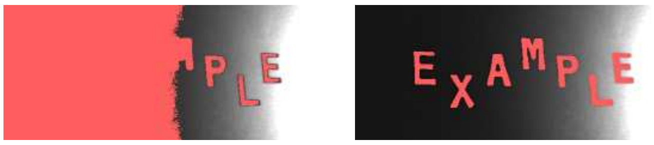


Image after light normalization.

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.



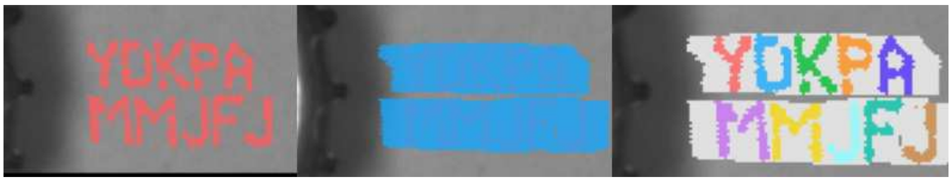
Results of ThresholdToRegion and ThresholdToRegion_Dynamic on images with uneven light.

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

Segmenting text

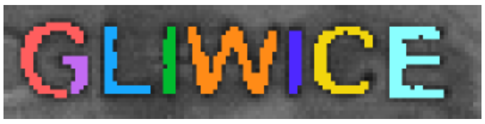
Text region segmentation is a process of splitting a region into lines and individual characters. The recognition step is only possible if each region contains a single character.

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.

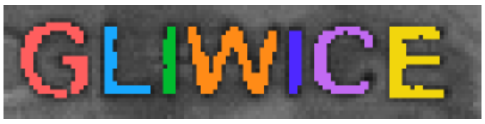


The process of splitting text into lines using region morphology filters.

When text 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\Aurora Vision\{Aurora Vision Product Name}\PretrainedFonts*.

The table below shows the list of available font models:

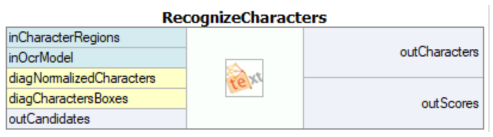
Font name	Font typeface	Set name	Characters
OCRA	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ.-/
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+
OCRB	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ.-/
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+
MICR	monospaced	ABC09	ABC0123456789
Computer	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ.-/
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+
DotMatrix	monospaced	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ+-./
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ+-01234556789./
		09	01234556789.+-/
Regular	proportional	AZ	ABCDEFGHIJKLMNOPQRSTUVWXYZ.-/
		AZ_small	abcdefghijklmnopqrstuvwxyz.-/
		09	0123456789.-/+
		AZ09	ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.-/+

Character recognition

Aurora Vision Library offers two types of character classifiers:

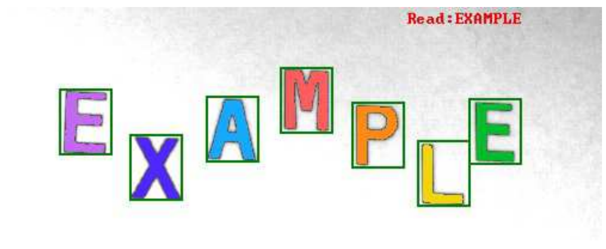
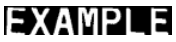
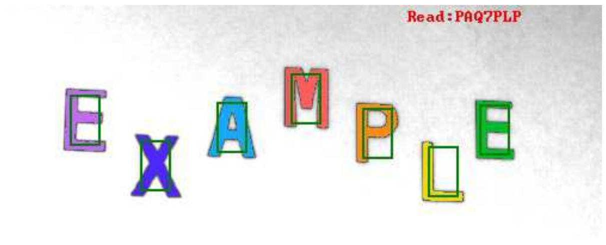
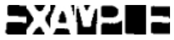
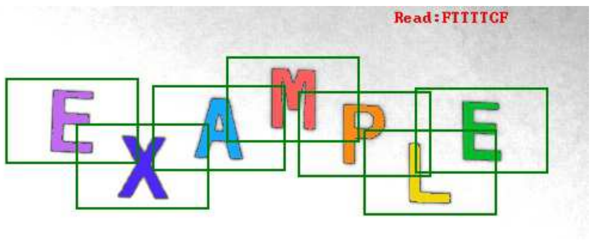
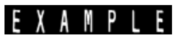
1. Classifier based on multi-layer perceptron (MLP).
2. Classifier based on support vector machines (SVM).

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)	
	(outCharacters)	(outScores)	(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

After reading result should be check if text follows constraints. It can be done using simple string manipulation.

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:

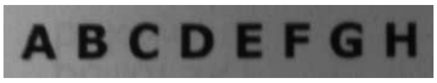
1. preparation of the training data set,
2. selection of the normalization size and character features,
3. setup of the OCR model,
4. training of the OCR model,
5. saving results to a file.

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.


Character samples acquired from a real usage.

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.

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.

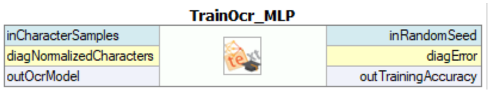
Various size characters before and after the normalization process.

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.

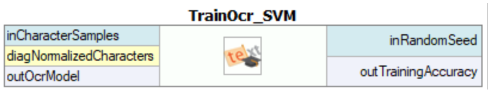
During the normalization process some character details will be lost, e.g. the aspect ratio of a character. In the training process, some additional information can be added, which can compensate for the information loss in the normalization process. For further information please refer to the documentation 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 function SaveOcrModel should be used.

Camera Calibration and World Coordinates

Camera Calibration

Camera calibration, also known as camera resectioning, is a process of estimating parameters of a camera model: a set of parameters that describe the internal geometry of image capture process. Accurate camera calibration is is essential for various applications, such as multi-camera setups where images relate to each other, removing geometric distortions due to lens imperfections, or precise measurement of real-world geometric properties (positions, distances, areas, straightness, etc.).

The model to be used is chosen depending on the camera type (e.g. projective camera, telecentric camera, line scan camera) and accuracy requirements. In a case of a standard projective camera, the model (known as pinhole camera model) consists of focal length, principal point location and distortion parameters.

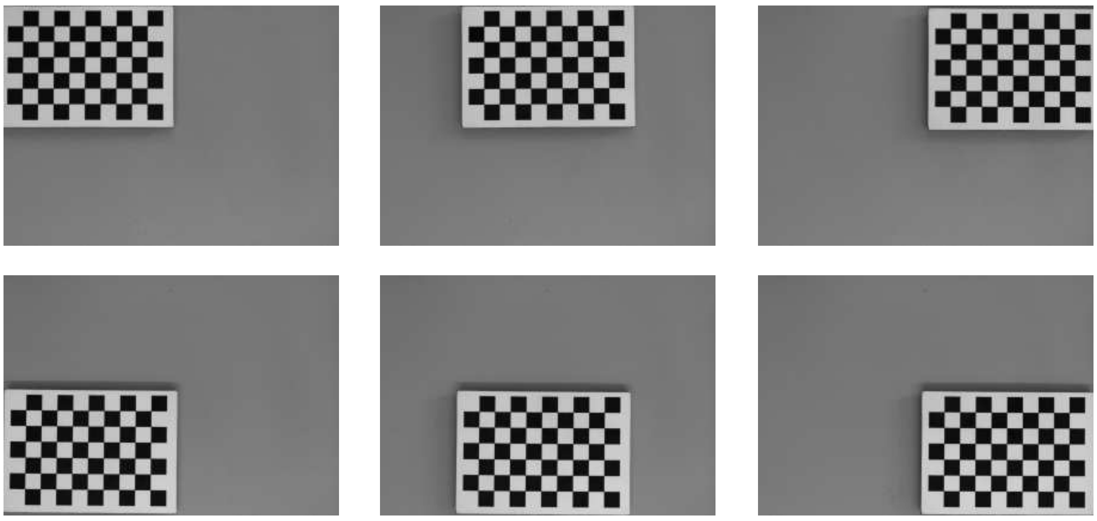
A few distortion model types are supported. The simplest - divisional - supports most use cases and has predictable behaviour even when calibration data is sparse. Higher order models can be more accurate, however they need a much larger dataset of high quality calibration points, and are usually needed for achieving high levels of positional accuracy across the whole image - order of magnitude below 0.1 pix. Of course this is only a rule of thumb, as each lens is different and there are exceptions.

The area scan camera models (pinhole or telecentric) contain only *intrinsic* camera parameters, and so it does not change with camera repositioning, rotations, etc. Thanks to that, there is no need for camera calibration in the production environment, the camera can be calibrated beforehand. As soon as the camera has been assembled with the lens and lens adjustments (zoom/focus/f-stop rings) have been tightly locked, the calibration images can be taken and camera calibration performed. Of course any modifications to the camera-lens setup void the calibration parameters, even apparently minor ones such as removing the lens and putting it back on the camera in seemingly the same position.

On the other hand the line scan model contains parameters of whole imaging setup, i.e. camera and a moving element (usually a conveyor belt). Such approach, in contrast with area scan camera calibration, is necessary as the moving element of line scan camera system is tightly bound within the image acquisition geometry.

Camera model can be directly used to obtain an *undistorted* image (an image, which would have been taken by a camera with the same basic parameters, but without lens distortion present), however for most use cases the camera calibration is just a prerequisite to some other operation. For example, when camera is used for inspection of planar surfaces (or objects lying on such surface), the camera model is needed to perform a *World Plane* calibration (see *World Plane - measurements and rectification* section below).

In Aurora Vision Studio user will be prompted by a GUI when a camera calibration is needed to be performed. Alternatively, filters responsible for camera calibration may be used directly: CalibrateCamera_Pinhole, CalibrateCamera_Telecentric, CalibrateCamera_LineScan.



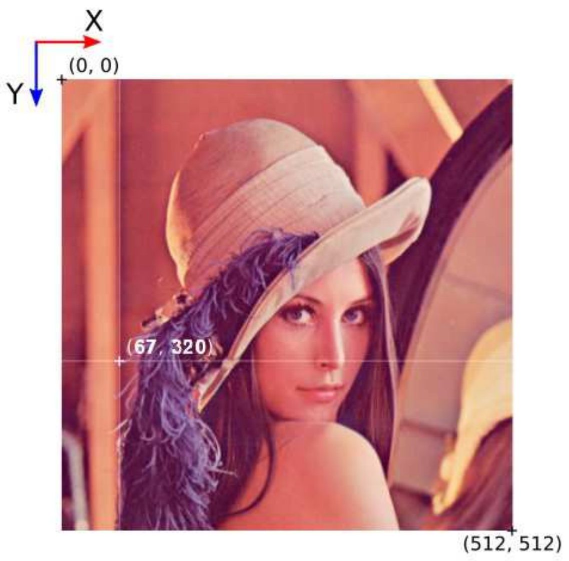
A set of grid pictures for basic calibration. Note that high accuracy applications require denser grids and higher amount of pictures. Also note that all grids are perpendicular to the optical axis of the camera, so the focal length won't be calculated by the filter.

World Plane - Measurements and Rectification

Vision systems which are concerned with observation and inspection of planar (flat) surfaces, or objects lying on such surfaces (e.g. conveyor belts) can take advantage of the *image to world plane transform* mechanism of Aurora Vision Studio, which allows for:

- Calculation of real world coordinates from locations on original image. This is crucial, for example, for interoperability with external devices, such as industrial robots. Suppose a object is detected on the image, and its location needs to be transmitted to the robot. The detected object location is given in image coordinates, however the robot is operating in real world with different coordinate system. A common coordinate system is needed, defined by a *world plane*.
- Image *rectification* onto the *world plane*. This is needed when performing image analysis using original image is not feasible (due to high degree of lens and/or perspective distortion). The results of analysis performed on a rectified image can also be transformed to real-world coordinates defined by a world plane coordinate system. Another use case is a multi-camera system – *rectification* of images from all the cameras onto common *world plane* gives a simple and well defined relation between those rectified images, which allows for easy superimposing or mosaic stitching.

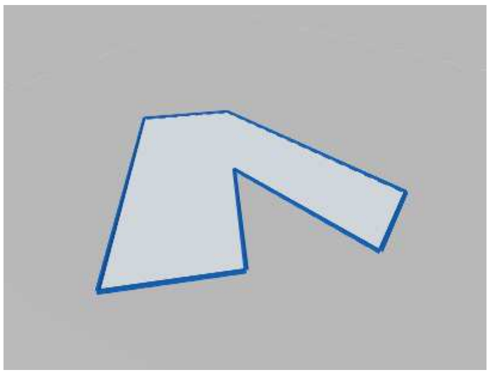
The image below shows the image coordinate system. Image coordinates are denoted in pixels, with the origin point (0, 0) corresponding to the top-left corner of the image. The X axis starts at the left edge of an image and goes towards the right edge. The Y axis starts at the top of the image towards image bottom. All image pixels have nonnegative coordinates.



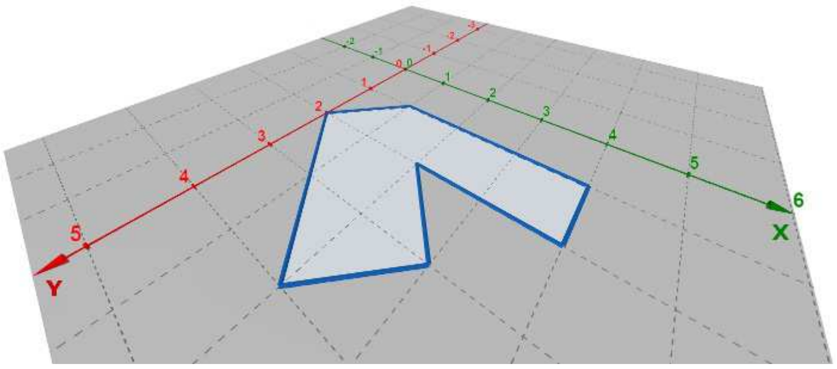
Directions and pixel positions in image coordinates.

The *world plane* is a distinguished flat surface, defined in the real 3D world. It may be arbitrarily placed with respect to the camera. It has a defined origin position and XY axes.

Images below present the concept of a *world plane*. First image presents an original image, as captured by a camera that has not been mounted quite straight above the object of interest. The second image presents a *world plane*, which has been aligned with the surface on which the object is present. This allows for either calculation of world coordinates from pixel locations on original image, or image rectification, as shown on the next images.



Object of interest as captured by an imperfectly positioned camera.



World plane coordinate system superimposed onto the original image.

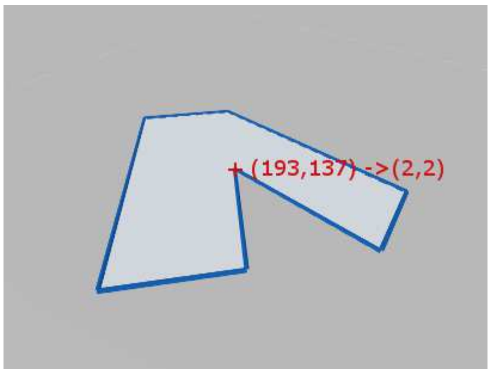


Image to world plane coordinate calculation.

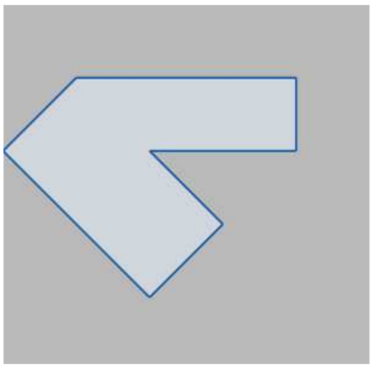


Image rectification, with cropping to an area from point (0,0) to (5,5) in world coordinates.

In order to use the *image to world plane transform* mechanism of Aurora Vision Studio, appropriate UI wizards are supplied:

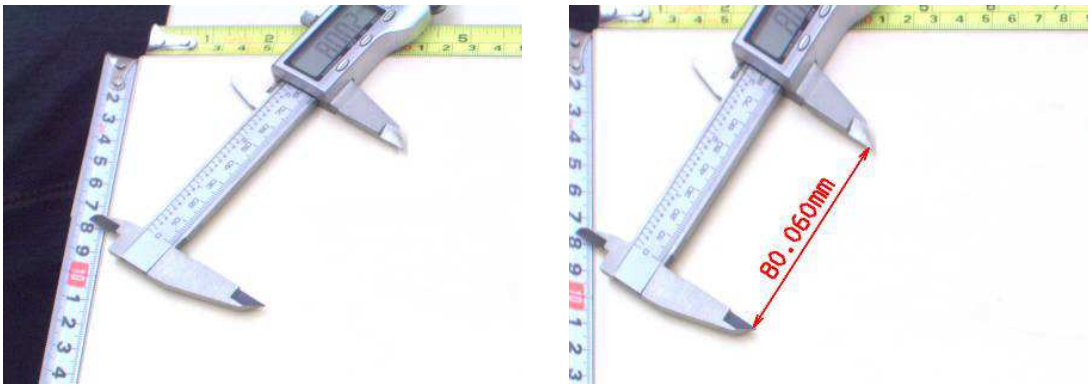
- For calculation of real world coordinates from locations on original image – use a wizard associated with the **inTransform** input of ImagePointToWorldPlane filter (or other from ImageObjectsToWorldPlane group).
- For image *rectification* onto the *world plane* – use a wizard associated with the **inRectificationMap** input of RectifyImage filter.

Although using UI wizards is the recommended course of action, the most complicated use cases may need a direct use of filters, in such a case following steps are to be performed:

1. Camera calibration – this step is highly recommended to achieve accurate results, although not strictly necessary (e.g. when lens distortion errors are insignificant).
2. World plane calibration – the CalibrateWorldPlane_* filters compute a RectificationTransform, which represents *image to world plane relation*
3. The *image to world plane relation* then can be used to:
   - Calculate of real world coordinates from locations on original image, and vice versa, see ImagePointToWorldPlane, WorldPlanePointToImage or similar filters (from ImageObjectsToWorldPlane or WorldPlaneObjectsToImage groups).
   - Perform image *rectification* onto the world plane, see CreateRectificationMap_* filters.

There are different use cases of world coordinates calculation and image rectification:

- Calculating world coordinates from pixel locations on original image without image rectification. This approach uses transformation output for example by CalibrateWorldPlane_* to calculate real world coordinates with ImageObjectsToWorldPlane_*
- Second scenario is very similar to the first one with the difference of using image rectification. In this case, after performing analysis on an rectified image (i.e. image remapped by RectifyImage), the locations can be transformed to a common coordinate system given by the world plane by using the *rectified image to world plane* relation. It is given by auxiliary output **outRectifiedTransform** of RectifyImage filter. Notice that the *rectified image to world plane* relation is different than *original image to world plane* relation.
- Last use case is to perform image rectification and rectified image analysis without its features recalculation to real world coordinates.



Example of taking world plane measurements on the rectified image. Left: original image, as captured by a camera, with mild lens distortion. Right: rectified image with annotated length measurement.

Notes:

- *Image to world plane transform* is still a valid mechanism for telecentric cameras. Is such a case, the image would be related to world plane by an affine transform.
- Camera distortion is automatically accounted for in both world coordinate calculations and image rectification.
- The spatial map generated by CreateRectificationMap_* filters can be thought of as a map performing *image undistortion* followed by a *perspective removal*.

Extraction of Calibration Grids

Both *camera calibration* and *image to world plane transform* calculation use extracted *calibration grids* in the form of array of image points with grid indices, i.e. annotated points.

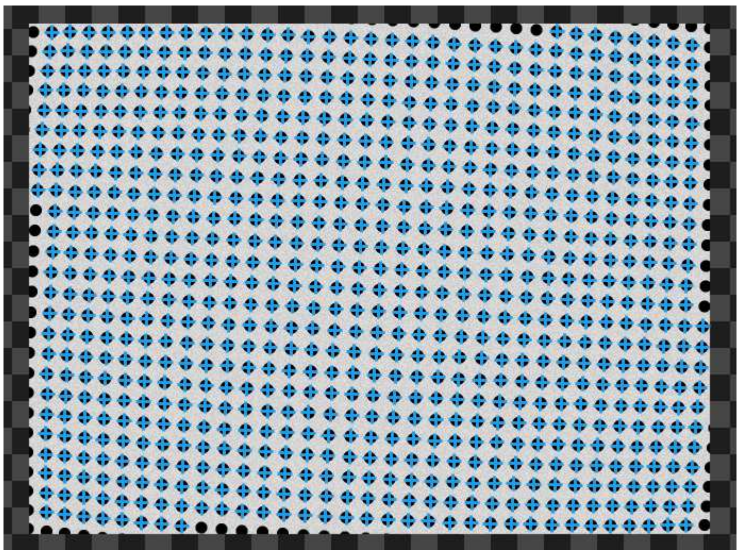
Note that the real-world coordinates of the grids are 2D, because the relative `z` coordinate of any point on the flat grid is `0`.

Aurora Vision Studio provides extraction filters for a few standard grid formats (see: DetectCalibrationGrid_Chessboard and DetectCalibrationGrid_Circles).

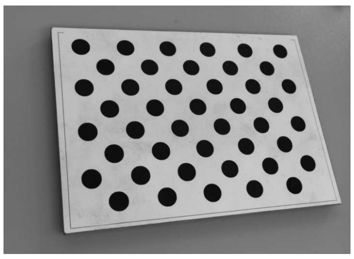
Using custom grids requires a custom solution for extracting the image point array. If the custom grid is a rectangular grid, the AnnotateGridPoints filter may be used to compute annotations for the image points.

Note that the most important factor in achieving high accuracy results is the precision and accuracy of extracted calibration points. The calibration grids should be as flat and stiff as possible (cardboard is not a proper backing material, thick glass is perfect). Take care of proper conditions when taking the calibration images: minimize motion blur by proper camera and grid mounts, prevent reflections from the calibration surface (ideally use diffusion lighting). When using a custom calibration grid, make sure that the points extractor can achieve subpixel precision. Verify that measurements of the real-world grid coordinates are accurate. Also, when using a chessboard calibration grid, make sure that the whole calibration grid is visible in the image. Otherwise, it will not be detected because the detection algorithm requires a few pixels wide quiet zone around the chessboard. Pay attention to the number of columns and rows, as providing misleading data may make the algorithm work incorrectly or not work at all.

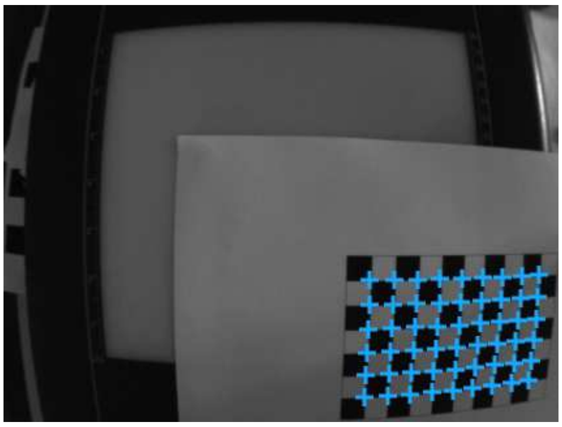
The recommended calibration grid to use in Aurora Vision Studio is a circles grid, see DetectCalibrationGrid_Circles. Optimal circle radius may vary depending on exact conditions, however a good rule of thumb is 10 pixels (20 pixel diameter). Smaller circles tend to introduce positioning jitter. Bigger circles lower the total amount of calibration points and suffer from geometric inaccuracies, especially when lens distortion and/or perspective is noticeable. Note: it is important to use a symmetric board as shown in the image below. Asymmetric boards are currently not supported.



Symmetric circle grid is the recommended one to use in Aurora Vision Studio.



Unsupported asymmetric circle grid.



Detected chessboard grid, with image point array marked.

Application Guide – Image Stitching

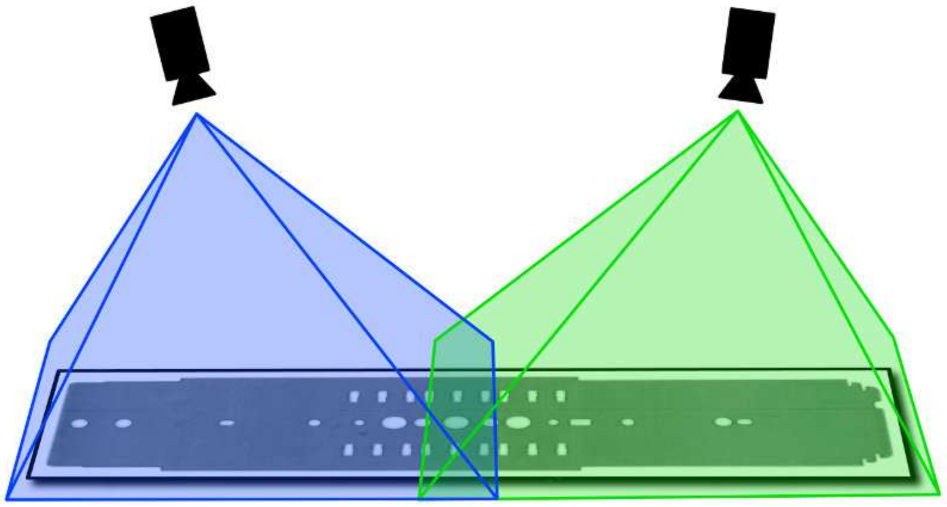
Seamless image stitching in multiple camera setup is, in its essence, an image rectification onto the world plane.

Note that high quality stitching requires a vigilant approach to the calibration process. Each camera introduces both lens distortion as well as perspective distortion, as it is never positioned perfectly perpendicular to the analyzed surface. Other factors that need to be taken into account are the camera-object distance, camera rotation around the optical axis, and image overlap between cameras.

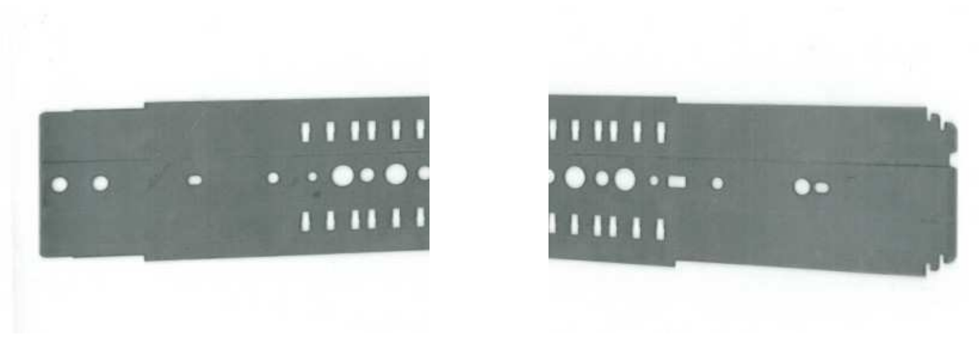
The process consists of two main steps. First, each camera is calibrated to produce a partial, rectified image. Then all partial images are simply merged using the JoinImages filter.

Image stitching procedure can be outlined as follows:

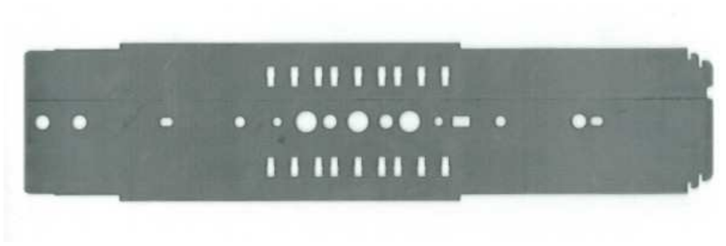
- Cover the inspection area with two or more cameras. Make sure that fields of view of individual cameras overlap a bit.
- Place a calibration grid onto the inspection area. For each camera, capture the image of a part of the calibration grid. The grid defines a world coordinate system used for stitching, and so it should contain some markers from which the coordinates of world plane points will be identifiable for each camera.
- Define the world coordinate extents for which each camera will be responsible. For example, lets define that camera 1 should cover area from 100 to 200 in X, and from -100 to 100 in Y coordinate; camera 2 - from 200 to 300 in X, and from -100 to 100 in Y.
- For each camera, use a wizard associated with the **inRectificationMap** input of RectifyImage filter to setup the image rectification. Use the captured image for camera calibration and world to image transform. Use the defined world coordinate extents to setup the rectification map generation (select "world bounding box" mode of operation). Make sure that the world scale for rectification is set to the same fixed value for all images.
- Use the JoinImages appropriately to merge outputs of RectifyImage filters.



A multi-camera setup for inspection of a flat object.



Input images, as captured by cameras.



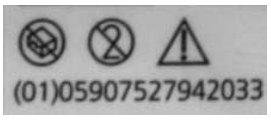
Stitching result.

Golden Template

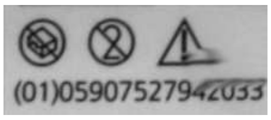
Golden Template technique performs a pixel-to-pixel comparison of two images. This technique is especially useful when the object's surface or object's shape is very complex.

Aurora Vision Studio offers three ways of performing the golden template comparison.

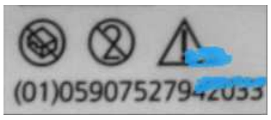
- **Comparison based on pixels intensity** - it can be achieved using the CompareGoldenTemplate_Intensity. In this method two images are compared pixel-by-pixel and the defect is classified based on a difference between pixels intensity. This technique is especially useful in finding defects like smudges,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



Golden template



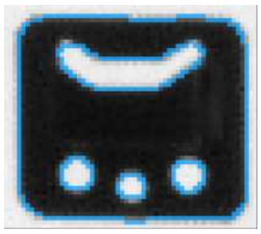
Defected object



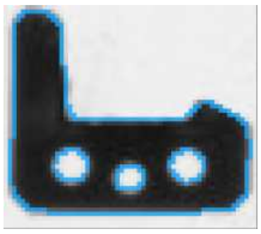
Found defects

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

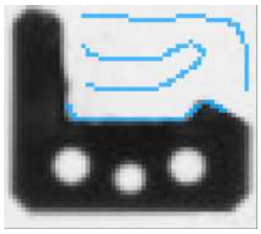
- **Comparison based on objects edges** - this method is very useful when defects may occur on the edge of the object and pixel comparison may fail due to different light reflections or the checking the object surface is not necessary. For matching object's edges use the CompareGoldenTemplate_Edges filter.



Golden template



Defected object



Found defects

Example usage of Golden Template technique using the edges comparison.

- **Second version of the comparison based on objects edges** - this method uses more than one image to create the model for the inspection. Due to that it is not vulnerable to pixel-sized errors and displacements. Advanced tips on how to use its parameters are located here: CompareGoldenTemplate2.

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

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.

Deep Learning

Table of contents:

1. Introduction

Deep Learning is a breakthrough machine learning technique in computer vision. It learns from training images provided by the user and can automatically generate solutions for a wide range of image analysis applications. Its key advantage, however, is that it is able to solve many of the applications which have been too difficult for traditional, rule-based algorithms of the past. Most notably, these include inspections of objects with high variability of shape or appearance, such organic products, highly textured surfaces or natural outdoor scenes. What is more, when using ready-made products, such as our Aurora Vision Deep Learning, the required programming effort is reduced almost to zero. On the other hand, deep learning is shifting the focus to working with data, taking care of high quality image annotations and experimenting with training parameters – these elements actually tend to take most of the application development time these days.

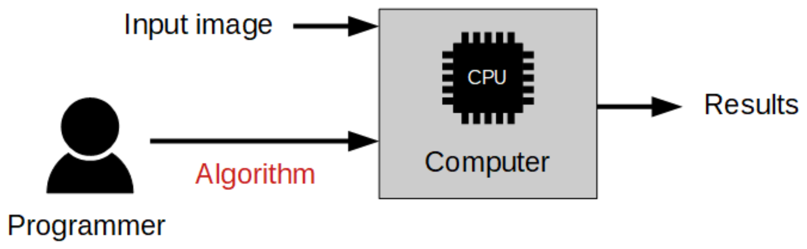
Typical applications are:

- detection of surface and shape defects (e.g. cracks, deformations, discoloration),
- detecting unusual or unexpected samples (e.g. missing, broken or low-quality parts),
- identification of objects or images with respect to predefined classes (i.e. sorting machines),
- location, segmentation and classification of multiple objects within an image (i.e. bin picking),
- product quality analysis (including fruits, plants, wood and other organic products),
- location and classification of key points, characteristic regions and small objects,
- optical character recognition.

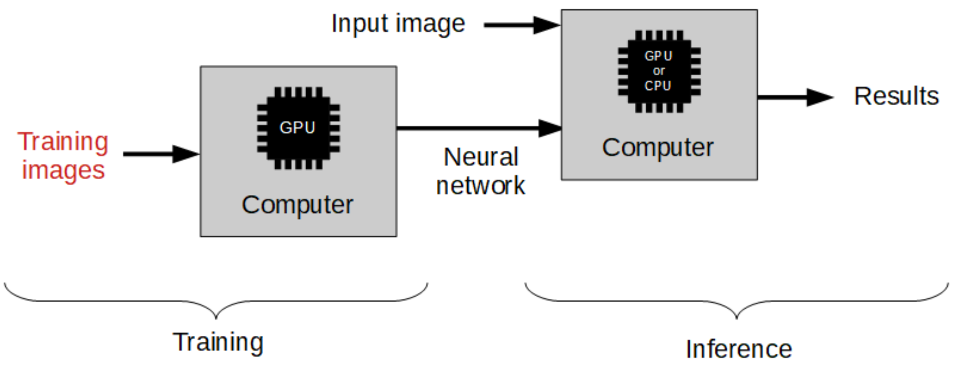
The use of deep learning functionality includes two stages:

1. Training – generating a model based on features learned from training samples,
2. Inference – applying the model on new images in order to perform the actual machine vision task.

The difference to the traditional image analysis approach is presented in the diagrams below:



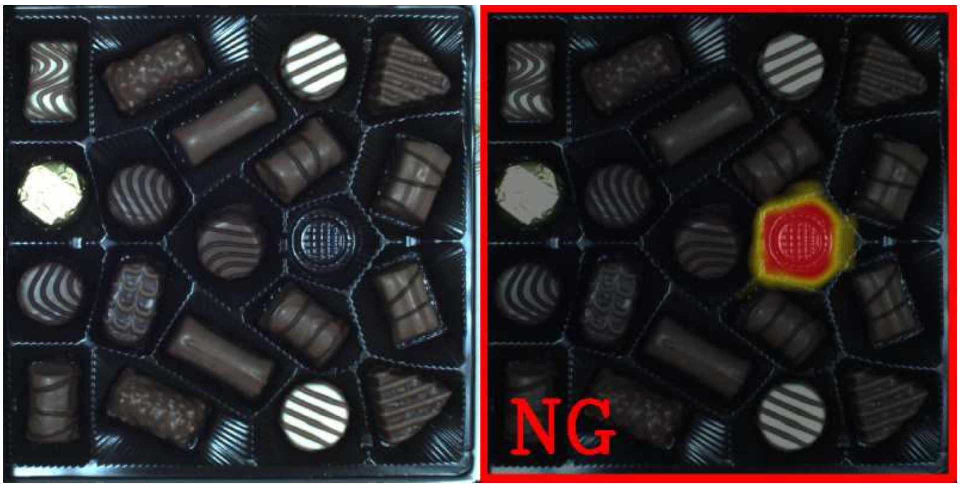
Traditional approach: The algorithm must be designed by a human specialist.



Machine learning approach: We only need to provide a training set of labeled images.

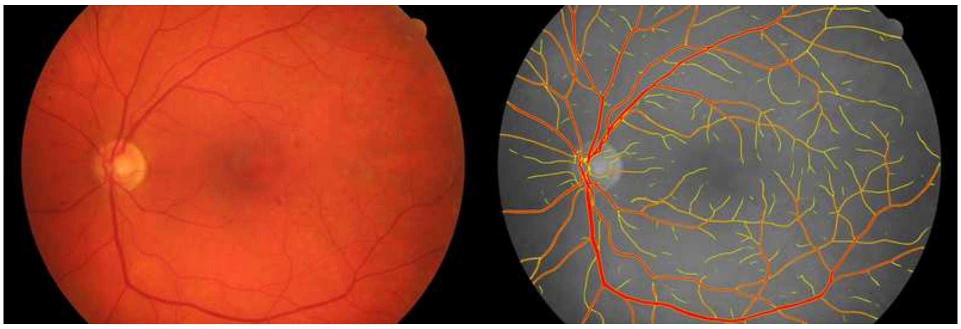
Overview of Deep Learning Tools

1. **Anomaly Detection** – this technique is used to detect anomalous (unusual or unexpected) samples. It only needs a set of fault-free samples to learn the model of normal appearance. Optionally, several faulty samples can be added to better define the threshold of tolerable variations. This tool is useful especially in cases where it is difficult to specify all possible types of defects or where negative samples are simply not available. The output of this tool are: a classification result (normal or faulty), an abnormality score and a (rough) heatmap of anomalies in the image.



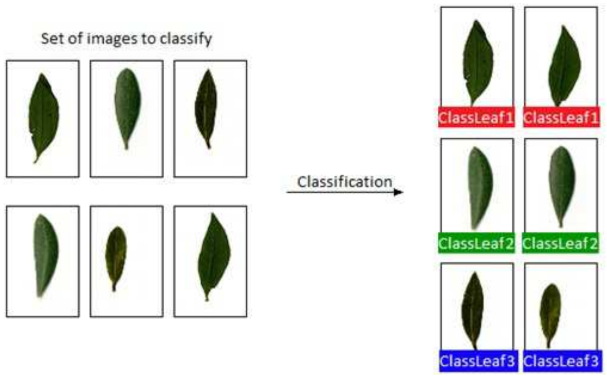
An example of a missing object detection using AvsFilter_DL_DetectAnomalies2 tool.
Left: The original image with a missing element. Right: The classification result with a heatmap of anomalies.

2. **Feature Detection (segmentation)** – this technique is used to precisely segment one or more classes of pixel-wise 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.



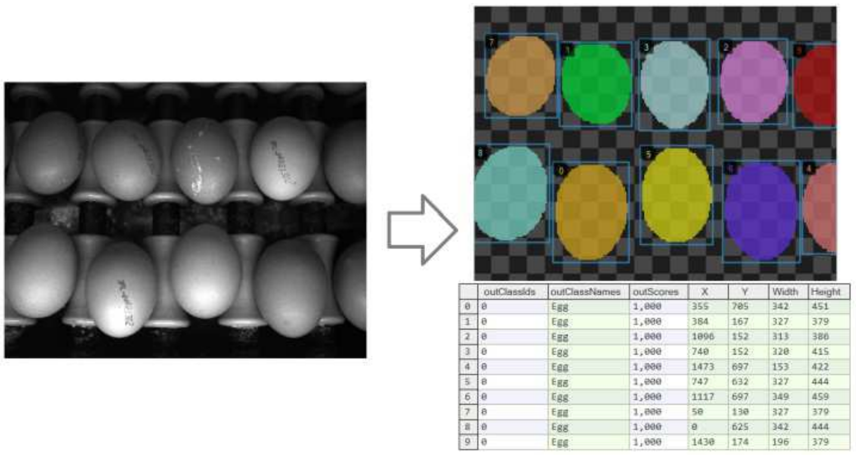
An example of image segmentation using AvsFilter_DL_DetectFeatures tool.
Left: The original image of the fundus. Right: The segmentation of blood vessels.

3. **Object Classification** – this technique is used to identify an object in a selected region with one of user-defined classes. First, it is necessary to provide a training set of labeled images. The result of this technique is: the name of detected class and a classification confidence level.



An example of object classification using AvsFilter_DL_ClassifyObject tool.

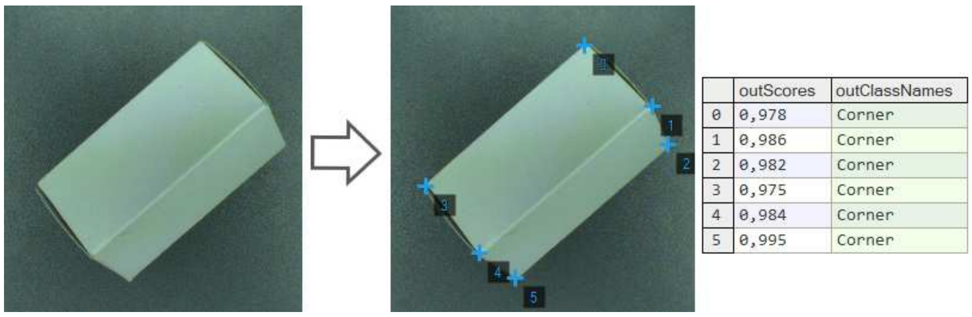
4. **Instance Segmentation** – this technique is used to locate, segment and classify one or multiple objects within an image. The training requires the user to draw regions corresponding to objects in an image and assign them to classes. The result is a list of detected objects – with their bounding boxes, masks (segmented regions), class IDs, names and membership probabilities.



	outClassIds	outClassNames	outScores	X	Y	Width	Height
0	0	Egg	1,000	355	705	342	451
1	0	Egg	1,000	384	167	327	379
2	0	Egg	1,000	1096	152	313	386
3	0	Egg	1,000	740	152	320	415
4	0	Egg	1,000	1473	697	153	422
5	0	Egg	1,000	747	632	327	444
6	0	Egg	1,000	1117	697	349	459
7	0	Egg	1,000	50	130	327	379
8	0	Egg	1,000	0	625	342	444
9	0	Egg	1,000	1430	174	196	379

An example of instance segmentation using AvsFilter_DL_SegmentInstances tool. Left: The original image. Right: The resulting list of detected objects.

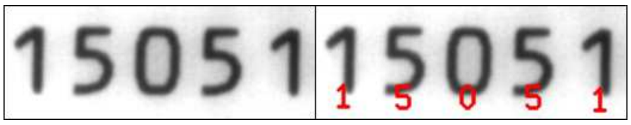
5. **Point Location** – this technique is used to precisely locate and classify key points, characteristic parts and small objects within an image. The training requires the user to mark points of appropriate classes on the training images. The result is a list of predicted point locations with corresponding class predictions and confidence scores.



	outScores	outClassNames
0	0,978	Corner
1	0,986	Corner
2	0,982	Corner
3	0,975	Corner
4	0,984	Corner
5	0,995	Corner

An example of point location using AvsFilter_DL_LocatePoints tool. Left: The original image. Right: The resulting list of detected points.

6. **Reading Characters** – this technique is used to locate and recognize characters within an image. The result is a list of found characters.



An example of optical character recognition using AvsFilter_DL_ReadCharacters tool. Left: The original image. Right: The image with the recognized characters drawn.

Basic Terminology

You do not need to have the specialistic scientific knowledge to develop your deep learning solutions. However, it is highly recommended to understand the basic terminology and principles behind the process.

Deep neural networks

Aurora Vision provides access to several standardized deep neural networks architectures created, adjusted and tested to solve industrial machine vision tasks. Each of the networks is a set of trainable convolutional filters and neural connections which can model complex transformations of an image with the goal to extract relevant features and use them to solve a particular problem. However, these networks are useless without proper amount of good quality data provided for training process. This documentation presents necessary practical hints on creating an effective deep learning model.

Depth of a neural network

Due to various levels of task 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 a neural network (i.e. the number of layers and filters) and the 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)
    - A problem is simple to define.
    - A problem could be easily solved by a human inspector.
    - A short time of execution is required.
    - Background and lighting do not change across images.
    - Well-positioned objects and good quality of images.
2. Standard depth (default, value 3)
    - Suitable for a majority of applications without any special conditions.
    - A modern CUDA-enabled GPU is available.
3. High depth (value 4-5)
    - **A big amount of training data is available.**
    - A problem is hard or very complex to define and solve.
    - Complicated irregular patterns across images.
    - Long training and execution times are not a problem.
    - A large amount of GPU RAM (≥4GB) is available.
    - Varying background, lighting 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.

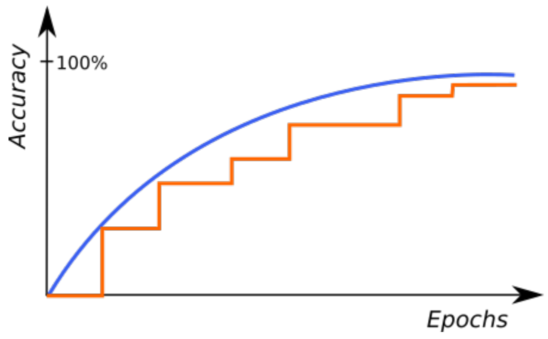
Training process

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:

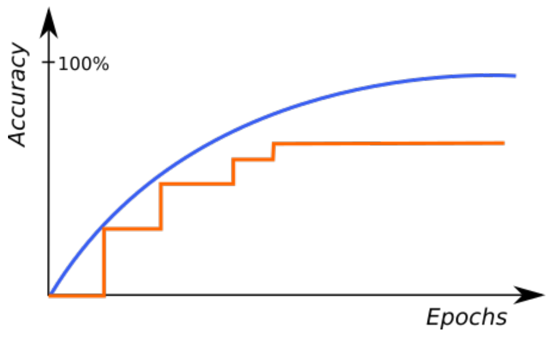
1. selection of a small subset (**batch**) of training samples,
2. calculation of an error measure for these samples,
3. updating the weights to achieve lower error for these samples.

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 the 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, the model can still improve.
2. Both training and validation scores has stopped improving – keep training for a few iterations more and stop if there is 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** to your training data (remembering exact samples rather than learning rules about features). It may also be caused by too small amount of diverse samples or too low complexity of the problem for a network selected (try lower **Network Depth**).



An example of correct training.



A graph characteristic for network overfitting.

The above graphs represent training progress in the Deep Learning Editor. The blue line indicates performance on the training samples, and the orange line represents performance on the 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.

Stopping Conditions

The user can stop the training manually by clicking the **Stop** button. Alternatively, it is also possible to set one or more stopping conditions:

1. **Iteration Count** – training will stop after a fixed number of iterations.
2. **Iterations without Improvement** – training will stop when the best validation score was not improved for a given number of iterations.
3. **Time** – training will stop after a given number of minutes has passed.
4. **Validation Accuracy** or **Validation Error** – training will stop when the validation score reaches a given value.

Preprocessing

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

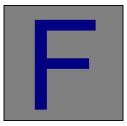
1. **Downsample** – reduction of the image size to accelerate training and execution times, at the expense of lower level of details possible to detect. Increasing this parameter by 1 will result in downsampling by the factor of 2 over both image dimension.
2. **Convert to Grayscale** – while working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

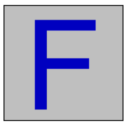
Augmentation

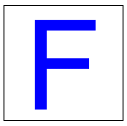
In case when the number of training images can be too small to represent all possible variations of samples, it is recommended to use data augmentations that add artificially modified samples during training. This option will also help avoiding overfitting.

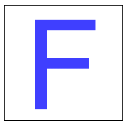
Below is a description of the available augmentations and examples of the corresponding transformations:

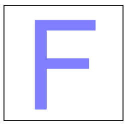
1. **Luminance** – change brightness of samples by a random percentage (between -ParameterValue and +ParameterValue) of pixel values (0-255). For a given augmentation values, samples as below can be added to the training set.


Luminance=-50.

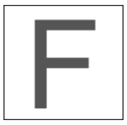

Luminance=-25.

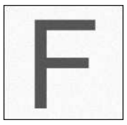

Original image.

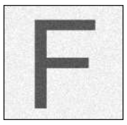

Luminance=25.

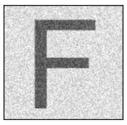

Luminance=50.

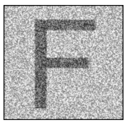
2. **Noise** – modify samples with uniform noise. Value of each channel and pixel is modified separately, by random percentage (between -ParameterValue and +ParameterValue) of pixel values (0-255). Please note that choosing an appropriate augmentation value should depend on the size of the feature in pixels. Larger value will have a much greater impact on small objects than on large objects. For a tile with the feature "F" with the size of 130x130 pixels and a given augmentation values, samples as below can be added to the training set.:

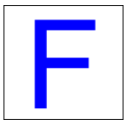

Original grayscale image.

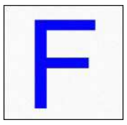

Grayscale image. Noise=4.

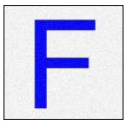

Grayscale image. Noise=10.

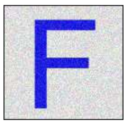

Grayscale image. Noise=25.

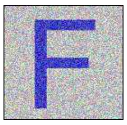

Grayscale image. Noise=50.


Original RGB image.

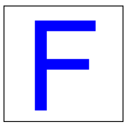

RGB image. Noise=4.

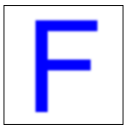

RGB image. Noise=10.

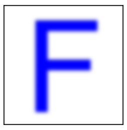

RGB image. Noise=25.

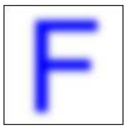

RGB image. Noise=50.

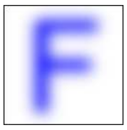
3. **Gaussian Blur** – blur samples with a kernel of a size randomly selected between 0 and the provided maximum kernel size. Please note that choosing an appropriate Gaussian Blur Kernel Size should depend on the size of the feature in pixels. Larger kernel sizes will have a much greater impact on small objects than on large objects. For a tile with the feature "F" with the size of 130x130 pixels and a given augmentation values, samples as below can be added to the training set.:


Original image.


Gaussian Blur=5.

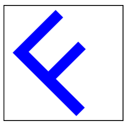

Gaussian Blur=10.

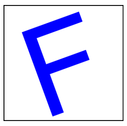

Gaussian Blur=25.

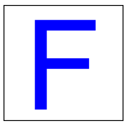

Gaussian Blur=50.

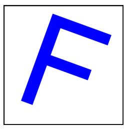
4. **Rotation** – rotate samples by a random angle between -ParameterValue and +ParameterValue. Measured in degrees.

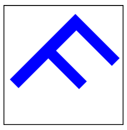
   In Detect Features, Locate Points and Detect Anomalies, for a tile with the feature "F" and given augmentation values, samples as below can be added to the training set.


Tile rotation=-45°.


Tile rotation=-20°.


Original tile.


Tile rotation=20°.


Tile rotation=45°.

   In Classify Object and Segment Instances, for an image with the feature "F" and given augmentation values, samples as below can be added to the training set.

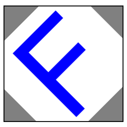


Image rotation=-45°.

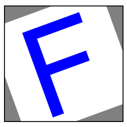
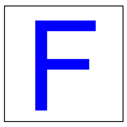


Image rotation=-20°.


Original image.

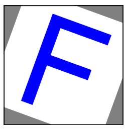


Image rotation=20°.

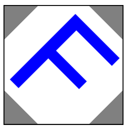
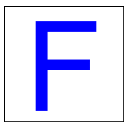
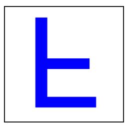
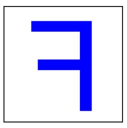


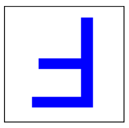
Image rotation=45°.

5. **Flip Up-Down** – reflect samples along the X axis.
6. **Flip Left-Right** – reflect samples along the Y axis.


No flips.

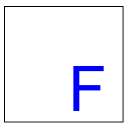

Up-Down flip.

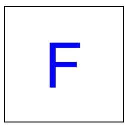

Left-Right flip.

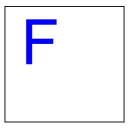

Both flips.

7. **Relative Translation** – translate samples by a random shift, defined as a percentage (between -ParameterValue and +ParameterValue) of the tile (in Detect Features, Locate Points and Detect Anomalies) or the image size (in Classify Object and Segment Instances). Works independently in both X and Y dimensions.

   In Detect Features, Locate Points and Detect Anomalies, for a tile with the feature "F" and given augmentation values, samples as below can be added to the training set.


Tile translation x=20%, y=20%.


Original tile.


Tile translation x=-20%, y=-20%.

   In Classify Object and Segment Instances, for an image with the feature "F" and given augmentation values, samples as below can be added to the training set.

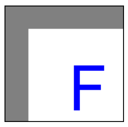
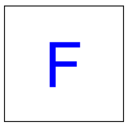


Image translation x=20%, y=20%.


Original image.

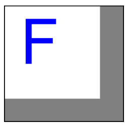
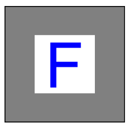
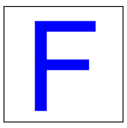
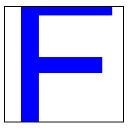


Image translation x=-20%, y=-20%.

8. **Scale** – resize samples relatively to their original size by a random percentage between the provided minimum scale and maximum scale.

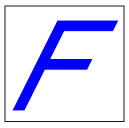

Resize=50%.

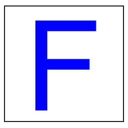

Original image.

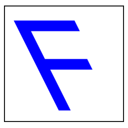

Resize=150%.

9. **Horizontal Shear** – shear samples horizontally by an angle between -ParameterValue and +ParameterValue. Measured in degrees.

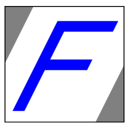
   In Detect Features, Locate Points and Detect Anomalies, for a tile with the feature "F" and given augmentation values, samples as below can be added to the training set.

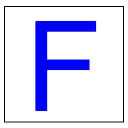

Horizontal Shear=-30.

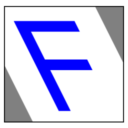

Original tile.


Horizontal Shear=30.

   In Classify Object and Segment Instances, for an image with the feature "F" and given augmentation values, samples as below can be added to the training set.

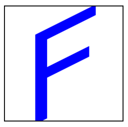

Horizontal Shear=-30.

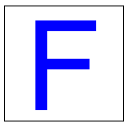

Original image.

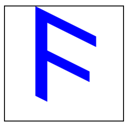

Horizontal Shear=30.

10. **Vertical Shear** – analogous to Horizontal Shear.

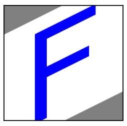
   In Detect Features, Locate Points, and Detect Anomalies, for a tile with the feature "F" and given augmentation values, samples as below can be added to the training set.

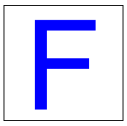

Vertical Shear=-30.

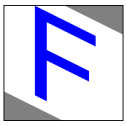

Original tile.


Vertical Shear=30.

   In Classify Object and Segment Instances, for an image with the feature "F" and given augmentation values, samples as below can be added to the training set.


Vertical Shear=-30.


Original image.


Vertical Shear=30.

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 a simple example, the Rotation should not be enabled if rotations are not expected in a production environment. Enabling augmentations also increases the network training time (but does not affect execution time!)

2. Anomaly Detection

Aurora Vision Deep Learning provides three ways of defect detection:

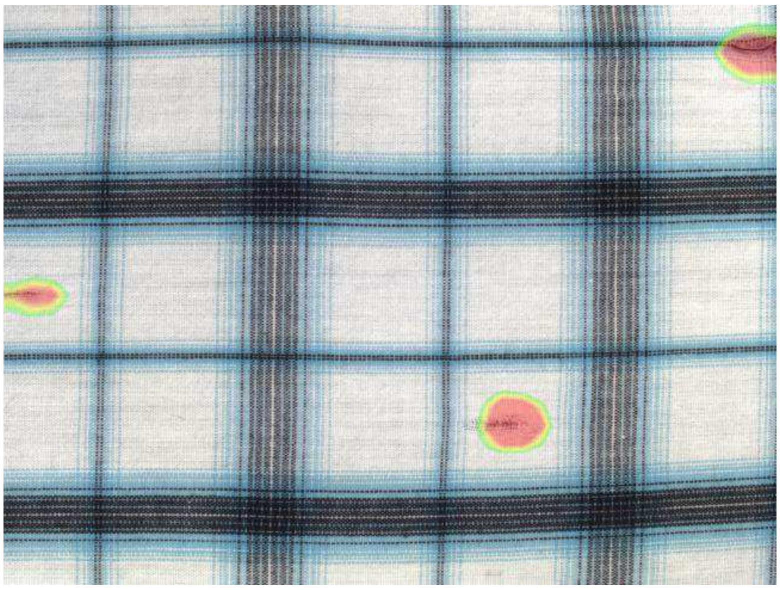
- AvsFilter_DL_DetectAnomalies1
- AvsFilter_DL_DetectAnomalies2 Single Class
- AvsFilter_DL_DetectAnomalies2 Golden Template

The AvsFilter_DL_DetectAnomalies1 (reconstructive approach) uses deep neural networks to remove defects from the input image by reconstructing the affected regions. It is used to analyze images in **fragments** of size determined by the Feature Size parameter.
This approach is based on reconstructing an image without defects and then comparing it with the original one. It filters out all patterns smaller than Feature Size that were not present in the training set.

The AvsFilter_DL_DetectAnomalies2 **Single Class** uses a simpler algorithm than Golden Template. It uses less space and the iteration time is shorter. It can be used with less complex objects.

The AvsFilter_DL_DetectAnomalies2 **Golden Template** is an appropriate method for positioned objects with complex details. The tool divides the images into regions and creates a separate model for each region. The tool has the **Texture Mode** dedicated for texture defects detection. It can be used for plain surfaces or the ones with a simple pattern.

To sum up, while choosing the tool for anomaly detection, first check the **Golden Template** with the **Texture Mode** on or off, depending on the object's kind. If the model takes too much space or the iteration is too long, please try the **Single Class** tool. If the object is complex and its position is unstable, please check the AvsFilter_DL_DetectAnomalies1 approach.


An example of textile defect detection using the AvsFilter_DL_DetectAnomalies2.

Parameters

- **Feature Size** is related to AvsFilter_DL_DetectAnomalies1 and AvsFilter_DL_DetectAnomalies2 **Single Class** approach. It corresponds to the expected defect size and it is the most significant one in terms of both quality and speed of inspection. It it is represented by a green square in the Image window of the Editor. The common denominator of all fragment based approaches is that the **Feature Size should be adjusted so that it contains common defects with some margin.**
For AvsFilter_DL_DetectAnomalies1 large Feature Size will cause small defects to be ignored, however the inference time will be shortened considerably. Heatmap precision will also be lowered. For AvsFilter_DL_DetectAnomalies2 **Single Class** large Feature Size increases training as well as inference time and memory requirements. Consider using Downscale parameter instead of increasing the Feature Size.
- **Sampling Density** is related to AvsFilter_DL_DetectAnomalies1 and AvsFilter_DL_DetectAnomalies2 **Single Class** approach. It controls the spatial resolution of both training and inspection. The higher 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.
- **Max Translation** is related to AvsFilter_DL_DetectAnomalies2 **Golden Template** approach. It is the maximal position change tolerance. If the parameter increases, the working area of a small model enlarges and the number of the created small models decreases.
- **Model Complexity** is related to AvsFilter_DL_DetectAnomalies2 **Golden Template** and AvsFilter_DL_DetectAnomalies2 **Texture** approach. Greater value may improve model effectiveness, especially for complex objects, at the expense of memory usage and interference time.

Metrics

Measuring accuracy of anomaly detection tools is a challenging task. The most straightforward approach is to calculate the Recall/Precision/F1 measures for the whole images (classified as GOOD or BAD, without looking at the locations of the anomalies). Unfortunately, such an approach is not very reliable due to several reasons, including: (1) when we have a limited number of test images (like 20), the scores will vary a lot (like Δ=5%) when just one case changes; (2) very frequently the tools we test will find random false anomalies, but will not find the right ones - and still will get high scores as the image as a whole is considered correctly classified. Thus, it may be tempting to use annotated anomaly regions and calculate the per-pixel scores. However, this would be too fine-grained. For anomaly detection tasks we do not expect the tools to be necessarily very accurate in terms of the location of defects. Individual pixels do not matter much. Instead, we expect that the anomalies are detected "more or less" at the right locations. As a matter of fact, some tools which are not very accurate in general (especially those based on auto-encoders) will produce relatively accurate outlines for the anomalies they find, while the methods based on one-class classification will usually perform better in general, but the outlines they produce will be blurred, too thin or too thick.

For these reasons, we introduced an intermediate approach to calculation of Recall. Instead of using the per-image or the per-pixel methods, we use a per-region one. Here is how we calculate Recall:

- For each anomaly region we check if there is any single pixel in the heatmap above the threshold. If it is, we increase **TP** (the number of True Positives) by one. Otherwise, we increase **FN** (the number of False Negatives) by one.
- Then we use the formula:

$$Recall = \frac{TP}{TP + FN}$$

The above method works for Recall, but cannot be directly applied to the calculation of Precision. Thus, for Precision we use a per-pixel approach, but it also comes with its own difficulties. First issue is that we often find ourselves having a lot of GOOD samples and a very limited set of BAD testing cases. This means unbalanced testing data, which in turn means that the Precision metric is highly affected with the overwhelming quantity of GOOD samples. The more GOOD samples we have (at the same amount of BAD samples), the lower Precision will be. It may be actually very low, often not reflecting the true performance of the tool. For that reason, we need to incorporate balancing into our metrics.

A second issue with Precision in real-world projects is that False Positives tend to naturally occur within BAD images, outside of the marked anomaly regions. This happens for several reasons, but is repeatable among different projects. Sometimes if there is a defect, it often means that something was broken and other parts of the object may be slightly affected too, sometimes in a visible way, sometimes with a level of ambiguity. And quite often the objects under inspection simply get affected by the process of artificially introducing defects (like someone is touching a piece of fabric and accidentally causes wrinkles that would normally not occur). For this reason, we calculate the per-pixel False Negatives only on GOOD images.

The complete procedure for calculation of Precision is:

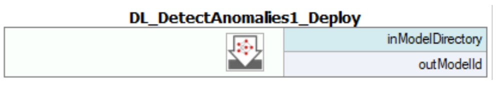
- We calculate the average **pp_TP** (the number of per-pixel True Positives) across all BAD testing samples.
- We calculate the average **pp_FP** (the number of per-pixel False Positives) across all GOOD testing samples.
- Then we use the formula:

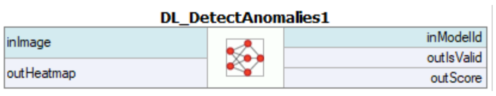
$$Precision = \frac{\overline{pp\_TP}}{\overline{pp\_TP} + \overline{pp\_FP}}$$

Finally we calculate the F1 score in the standard way, for practical reasons neglecting the fact that the Recall and Precision values that we unify were calculated in different ways. We believe that this metric is best for practical applications.

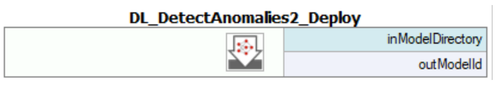
Model Usage

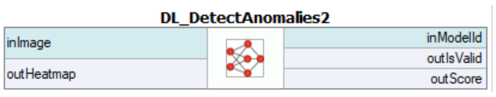
In Detect Anomalies 1 variant, a model should be loaded with AvsFilter_DL_DetectAnomalies1_Deploy prior to executing it with AvsFilter_DL_DetectAnomalies1. Alternatively, the model can be loaded directly by AvsFilter_DL_DetectAnomalies1 filter, but it will then require time-consuming initialization in the first program iteration.





In Detect Anomalies 2 variant, a model should be loaded with AvsFilter_DL_DetectAnomalies2_Deploy prior to executing it with AvsFilter_DL_DetectAnomalies2. Alternatively, model can be loaded directly by AvsFilter_DL_DetectAnomalies2 filter, but it will then require time-consuming initialization in the first program iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

3. Feature Detection (segmentation)

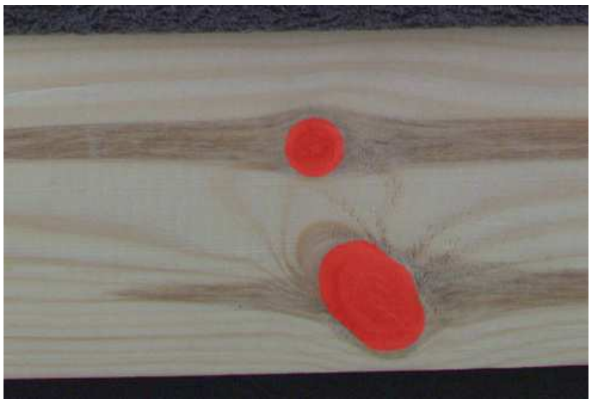
This technique is used to detect pixel-wise regions corresponding to defects or – in a general sense – to any image features. A feature here may be also something like the roads on a satellite image or an object part with a characteristic surface pattern. Sometimes it is also called pixel labeling as it assigns a class label to each pixel, but it does not separate instances of objects.

Training Data

Images used for training can be of different sizes and can have different ROIs defined. However, it is important to ensure that the scale and the characteristics of the features are consistent with that of the production environment.

Each and every feature should be marked on all training images, or the ROI should be limited to include only marked defects. Incompletely or inconsistently marked features are one of the main reasons of poor accuracy. REMEMBER: If you leave even a single piece of some feature not marked, it will be used as a negative sample and this will highly confuse the training process!

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 wood knots marked with low precision.

An example of tile cracks marked with high precision.

Multiple classes of features

It is possible to detect many classes of features separately using one model. For example, road and building like in the image below. Different 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. On the other hand, if there are two features that may be mutually confusing (e.g. roads and rivers), it is recommended to have separate classes for them and mark them, even if one of the classes is not really needed in the results. Having the confusing feature clearly marked (and not just left as the background), the neural network will focus better on avoiding misclassification.

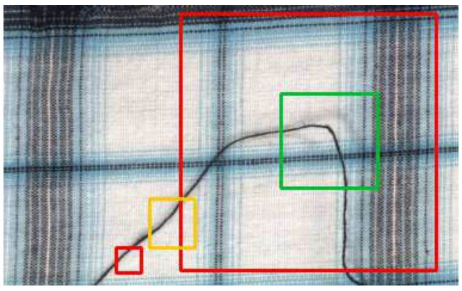
An example of marking two different classes (red roads and yellow buildings) in the one image.

Patch Size

Detect Features is an end-to-end segmentation tool which works best when analysing an image in a medium-sized square window. The size of this window is defined by the Patch Size parameter. It should be not too small, and not too big. Typically much bigger than the size (width or diameter) of the feature itself, but much less than the entire image. In a typical scenario the value of 96 or 128 works quite well.

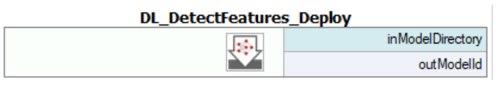
Performance Tip 1: a larger Patch Size increases the training time and requires more GPU memory and more training samples to operate effectively. When Patch Size exceeds 128 pixels and still looks too small, it is worth considering the **Downsample** option.

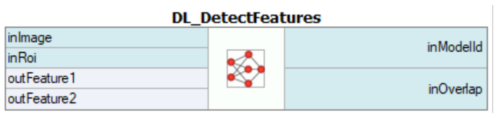
Performance Tip 2: if the execution time is not satisfying you can set the inOverlap filter input to False. It should speed up the inspection by 10-30% at the expense of less precise results.


Examples of Patch Size: too large or too small (red), maybe acceptable (yellow) and good (green). Remember that this is just an example and may vary in other cases.

Model Usage

A model should be loaded with AvsFilter_DL_DetectFeatures_Deploy filter before using AvsFilter_DL_DetectFeatures filter to perform segmentation of features. Alternatively, the model can be loaded directly by AvsFilter_DL_DetectFeatures filter, but it will result in a much longer time of the first iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

Parameters:

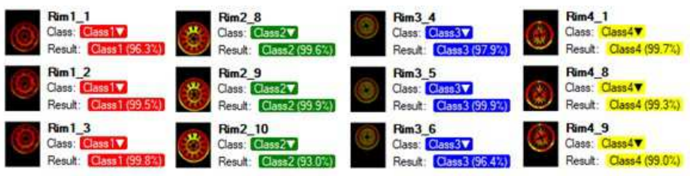
- To limit the area of image analysis you can use **inRoi** input.
- To shorten feature segmentation process you can disable **inOverlap** option. However, in most cases, it decreases segmentation quality.
- Feature segmentation results are passed in a form of bitmaps to **outHeatmaps** output as an array and **outFeature1**, **outFeature2**, **outFeature3** and **outFeature4** as separate images.

4. Object Classification

This technique is used to identify the class of an object within an image or within a specified region.

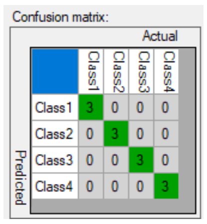
The Principle of Operation

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



Result of classification after training.

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



Confusion matrix:

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

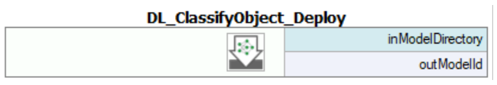
Confusion matrix presents correct (diagonal) and incorrect assignment of samples to the user defined classes.

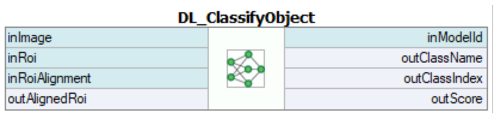
Training Parameters

In addition to the default training parameters (list of parameters available for all Deep Learning algorithms), the AvsFilter_DL_ClassifyObject tool provides a **Detail Level** parameter which enables control over the level of detail needed for a particular classification task. For majority of cases the default value of 1 is appropriate, but if images of different classes are distinguishable only by small features (e.g. granular materials like flour and salt), increasing value of this parameter may improve classification results.

Model Usage

A model should be loaded with AvsFilter_DL_ClassifyObject_Deploy filter before using AvsFilter_DL_ClassifyObject filter to perform classification. Alternatively, model can be loaded directly by AvsFilter_DL_ClassifyObject filter, but it will result in a much longer time of the first iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

Parameters:

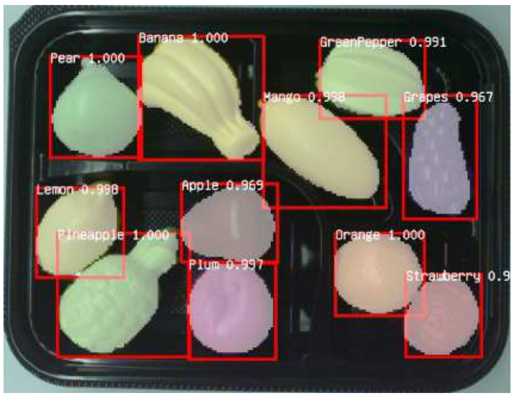
- To limit the area of image analysis you can use **inRoi** input.
- Classification results are passed to **outClassName** and **outClassIndex** outputs.
- The score value **outScore** indicates the confidence of classification.

5. Instance Segmentation

This technique is used to locate, segment and classify one 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 for detecting features like scratches or edges which may possibly have no object-like boundaries.

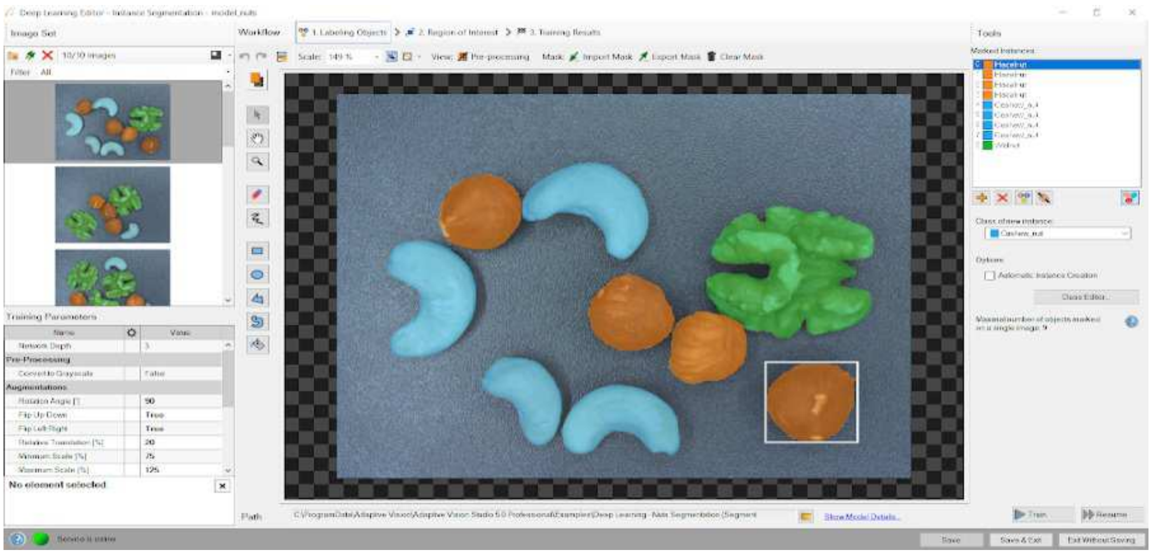
Original image.



Visualized instance segmentation results.

Training Data

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



Name	⚙	Value
Network Depth		3
Pre-Processing		
Convert to Grayscale		False
Augmentations		
Rotation Angle [°]		90
Flip Up Down		True
Flip Left Right		True
Relative Translation [%]		20
Minimum Scale [%]		75
Maximum Scale [%]		125

No element selected

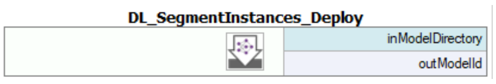
Editor for marking objects.

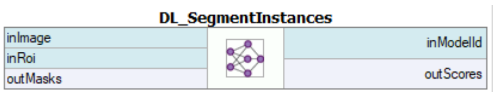
Training Parameters

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

Model Usage

A model should be loaded with AvsFilter_DL_SegmentInstances_Deploy filter before using AvsFilter_DL_SegmentInstances filter to perform classification. Alternatively, model can be loaded directly by AvsFilter_DL_SegmentInstances filter, but it will result in a much longer time of the first iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

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. By default it is equal to the maximum number of objects in the training data.
- Results describing detected objects are passed to following outputs:
  - bounding boxes: **outBoundingBoxes**,
  - class IDs: **outClassIds**,
  - class names: **outClassNames**,
  - classification scores: **outScores**,
  - masks: **outMasks**.

6. Point Location

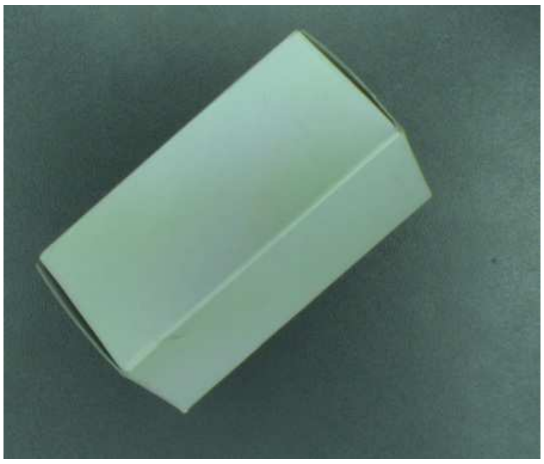
This technique is used to precisely locate and classify key points, characteristic parts and small objects within an image. The result of this technique is a list of predicted point locations with corresponding class predictions and confidence scores.

When to use point location instead of instance segmentation:

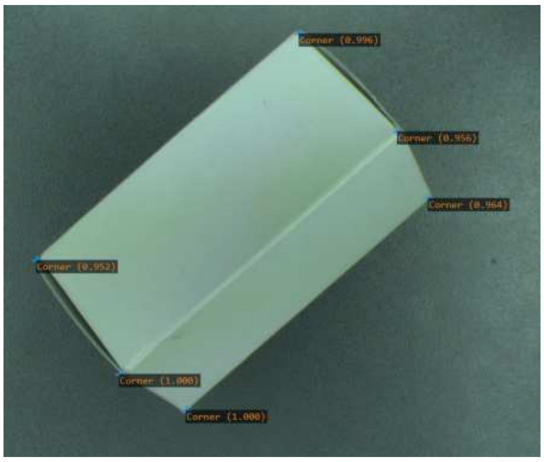
- precise location of key points and distinctive regions with no strict boundaries,
- location and classification of objects (possibly very small) when their segmentation masks and bounding boxes are not needed (e.g. in object counting).

When to use point location instead of feature detection:

- coordinates of key points, centroids of characteristic regions, objects etc. are needed.



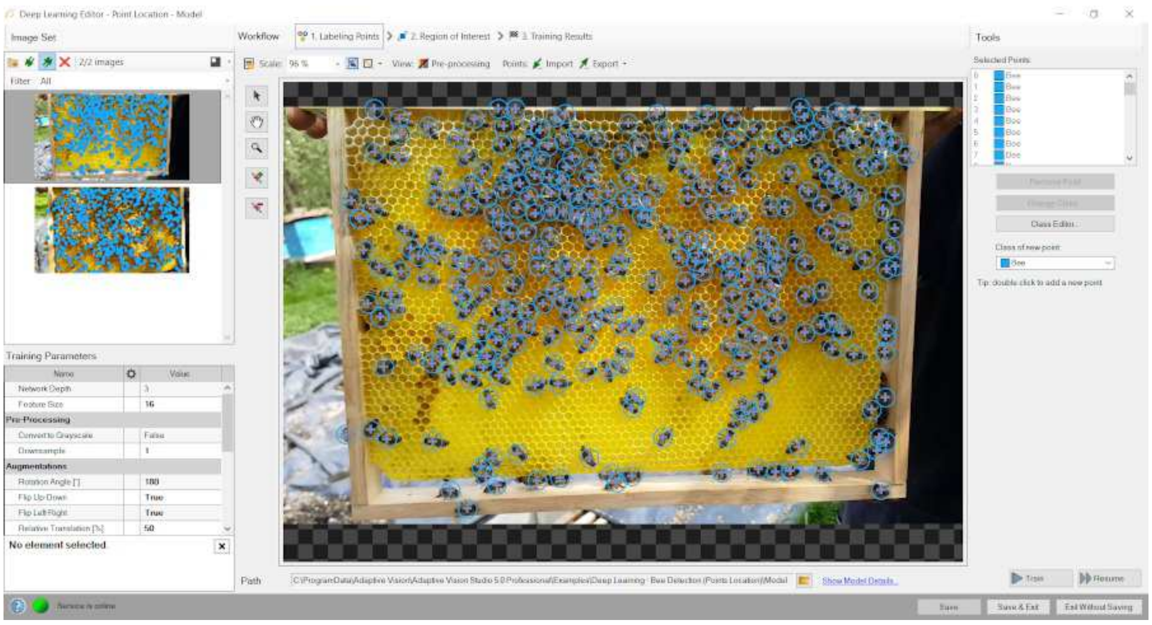
Original image.



Visualized point location results.

Training Data

The training phase requires the user to mark points of appropriate classes on the training images.



Training Parameters

Name		Value
Network Depth		3
Feature Size		16
Pre-Processing		
Convert to Grayscale		False
Downsample		1
Augmentations		
Rotation Angle [°]		180
Flip Up-Down		True
Flip Left-Right		True
Relative Translation [%]		50

No element selected

Editor for marking points.

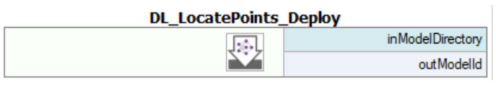
Feature Size

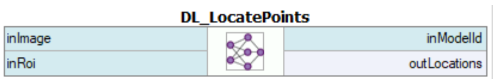
In the case of the Point Location tool, the Feature Size parameter corresponds to the size of an object or characteristic part. If images contain objects of different scales, it is recommended to use a Feature Size slightly larger than the average object size, although it may require experimenting with different values to achieve the best possible results.

Performance tip: a larger feature size increases the training time and needs more memory and training samples to operate effectively. When feature size exceeds 64 pixels and still looks too small, it is worth considering the **Downsample** option.

Model Usage

A model should be loaded with AvsFilter_DL_LocatePoints_Deploy filter before using AvsFilter_DL_LocatePoints filter to perform point location and classification. Alternatively, model can be loaded directly by AvsFilter_DL_LocatePoints filter, but it will result in a much longer time of the first iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

Parameters:

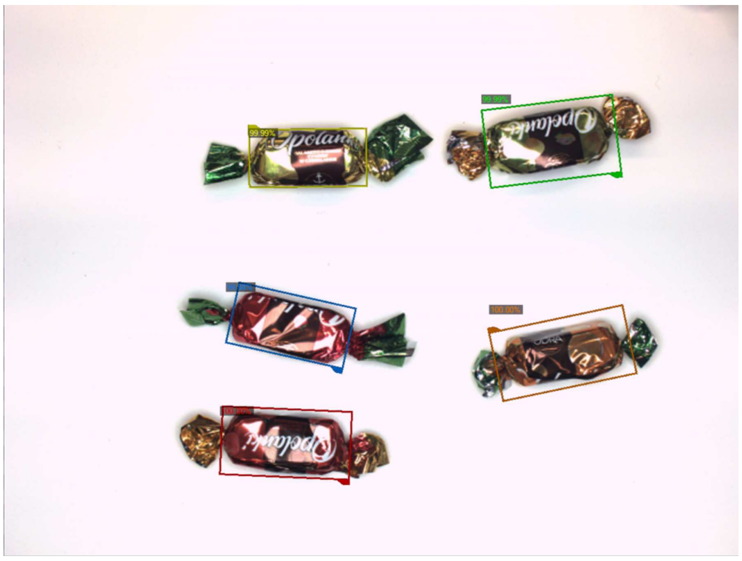
- To limit the area of image analysis you can use **inRoi** input.
- To set minimum detection score **inMinDetectionScore** parameter can be used.
- **inMinDistanceRatio** parameter can be used to set minimum distance between two points to be considered as different. The distance is computed as MinDistanceRatio * FeatureSize. If the value is not enabled, the minimum distance is based on the training data.
- To increase detection speed but with potentially slightly worse precision **inOverlap** can be set to False.
- Results describing detected points are passed to following outputs:
  - point coordinates: **outLocations**,
  - class IDs: **outClassIds**,
  - class names: **outClassNames**,
  - classification scores: **outScores**.

7. Locating objects

This technique is used to locate and classify one or multiple objects within an image. The result of this technique is a list of rectangles bounding the predicted objects with corresponding class predictions and confidence scores.

The tool returns the rectangle region containing the predicted objects and showing their approximate location and orientation , but it doesn't return the precise position of the key points of the object or the segmented region. It is an intermediate solution between the Point Location and the Instance Segmentation.

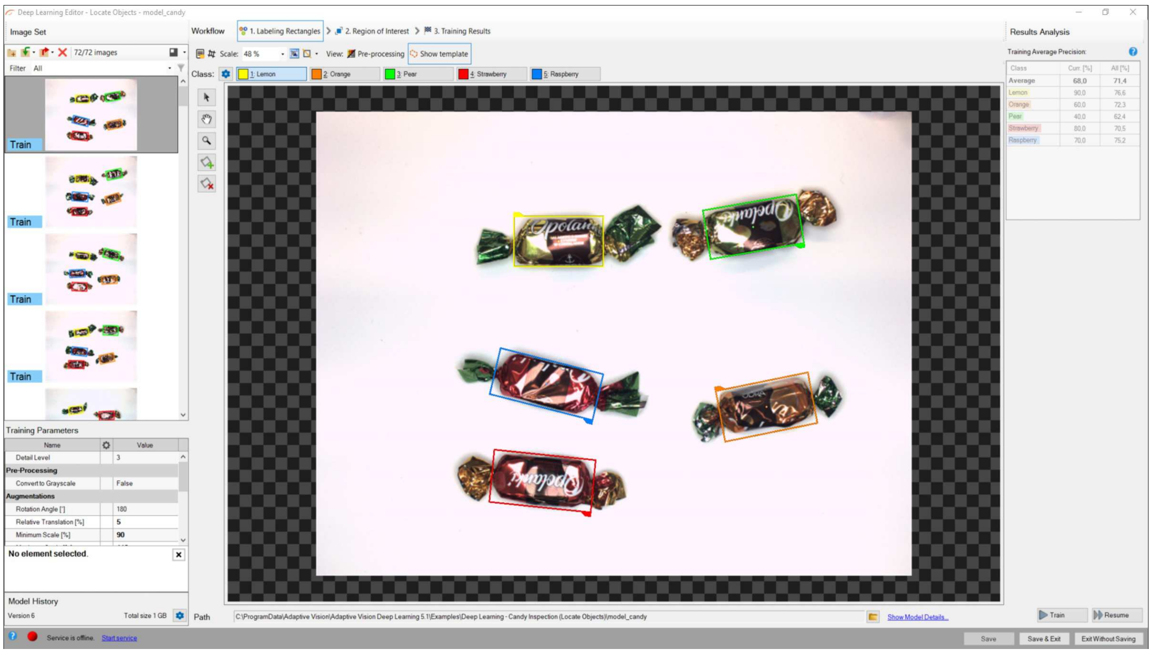
Original image.



Visualized object location results.

Training Data

The training phase requires the user to mark rectangles bounding objects of appropriate classes on the training images.



Training Average Precision:

Class	Curr. [%]	All [%]
Average	**68.0**	**71.4**
Lemon	90.0	76.6
Orange	60.0	72.3
Pear	40.0	62.4
Strawberry	80.0	70.5
Raspberry	70.0	75.2

Training Parameters

Name	⚙	Value
Detail Level		3
Pre-Processing		
Convert to Grayscale		False
Augmentations		
Rotation Angle [°]		180
Relative Translation [%]		**5**
Minimum Scale [%]		**90**

No element selected.

Model History

Version 6 Total size 1 GB

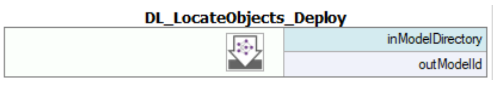
Path C:\ProgramData\Adaptive Vision\Adaptive Vision Deep Learning 5.1\Examples\Deep Learning - Candy Inspection (Locate Objects)\model_candy

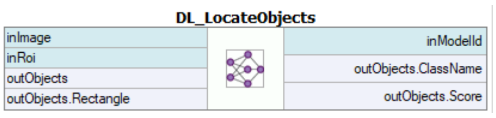
Service is offline. Start service

Editor for marking objects.

Model Usage

A model should be loaded with AvsFilter_DL_LocateObjects_Deploy filter before using AvsFilter_DL_LocateObjects filter to perform object location and classification. Alternatively, model can be loaded directly by AvsFilter_DL_LocateObjects filter, but it will result in a much longer time of the first iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

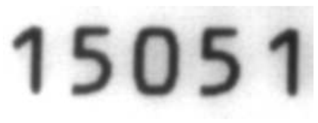
Parameters:

- To limit the area of image analysis you can use **inRoi** input.
- To set minimum detection score **inMinDetectionScore** parameter can be used.
- Results describing detected objects are passed to the object output: **outObjects**.

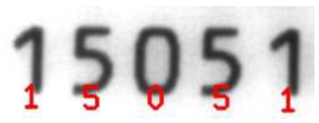
8. Reading Characters

This technique is used to locate and recognize characters within an image. The result is a list of found characters.

This tool uses the pretrained model and cannot be trained.



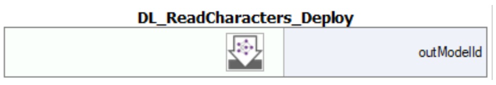
Original image.

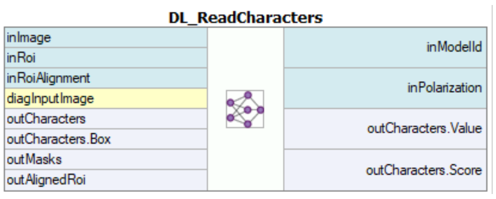


Visualized results of the read characters.

Model Usage

A model should be loaded with AvsFilter_DL_ReadCharacters_Deploy filter before using AvsFilter_DL_ReadCharacters filter to perform recognition. Alternatively, model can be loaded directly by AvsFilter_DL_ReadCharacters filter, but it will result in a much longer time of the first iteration.





Running Aurora Vision Deep Learning Service simultaneously with these filters is discouraged as it may result in degraded performance or errors.

Parameters:

- To limit the area of the image analysis and/or to set a text orientation you can use **inRoi** input.
- The average size (in pixels) of characters in the analysed area should be set with **inCharHeight** parameter.
- To improve a performance with a font with exceptionally thin or wide characters you can use **inWidthScale** input. To some extent, it may also help in a case of characters being very close to each other.
- To restrict set of recognized characters use **inCharRange** parameter.

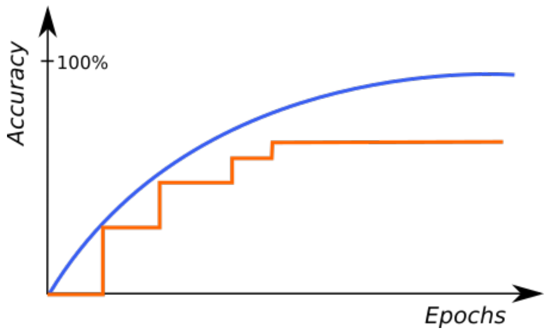
9. Troubleshooting

Below you will find a list of most common problems.

1. Network overfitting

A situation when a 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.

Possible solutions:

- Provide more real samples of different objects.
- Use more augmentations.
- Reduce Network Depth.

2. Susceptibility to changes in lighting conditions

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

Causes:

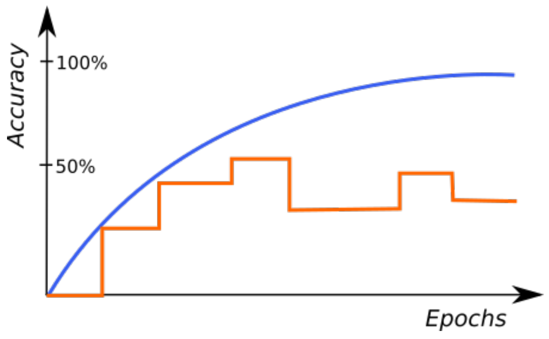
- Samples with variable lighting were not provided.

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.
- There is 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.
- Use lower Network Depth

See Also

- Deep Learning training API documentation - instruction how to perform training of Deep Learning models.

Zebra
Aurora™ Vision

This article is valid for version 5.3.4
©2007-2023 Aurora Vision