Zebra
Aurora™ Vision

Aurora Vision Studio 5.7

Programming Tips

Created: 11/10/2026

Product version: 5.7.5.80527

Table of content:

- Formulas Migration Guide to version 5.0
- Dealing with Domain Errors
- Programming Finite State Machines
- Recording Images
- Sorting, Classifying and Choosing Objects
- Optimizing Image Analysis for Speed
- Understanding OrNil Filter Variants
- Working with XML Trees

Formulas Migration Guide to version 5.0

Introduction

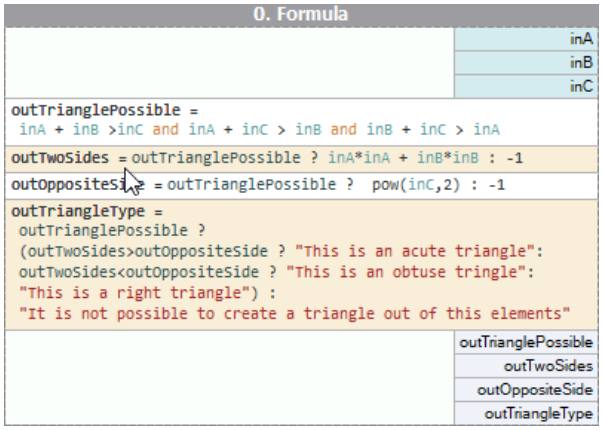
Aurora Vision Studio 5.0 comes with many new functions available directly in the formula block. Moreover, many operations, which were previously done using filters, can now be accomplished directly in the formulas. This way, the complexity of the inspection program can be reduced significantly. This document focuses on pointing out some of the key differences between formulas in Aurora Vision Studio 5.0 and its previous versions. For all functions please refer to the Formulas article in the Programming Reference section of this documentation.

Conditional operator

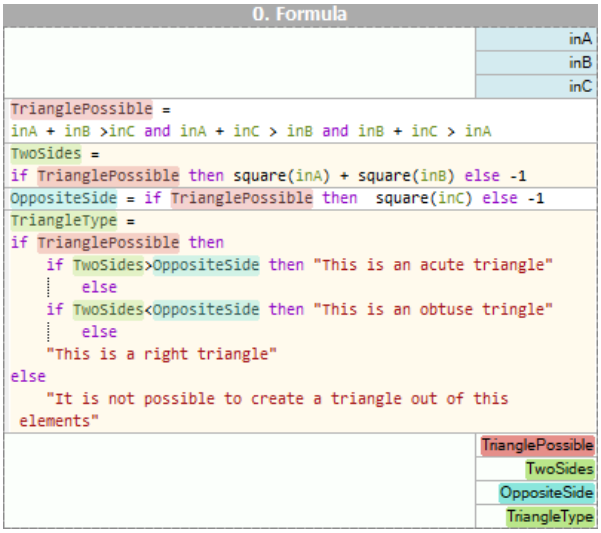
Before 5.0, a ternary operator **?:** was used to perform a conditional execution inside a formula. This compact syntax, however, might prove quite challenging and hard to understand, especially to users, who are not familiar with textual programming. In order to overcome this issue, a more user friendly **if-then-else** syntax has been introduced, while also keeping the option of using the ternary operator.

In the example below, apart from the new conditional syntax, a new **square** function has also been used, instead of inA*inA.

Before 5.0



From 5.0



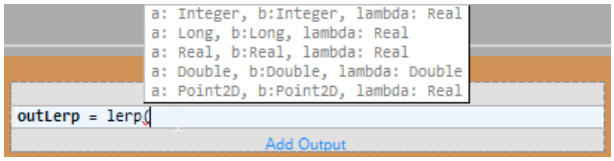
Mathematical functions

- Lerp – computes a linear interpolation between two numeric values.

Before 5.0



From 5.0



- Square – raises the argument to the power of two.
- Hypot – calculates the hypotenuse of triangle.
- Clamp – limits the specific value to the range.

Statistic and array processing functions

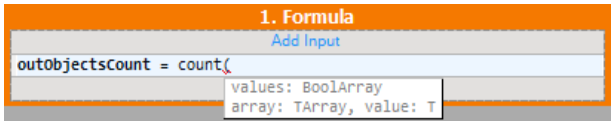
- indexOfMin – returns the index of the smallest of the values in the input array.
- indexOfMax – returns the index of the largest of the values in the input array.

- Count value in array – depending on the chosen variant, returns the number of items equal to True or counts the number of elements in the array that are equal to the value given in the second argument.

Before 5.0



From 5.0

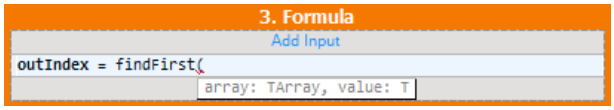


- contains – checks if the specified array contains a value.
- Find – searches the specified array for instances equal to the given value and return the index of the first found item.

Before 5.0



From 5.0

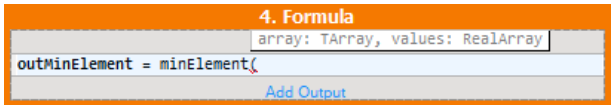


- findLast – searches the specified array for instances equal to the given value and return the index of the last found item.
- findAll – searches the specified array for instances equal to the given value and return an array of indices of all found items.
- minElement – returns an array element that corresponds to the smallest value in the array of values.

Before 5.0



From 5.0



- maxElement – returns an array element that corresponds to the biggest value in the array of values.
- removeNils – removes all Nil elements from an array.

Before 5.0



From 5.0

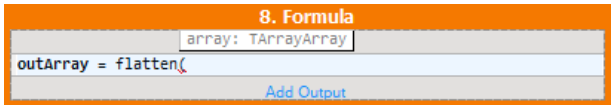


- flatten – takes an array of arrays, and concatenates all nested arrays creating a single one-dimensional array containing all individual elements.

Before 5.0



From 5.0

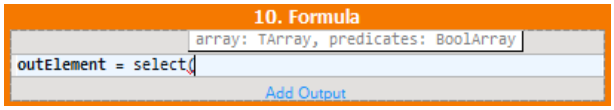


- select – selects the elements from the array of items for which the associated predicate is True.

Before 5.0



From 5.0



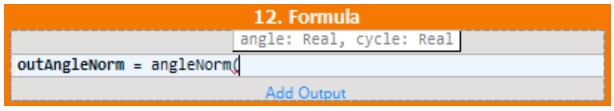
Geometry processing functions

- angleNorm – normalizes the given angle.

Before 5.0

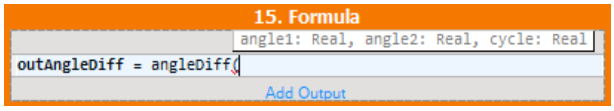


From 5.0

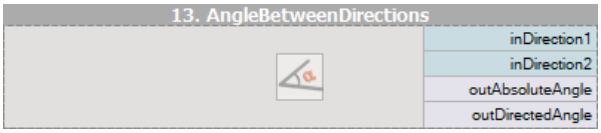


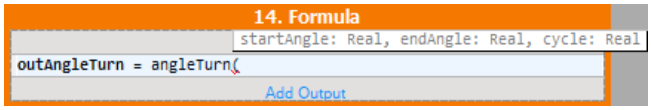
- angleDiff – calculates an absolute difference between two given angles.
  angleTurn – calculates a directional difference between two given angles.

From 5.0



Before 5.0



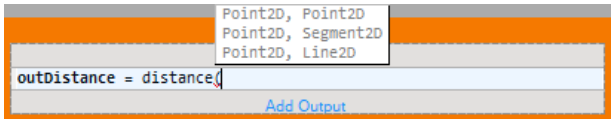


- distance – calculates the distance between a point and the closest point of a geometric primitive (one of: Point2D, Segment2D or Line2D).

Before 5.0

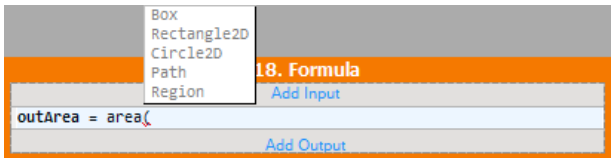


From 5.0

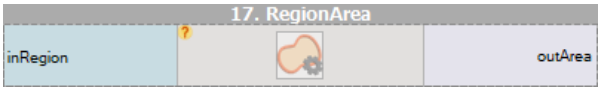


- area – calculates the surface area of a given geometric primitive.

From 5.0



Before 5.0



- dot – calculates the dot product of two vectors.
- normalize – normalizes the specified vector.

For complete information about the syntax and semantics please refer to the Formulas article in the Programming Reference section of this documentation.

Dealing with Domain Errors

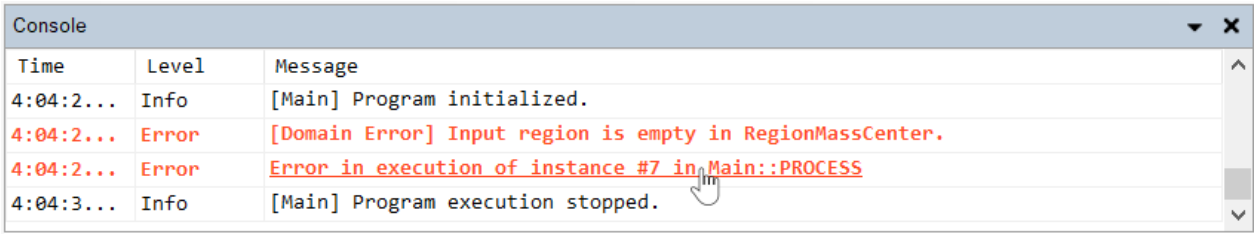
Introduction

Domain Errors stop the program execution when a precondition of a filter is not met. In general it is not possible to predict all possible *Domain Errors* automatically and this is responsibility of the user to assure that they will not appear in the final version of the program.

Example *Domain Errors*:

- The SelectChannel filter trying to get the second channel of a monochromatic image will raise "[Domain Error] Channel index out of range in SelectChannel.".
- The DivideIntegers filter when invoked with **inB** = 0 will raise "[Domain Error] Divisor is equal to zero on input in DivideIntegers.".
- The RegionMassCenter filter invoked on an empty region will raise "[Domain Error] Input region is empty in RegionMassCenter.".

Domain Errors are reported in the Console window. You can highlight the filter instance that produced a *Domain Error* by clicking on the link that appears there:



A link to a filter in the Console window.

Dealing with Empty Objects

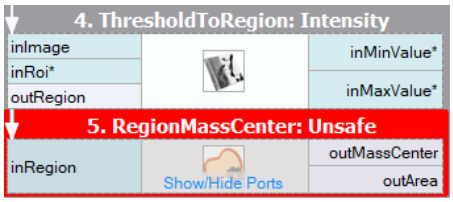
One of the most common sources of *Domain Errors* are the filters whose outputs are undefined for the empty object. Let us take the RegionMassCenter filter under consideration: where is the center of an empty region's mass? The answer is: it is undefined. Being unable to compute the result, the filter throws a *Domain Error* and stops the program execution.

If a program contains such filters and it is not possible to assure that the precondition will always be met, then the special cases have to be explicitly resolved. One way to deal with empty objects is to change the filter variant from *Unsafe* to *OrNil*. It will simply skip the execution and assign the Nil value to the filter's outputs. You can find more information on *OrNil* filter variants here.

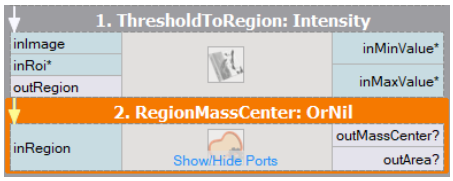
Another possible solution is to use one of the *guardian* filters, which turn empty objects into conditional processing. These filters are: SkipEmptyArray, SkipEmptyRegion, SkipEmptyPath, SkipEmptyProfile, SkipEmptyHistogram and SkipEmptyDataHistogram.

Examples

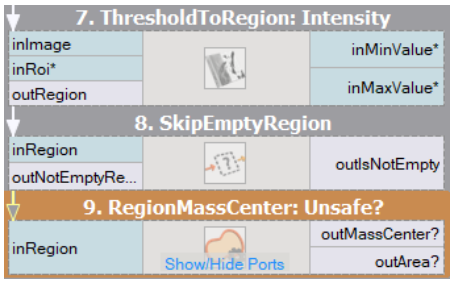
An example of a possibly erroneous situation is a use of the ThresholdToRegion filter followed by RegionMassCenter. In some cases, the first filter can produce an empty region and the second will then throw a *Domain Error*. To prevent that, the *OrNil* variant should be used to create an appropriate data flow.



RegionMassCenter throwing a Domain Error on empty input region. The program execution is stopped.



RegionMassCenter properly used in OrNil variant.



RegionMassCenter properly preceded with SkipEmptyRegion. Conditional processing appears.

Another typical example is when one is trying to select an object having the maximum value of some feature (e.g. the biggest blob). The GetMaximumElement filter can be used for this purpose, but it will throw a *Domain Error* if there are no objects found (if the array is empty). In this case using the *OrNil* option

instead of the *Unsafe* one solves the problem.

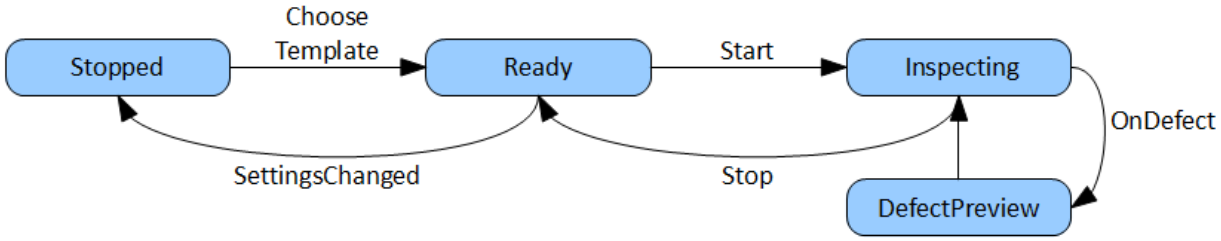
Programming Finite State Machines

Introduction

Industrial applications often require the algorithm to work in several different modes. Two most typical examples are:

- An application that has several user interface modes, e.g. the inspection mode and the model definition mode.
- An application that guides a robot arm with modes like "searching for an object", "picking an object" etc.

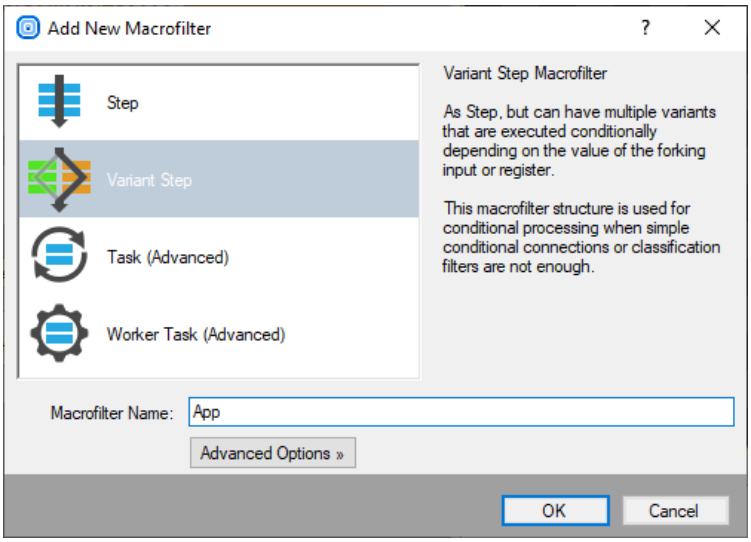
Such applications are best described and programmed as Finite State Machines (FSM). Below is an example diagram depicting one in a graphical way:



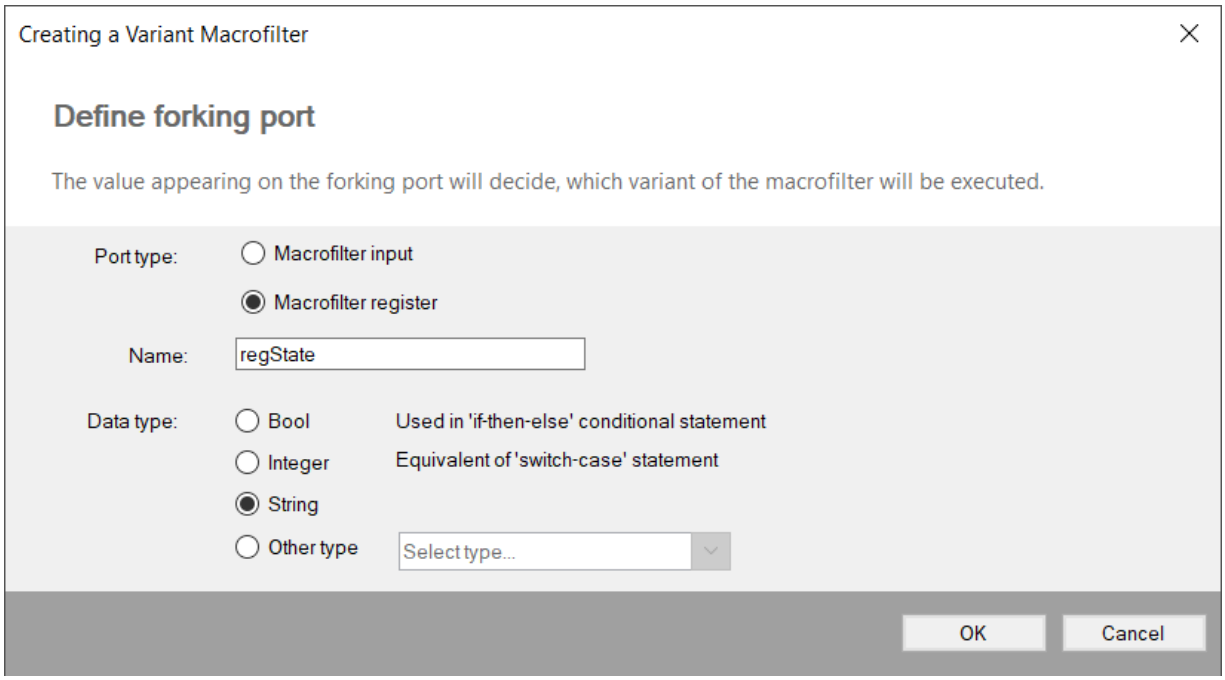
Instructions

In Aurora Vision Studio, Finite State Machines can be created with variant macrofilters and registers. The general program schema consists of a *main loop* macrofilter (a task, usually the "Main" macrofilter) and a variant macrofilter within it with variants corresponding to the states of the Finite State Machine. Individual programs may vary in details, but in most cases the following instructions provide a good starting point:

1. Create a *Variant Step* macrofilter (e.g. "App") for the State Machine with variants corresponding to individual states.



2. Use a forking register (e.g. "regState") of the String type, so that you can assign clear names to each state.



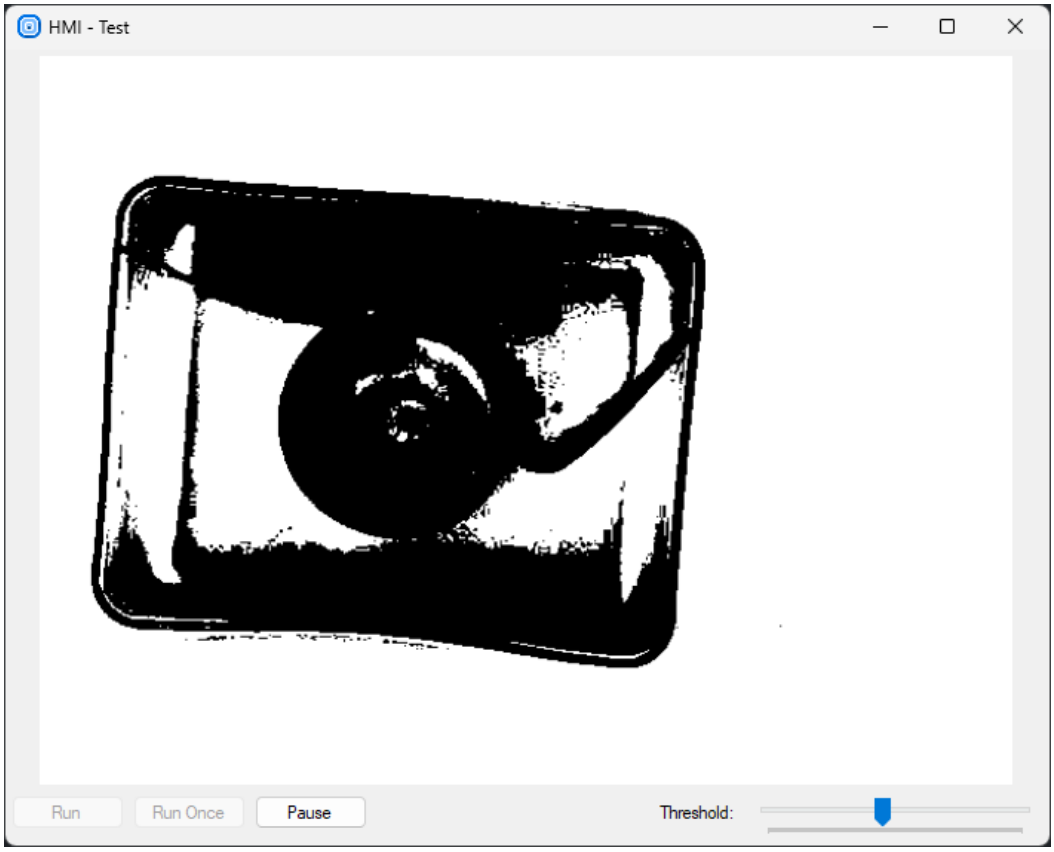
3. At first, you will have only one default variant. Remove it and add one variant with a meaningful label for each state.



4. Do not forget to set an appropriate initial value for the forking register (the initial state).
5. Add macrofilter inputs – usually for the input image and major parameters.
6. Add macrofilter outputs that will contain information about the results. In each state these outputs can be computed in a different way.
7. Create an instance of this variant macrofilter in some task with a loop, e.g. in the "Main" macrofilter.
8. In each state compute the value of the next state and connect the result to the *next* port of the forking register. Typically, an HMI button's outputs are used here as inputs to formula blocks.
9. Optional: Create a formula in the main loop task for enabling or disabling individual controls, depending on the current state (also requires exposing the current state value as the variant step's output).

Example

For a complete example, please refer to the "HMI Start-Stop" example program. It is based on two states: "Inspecting" and "Stopped". In the first state the input images are processed, in the second they are not. Two buttons, *Start* and *Stop*, allow the user to control the current state.



An example application with a simple Finite State Machine

Note: There is also a standard control, ProgramControlBox, which provides "Start", "Iterate" and "Pause" buttons. It is very easy to use, but it is not customizable. Finite State Machines allow for creating any custom set of states and transitions.

Blocking Filters

Please note that while using the Finite State Machine approach to create more complex program logic, you should avoid the use of image acquisition filters configured for the triggered mode (e.g. GigEVision_GrabImage). These filters are *blocking*, which means that the program execution will halt waiting for the trigger and no other computations are performed during this time. Events, such as button clicks, will not get processed until the next image is acquired.

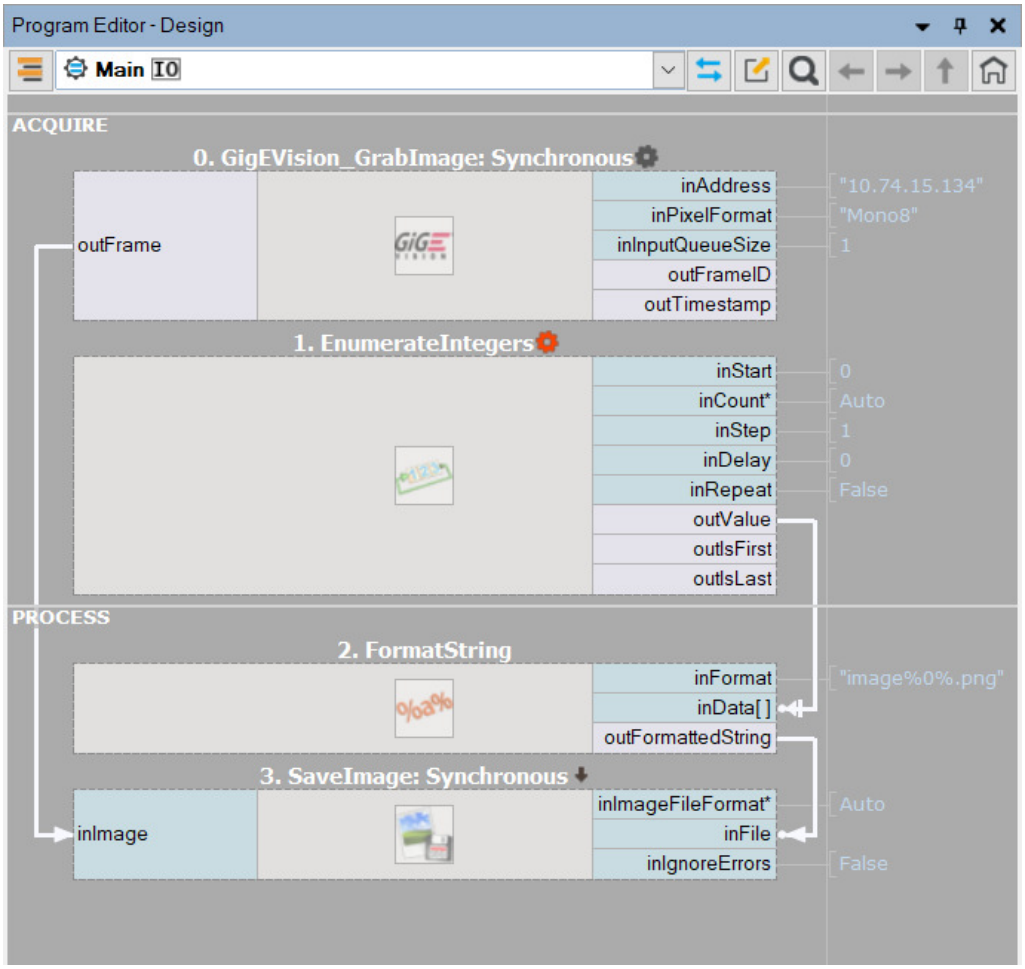
If your application is using a camera or multiple cameras in the triggered mode, then it is advisable to use filters with timeout – GigEVision_GrabImage_WithTimeout or GenICam_GrabImage_WithTimeout. These filters return *Nil* after the time is out and no image has been acquired. It is thus possible to use them in a loop, in which events can be processed and the inspection part is executed conditionally – only when there is a new input image.

See also: Handling Events in Low Frame-Rate Applications.

Recording Images

Continuous Recording

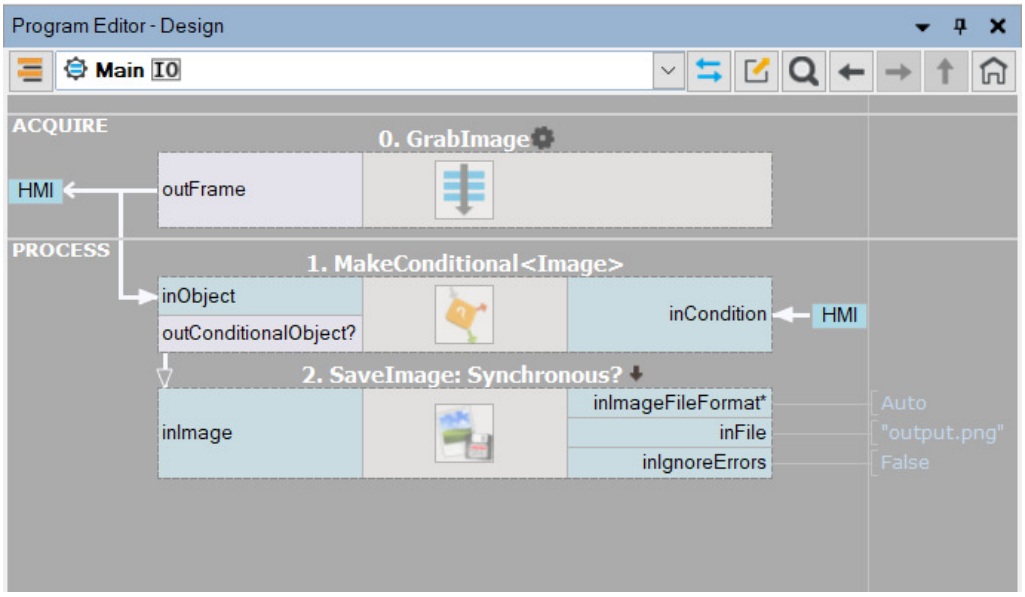
Images can be recorded to a disk with this simple program:



The RecordAllImages program.

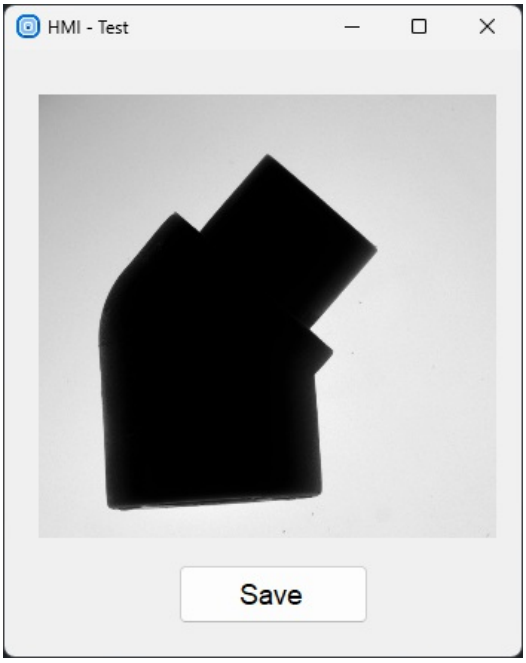
Recording Single Images

Sometimes it might be useful to have a simple HMI with a button triggering the action of saving the current image to a file. This can be done by conditional execution of the SaveImage filter:



The RecordSingleImages program.

The HMI for this program consists of an image preview and an impulse button:



HMI for recording single images.

Sorting, Classifying and Choosing Objects

Introduction

Sorting, classifying and choosing objects are three types of general programming tasks that are often an important part of machine vision inspections. For example, it might be asked how to sort detected objects by the Y coordinate, how to decide which objects are correct on the basis of a computed feature, or how to select the region having the highest area. In Aurora Vision Studio, by design, there are no filters that do just that. Instead, for maximum flexibility such tasks are divided into two steps:

1. Firstly, an array of values (features) describing the objects of interest has to be created.
2. Secondly, both the array of objects, and the array of the corresponding values, have to be passed to an appropriate sorting, classifying or choosing filter.

This approach makes it possible to use arbitrary criteria, also ones that are very specific to a particular project and have not been anticipated by the original creators of the software.

Sorting Elements of an Array

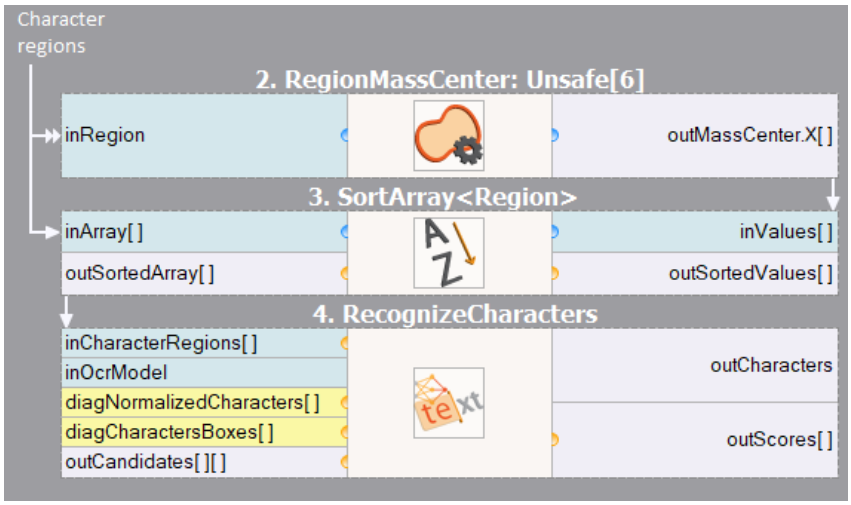
Let us consider the question of how to sort an array of objects by one of the coordinates or by some other computed feature. We will show how to achieve this on an example with manual sorting of characters from left to right for an OCR. Here is the input image with an overlay of the detected character regions:



Segmented characters which might be out of order.

After passing this array of character regions to the RecognizeCharacters filter with the **inCharacterSorting** input set to *None*, we will get the result "MAEPXEL", which has all characters recognized correctly, but in a random order.

Here is a sample program fragment that sorts the input regions by the X coordinates of their mass centers:



Sorting regions from left to right before passing them to OCR.

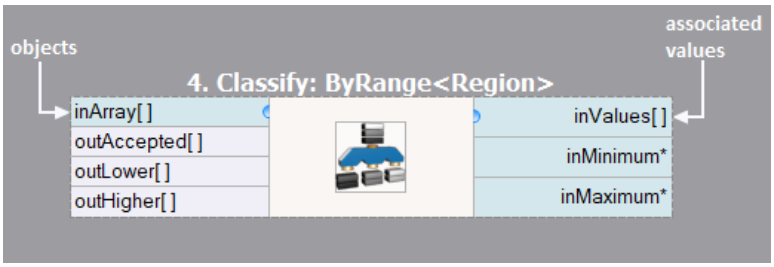
The final result is "EXAMPLE", which is both correct and in order.

Classifying Elements of an Array

An example of object classification has already been shown in the very first program example, in the First Program: Simple Blob Analysis article, where blobs were classified by the elongation feature with a dedicated filter, ClassifyRegions. Classification is a common step in many programs that process multiple objects and there is a group Classify of general filters which can classify objects by various criteria. The general scheme of object classification is:

1. Detect objects – for example using Template Matching or segmentation (e.g. ThresholdToRegion + SplitRegionIntoBlobs).
2. Compute some features – for example the area, elongation, mean brightness etc.
3. Classify the objects – using one of the Classify filters.

The general idea of how to use the ClassifyByRange filter is shown below:



The usage of the ClassifyByRange filter.

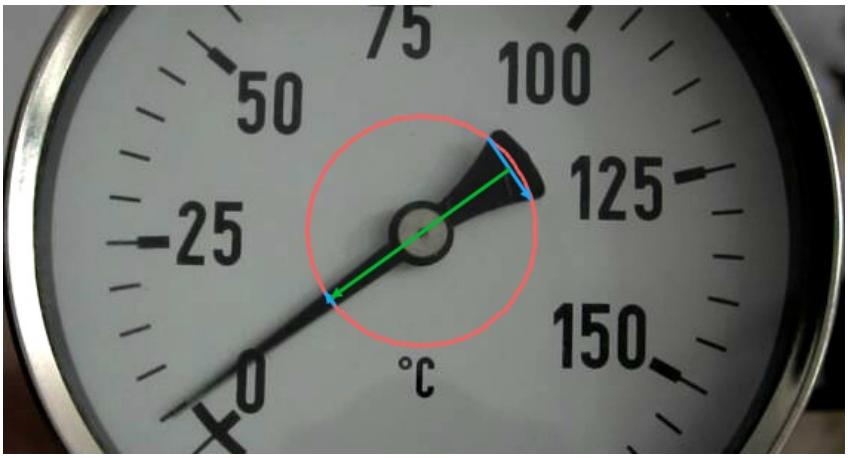
There are four elements:

- The objects to be classified are connected to the **inArray** input.
- The values corresponding to the objects are connected to the **inValues** input.
- The **inMinimum** and **inMaximum** inputs define the acceptable ranges for the values.
- The 3 outputs contain 3 arrays – containing the objects whose values were respectively: in the range, below the range and above the range.

Other filters that can be used for classification are: ClassifyByPredicate – when instead of associated real values we have associated booleans (*True* / *False*) and ClassifyByCase – when instead of associated real values we have associated indices of classes that the corresponding objects belong to.

Choosing an Element out of an Array

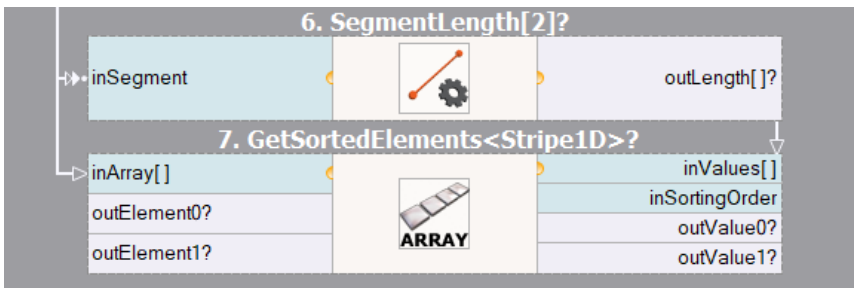
Choosing differs from sorting and classification mainly in that it is intended to produce a single value instead of an array. Let us consider the example of finding the orientation of a meter's needle:



The task of finding the orientation of a meter's needle.

There are two segments (blue) detected with the ScanExactlyNStripes filter along a circle (red). At this stage we have an array of two segments, but we do not know, which is the smaller and which is the bigger. This information is necessary to determine the orientation (green).

To solve this problem we need to compute an array of the segment's lengths using the SegmentLength filter and then get the appropriate elements with the GetSortedElements filter. This filter will produce the smaller element on the first output and the bigger element on the second output (see the picture below). Alternatively, we could use two separate filters: GetMinimumElement and GetMaximumElement.



The solution for choosing the smaller segment and the bigger segment.

Choosing one Object out of Several Individual Objects

Sometimes we have several individual objects to choose from instead of an array. For example we might want to choose one of two images for display in the HMI depending on whether some check-box is checked or not. In such case it is advisable to use the ChooseByPredicate filter, which requires two objects on the inputs and returns one of them on the output. Which one is chosen is determined by the **inCondition** input.

Warning: For efficiency reasons you should use the ChooseByPredicate filter only if the two input objects can be obtained without lengthy computations. If there are two alternative ways to compute a single value, then variant macrofilters should be used instead.

c++

The ChooseByPredicate filter is very similar to the ternary (?:) operator from the C/C++ programming language.

Choosing an Object out of the Loop

Yet another case is choosing an object out of objects that appear in different iterations. As with arrays, also here we need an associated criterion, which can be real numbers, boolean values or status of a conditional value. The available filters are:

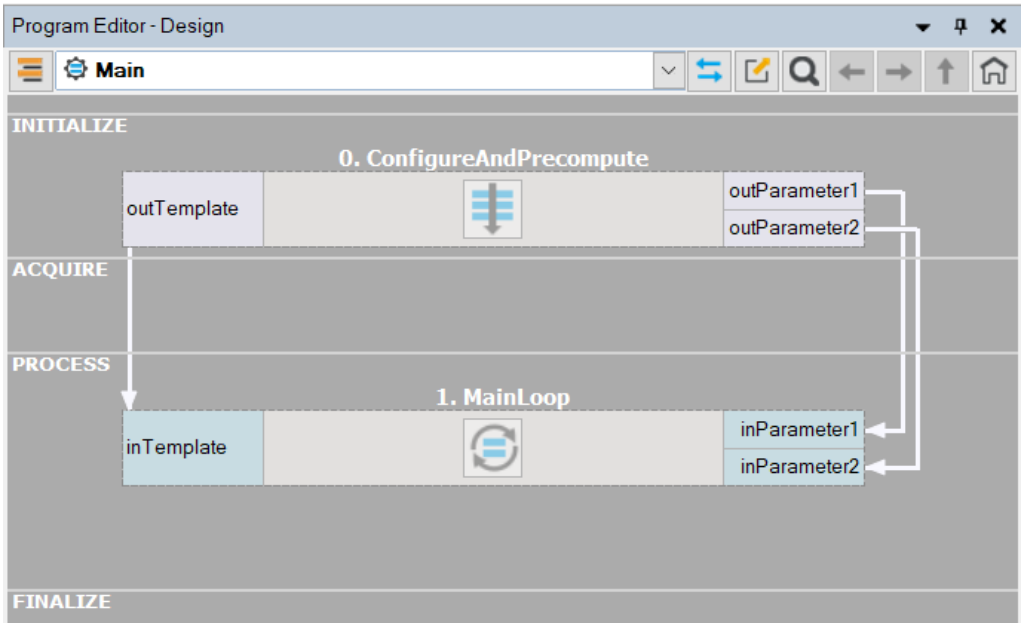
- LoopMaximum – chooses the object whose associated value was the highest.
- LoopMinimum – chooses the object whose associated value was the lowest.
- LastMarkedObject – chooses the most recent object whose associated boolean value was *True*.
- LastNotNil – chooses the most recent object which actually existed (was different than *Nil*).

Optimizing Image Analysis for Speed

General Rules

Rule #1: Avoid Unnecessary Computations.

- Appropriate Image Resolution. Choose an image resolution that suits your needs. Higher resolutions slow down processing.
- Use the **inRoi** input. Limit processing to only those pixels required in subsequent steps.
- Optimize Sequential Operations. If multiple image processing operations occur within a specific area, consider using CropImage initially.
- Image Type Efficiency. Stick to UInt8 (8-bit) images unless necessary; avoid overusing other types.
- Single-Channel Images. Opt for single-channel images when color information is not being processed.
- Pre-compute Tasks. Shift computations that only need to be performed once out of the main program loop or into a separate program. Below is an example structure of the ' 'Main ' macrofilter, illustrating this approach with two macrofilters: the first handles one-time computations, and the second manages the main program loop as a Task.



Typical program structure separating precomputing from the main loop.

Rule #2: Prefer simple solutions.

- Do not use Template Matching if simpler techniques such as Blob Analysis or 1D Edge Detection would suffice.
- Prefer pixel-precise image analysis techniques (Region Analysis) and the *Nearest Neighbour* (instead of *Bilinear*) image interpolation.
- Consider extracting higher level information early in the program pipeline – for example, it is much faster to process Regions than Images.

Rule #3: Mind the influence of the user interface.

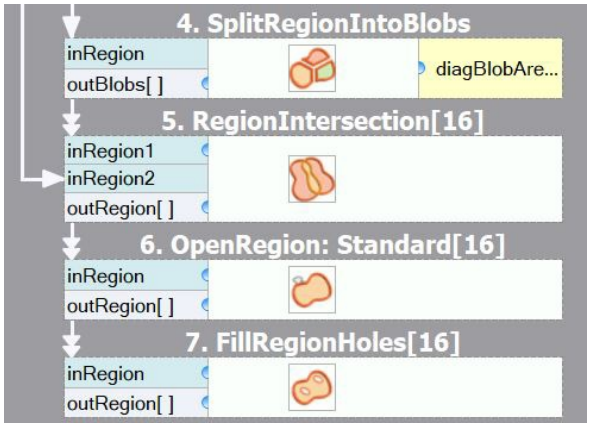
- Note that in the development environment, displaying data on the preview windows takes a lot of time. Choose *Program » Previews Update Mode » Disable Visualization* to get performance closer to what you can expect in the runtime environment.
- In the runtime environment, use the VideoBox control for image display. It is highly optimized and can display hundreds of images per second.
- Using the VideoBox controls, prefer the setting of **SizeMode: Normal**, especially if the image to be displayed is large. Also, consider using DownsampleImage or ResizeImage.
- Reduce the number of data exchange points with the HMI. For more complex user interfaces, implement a separate Worker Task to handle data refreshing.
- Prefer the *Update Data Previews Once an Iteration* option.
- Be mindful of Diagnostic Mode. Turn it off whenever you need to test speed.

- Pay attention to the information provided by the Statistics window. Before optimizing the program, make sure that you know what really needs optimizing.

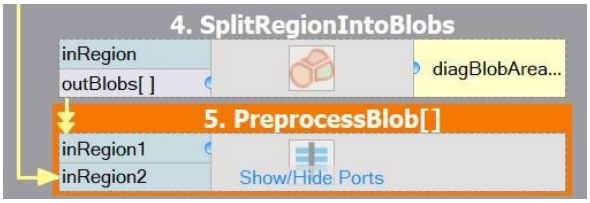
Rule #4: Mind the influence of the data flow model.

Data flow programming allows for creating high speed machine vision applications nearly as well as the standard C++ programming. This, however, requires meeting an assumption that we are using high-level tools, and image analysis is the main part. On the other hand, for low level programming tasks – like using many simple filters to process high numbers of pixels, points, or small blobs – all interpreted languages will perform significantly slower than C++.

- For performance-critical low-level programming tasks, consider User Filters.
- Prefer formula blocks over arithmetic filters like AddIntegers or DivideReals.
- Use a lower number of higher level filters (e.g. RotatePath) instead of a big number of low level filters or formulas (e.g. calculating coordinates of all individual points of the path).
- Avoid using low-level filters (such as MergeDefault or ChooseByPredicate) with non-primitive types such as Image or Region. Filters perform full copying of at least one of the input objects. Prefer using Variant Step Macrofilters instead.
- Be aware of the connections with conversions (the arrow head with a dot) – there are additional computations, which in some cases (e.g. RegionToImage) might take some time. If the same conversion is used many times, then it might be better to use the converting filter directly.
- The sequence of filters with array connections may produce a lot of data on the outputs. If only the final result is important, then consider extracting a macrofilter that will be executed in array mode as a whole, and inside of it, all the connections will be basic. For example:



Before optimizing: There are several arrays of intermediate results.



After optimizing: Only the final array is computed, reducing memory consumption. Please note that the macrofilter input is of Region, not RegionArray type.

Common Optimization Tips

Apart from the above general rules, there are also some common optimization tips related to specific filters and techniques. Here is a checklist:

- Template Matching: Do not mark the entire object as the template region, but only mark a small part that has a unique shape.
- Template Matching: Prefer high pyramid levels, i.e., leave the **inMaxPyramidLevel** set to *Auto*, or to a high value such as between *4* and *6*.
- Template Matching: Prefer setting **inEdgePolarityMode** to not **Ignore** and **inEdgeNoiseLevel** to Low.
- Template Matching: Use as high a value of the **inMinScore** input as possible.
- Template Matching: If you are processing high-resolution images, consider setting the **inMinPyramidLevel** to *1* or even *2*.
- Template Matching: When creating template matching models, try to limit the range of angles using the **inMinAngle** and **inMaxAngle** inputs.
- Template Matching: Do not expect high speed when allowing rotations and scaling at the same time. Additionally, model creation can take a significant amount of time or even fail with an "out of memory" error.
- Template Matching: Consider limiting **inSearchRegion**. It might be set manually, but sometimes it also helps to use Region Analysis techniques before Template Matching.
- Template Matching: Decrease **inEdgeCompleteness** to achieve higher speed at the expense of lower reliability. This might be useful when the pyramid cannot be made higher due to information loss.

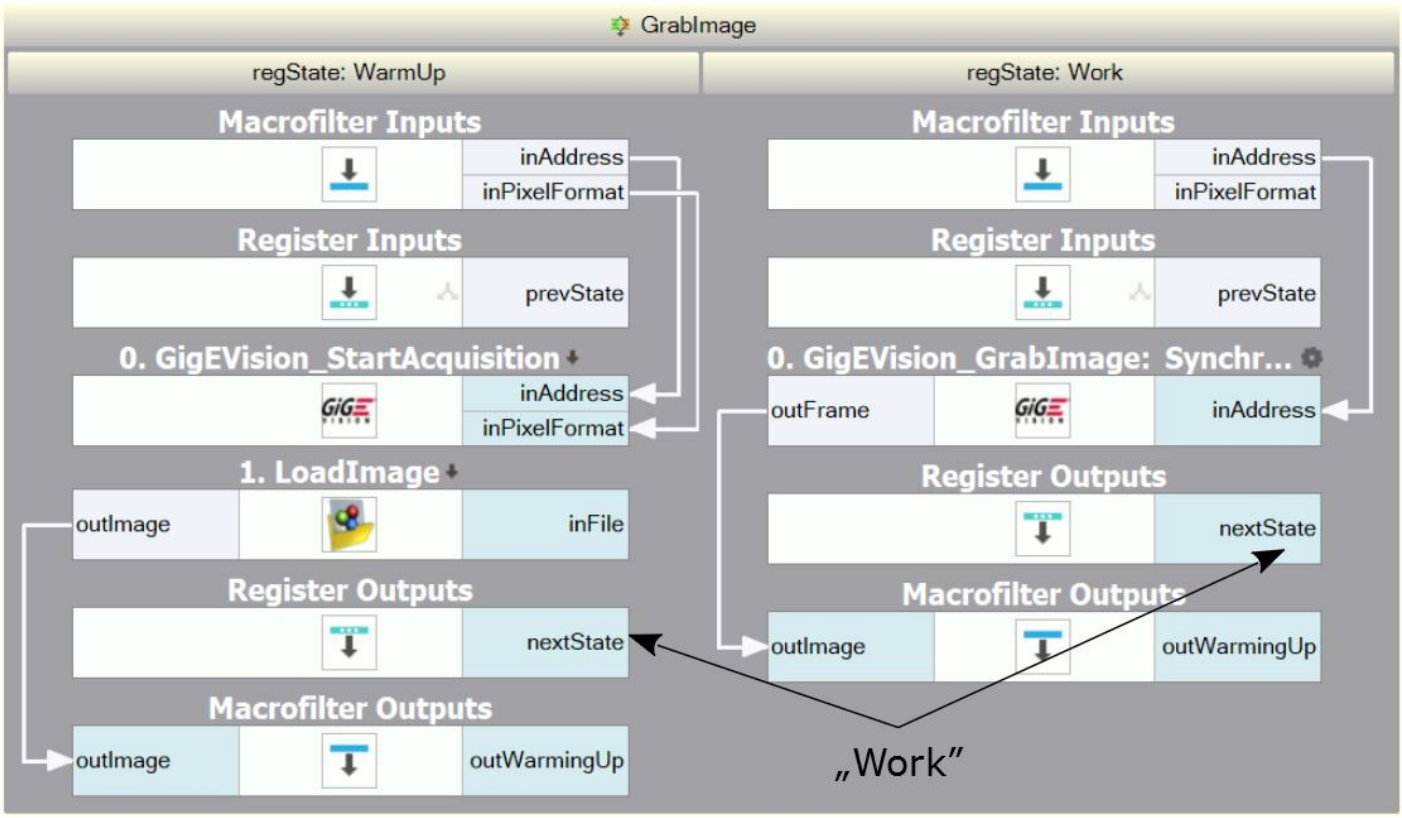
- Do not use these filters in the main program loop: CreateEdgeModel1, CreateGrayModel, TrainOcr_MLP, and TrainOcr_SVM.
- If you always transform images in the same way, consider filters from the Image Spatial Transforms Maps category instead of those from Image Spatial Transforms.
- Do not use image local transforms with arbitrarily shaped kernels: DilateImage_AnyKernel, ErodeImage_AnyKernel, and SmoothImage_Mean_AnyKernel. Consider the alternatives without the "_AnyKernel" suffix.
- SmoothImage_Median can be particularly slow. Use Gaussian or Mean smoothing instead, if possible.

Application Warm-Up (Advanced)

An important practical issue in industrial applications with triggered cameras is that the first iteration of a program often needs to be executed at full speed. However, there are additional computations performed in the first iterations that must be taken into account:

- Memory buffers (especially images) for output data are allocated.
- Memory buffers get loaded into the cache memory.
- External DLL libraries get delay-loaded by the operating system.
- Modern CPU mechanisms, like branch prediction, get trained.
- Connections with external devices (e.g. cameras) get established.
- Some filters, especially those from 1D Edge Detection and Shape Fitting, precompute some data.

These issues arise from both the simplified data-flow programming model, as well as from the modern architectures of computers and operating systems. Some, but not all, of them can be solved with the use of the Aurora Vision Library (see: When to use Aurora Vision Library?). There is however, an idiom that might be useful also with Aurora Vision Studio – it is called "Application Warm-Up" and consists in performing one or a couple of iterations on test images (recorded) before the application switches to the operational stage. This can be achieved with the following "GrabImage" variant macrofilter:



An example "GrabImage" macrofilter designed for application warming-up.

The "GrabImage" variant macrofilter shown above is an example of how application warm-up can be achieved. It starts its operation in the "WarmUp" variant, where it initializes the camera and produces a test image loaded from a file (which has exactly the same resolution and format as the images acquired from the camera). Then it switches to the "Work" variant, where the standard image acquisition filter is used. There is also an additional **outIsWarmingUp** output that can be used, for example, to suppress the output signals during the warm-up stage.

Configuring Parallel Computing

The filters of Aurora Vision Studio internally use multiple threads to utilize the full power of multi-core processors. By default, they use as many threads as there are physical processors. This is the best setting for the majority of applications, but in some cases, a different number of threads might result in faster execution. If you need maximum performance, it is advisable to experiment with the ControlParallelComputing filter with both higher and lower number of threads. In particular:

- If the number of threads is **higher** than the number of physical processors, then it is possible to utilize the Hyper-Threading technology.
- If the number of threads is **lower** than the number of physical processors (e.g., 3 threads on a quad-core machine), then the system has at least one core available for background threads (like image acquisition, GUI, or computations performed by other processes), which may improve its responsiveness.

Configuring Image Memory Pools

Memory allocation is among the significant factors affecting filter performance. Most of the filters available in Aurora Vision Studio re-use their memory buffers between consecutive iterations, which is highly beneficial for their performance. Some filters, however, still allocate temporary image buffers, because doing otherwise would make them less convenient in use. To overcome this limitation, there is the filter ControlImageMemoryPools which can turn on a custom memory allocator for temporary images.

There is also a way to pre-allocate image memory before the first iteration of the program starts. For this purpose, use the InspectImageMemoryPools filter at the end of the program, and – after a the program is executed – copy its **outPoolSizes** value to the input of a ChargeImageMemoryPools filter executed at the beginning. In some cases, this will improve the performance of the first iteration.

Using GPGPU/OpenCL Computing

Some filters in Aurora Vision Studio allow computations to be moved to an OpenCL-capable device, like a graphics card, in order to speed up execution. After proper initialization, OpenCL processing is performed completely automatically by suitable filters without changing their use pattern. Refer to "Hardware Acceleration" section of the filter documentation to find which filters support OpenCL processing and what are their requirements. Be aware that the resulting performance after switching to an OpenCL device may vary and may not always be a significant improvement relative to CPU processing. Actual performance of the filters must always be verified on the target system by proper measurements.

To use OpenCL processing in Aurora Vision Studio, the following is required:

- a processing device installed in the target system supporting OpenCL C language version 1.1 or greater,
- a proper and up-to-date device driver installed in the system,
- proper OpenCL runtime software provided by its vendor.

OpenCL processing is supported, for example, in the following filters: RgbToHsi, HsiToRgb, ImageCorrelationImage, DilateImage_AnyKernel.

To enable OpenCL processing in filters, an InitGPUProcessing filter must be executed at the beginning of a program. Please refer to that filter documentation for further information.

When to use Aurora Vision Library?

Aurora Vision Library is a separate product for C++ programmers. The performance of the functions it provides is roughly the same as of the filters provided by Aurora Vision Studio. There are, however, some important cases when the overall performance of the compiled code is better.

Case 1: High number of simple operations

There is an overhead of about 0.004 ms on each filter execution in Studio. That value may seem minimal, but if we consider an application that analyzes 50 blobs in each iteration and executes 20 filters for each blob, it may sums up to a total of 4 ms. This may already be significant. If this is only a small part of a bigger application, then User Filters might be the right solution. If, however, this is how the entire application works, then the library should be used instead.

Case 2: Memory re-use for big images

Each filter in Aurora Vision Studio keeps its output data on the output ports. Consecutive filters do not re-use this memory but instead create new data. This is very convenient for effective development of algorithms as the user can see all intermediate results. However, if the application performs complex processing of very big images (e.g. from 10 megapixel or line-scan cameras), then the issue of memory re-use might become critical. Aurora Vision Library may then be useful, because only at the level of C++ programming the user can have the full control over the memory buffers.

Aurora Vision Library also makes it possible to perform in-place data processing, i.e., modifying the input data directly instead of creating new objects. Many simple image processing operations can be

performed in this way. Especially the Image Drawing functions and image transformations in small regions of interest may get a significant performance boost.

Case 3: Initialization before first iteration

Filters of Aurora Vision Studio get initialized in the first iteration. This is, for example, when the image memory buffers are allocated, because before the first image is acquired, the filters do not know how much memory they will need. Sometimes, however, the application can be optimized for specific conditions, and it is important that the first iteration is not any slower. On the level of C++ programming this can be achieved with preallocated memory buffers and with separated initialization of some filters (especially for 1D Edge Detection and Shape Fitting filters, as well as for image acquisition and I/O interfaces). See also: Application Warm-Up.

Understanding OrNil Filter Variants

What are OrNil filter variants

Filter variants distinguish various approaches to a single task and are packed into a single task related group. Most of the time filter variants will use a distinct algorithm or a different set of input data to achieve the same result at the end. In case of *OrNil* filter variants the difference lies in the error handling output data type. *OrNil* filter variants are available when a filter expects a non-empty data collection on input and make it possible to skip the execution instead of stopping it with a Domain Error in the Unsafe variant. When this happens all affected outputs are set to the Nil value.

Why were OrNil filter variants introduced

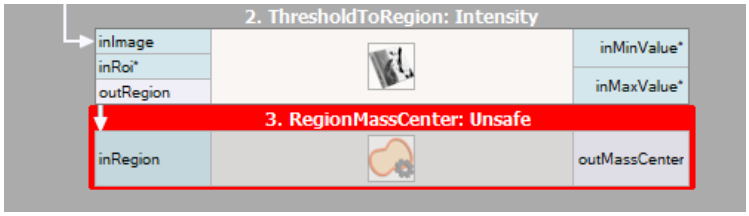
Filters like RegionMassCenter, GetArrayElement or PathBoundingRectangle expect a non-empty data collection on input in order to extract certain attributes based on the values. When the collection is empty and the Unsafe variant is used, the execution is stopped with a Domain Error. If it is beneficial to simply skip such operation instead of stopping the program, SkipEmpty(Collection) filters can be used to convert execution to conditional processing and skip it entirely when the collection is empty. *OrNil* filter variants are a natural progression as they require less work and provide the same capability.

When should OrNil filter variants be used

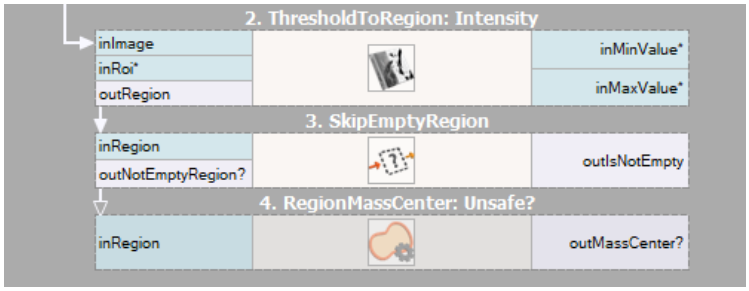
Use *OrNil* filter variants when there's a need to skip the execution after invalid input data was fed to a filter instead of stopping it.

When do OrNil filter variants skip the execution

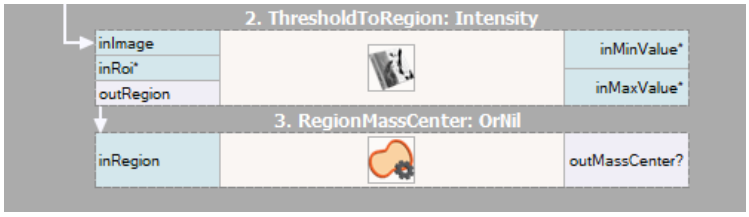
The conditions vary depending on the filter. Most of the time the main condition is an empty data collection on input. Some *OrNil* filter variants may have additional conditions described on their help pages. Do note though that OrNil filter variants will still throw Domain Error exceptions if the values provided are outside of the domain, e.g. negative array index.



RegionMassCenter throwing a Domain Error on empty input region. The program execution is stopped.



RegionMassCenter not being executed due to conditional processing introduced with SkipEmptyRegion. Program execution continues.



RegionMassCenter in the OrNil variant safely executed and introduces conditional processing on its output. Program execution continues.

Working with XML Trees

Table of contents

- Introduction
- Accessing XML Nodes
- Creating an XML Document
- Selecting Elements Using XPath Query
- Common Practices (tips)

Introduction

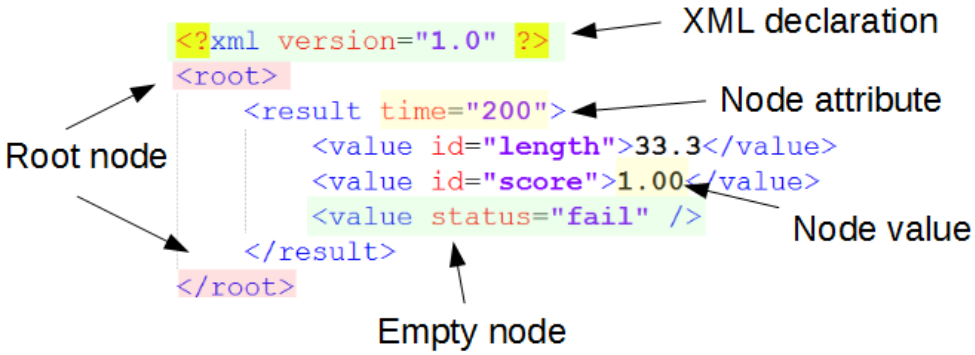
Extensible Markup Language (XML) is a markup language used for storing data in human readable text-format files. XML standard is very popular in exchanging data between two systems. XML provides a clear way to create text files which can be readable both by a computer application and a human.

An XML file is organized into a tree structure. The basic element of this tree structure is called a node. A node has a name, attributes and can have a text value. The example below shows a node with the name 'Value' and with two attributes 'a', 'b'. The example node contains text '10.0'.

```
<Value a="1.0" b="2.0">10.0</Value>
```

An XML node can also contain other nodes which are called children. The example below shows a node named 'A' with two children: 'B' and 'C'.

```
<A>
  <B />
  <C />
</A>
```



Descriptions of XML elements.

The XML technology is commonly used for:

- Configuration files
- Data exchange format between applications
- Structured data sets
- Simple file-based databases

To load or store data in the XML format, two filters are necessary: Xml_LoadFile and Xml_SaveFile. Also two conversions are possible: XmlNodeToString and StringToXmlNode. These filters are commonly used to perform operations on data received from different sources like Serial Port or TCP/IP.

Accessing XML Nodes

After loading an XML file all node properties can be accessed with AccessXmlNode filter.

Xml_GetChildNode and filters from group Xml_GetAttribute are used to access directly a node's element using its known number or name.

To perform more complex access operations on elements using specified criteria, please read: Selecting

Elements Using XPath Query.

Creating an XML Document

The process of creating XML documents is organized in a bottom-up approach. At the beginning, the lowest nodes in the tree must be created. Then, such nodes can be added to a node at a higher level in the tree and that node can be added to some yet higher node and so on. The node without a parent which is the highest level in hierarchy is called root.

To create an XML node Xml_CreateNode filter should be used. Then, the newly created node should be added to the root node. The new node can have text value assigned.

Created XML nodes should be added to the tree using Xml_AddChildNodes and Xml_AddChildNodes_OfArray filters. New nodes will be added to the end of the list of children.

Also attributes can be added to a node using Xml_SetAttributes filter.

Selecting Elements Using XPath Query

To perform other operations on more complex XML trees filters from group Xml_SelectNodeValues can be used. Modification of the node's properties can be also done by using Xml_SetAttributes and Xml_SetNodeValues filters.

Selecting and altering filters is done using XPath criteria. XPath is a query language used to selecting sub-trees and attributes from an XML tree. All selecting and setting filters that are using XPath support the XPath 1.0 Specification.

The table below shows basic examples of XPath usage of filter Xml_SelectMultipleNodes:

Example	XPath	Description
<?xml version="1.0" ?> <root> I<result time="200"> II<value type="length">33.3</value> II<value type="score">1.00</value> II<value status="fail" /> I</result> I<result time="250"> II<value type="length">10.6</value> II<value type="score">0.30</value> II<value status="pass" /> I</result> </root>	**/root**	Selecting all nodes of type 'root' that are at the first level of the tree.
<?xml version="1.0" ?> <root> I<result time="200"> II<value type="length">33.3</value> II<value type="score">1.00</value> II<value status="fail" /> I</result> I<result time="250"> II<value type="length">10.6</value> II<value type="score">0.30</value> II<value status="pass" /> I</result> </root>	**/root/result[2]**	Selecting the second child of 'root' node which is of type 'result'.
<?xml version="1.0" ?> <root> I<result time="200"> II<value type="length">33.3</value> II<value type="score">1.00</value> II<value status="fail" /> I</result> I<result time="250"> II<value type="length">10.6</value> II<value type="score">0.30</value> II<value status="pass" /> I</result> </root>	**/root/result/value**	Selecting all children of type 'value' of parent nodes '/root/result' of a specified XML node.

<?xml version="1.0" ?> <root> |<result time="200"> ||<value type="length">33.3</value> ||<value type="score">1.00</value> ||<value status="fail" /> |</result> |<result time="250"> ||<value type="length">10.6</value> ||<value type="score">0.30</value> ||<value status="pass" /> |</result> </root>	**/root/result[2]/value[2]**	Selecting specified second child of type 'root' and the second child of 'result'.
<?xml version="1.0" ?> <root> |<result time="200"> ||<value type="length">33.3</value> ||<value type="score">1.00</value> ||<value status="fail" /> |</result> |<result time="250"> ||<value type="length">10.6</value> ||<value type="score">0.30</value> ||<value status="pass" /> |</result> </root>	**//value**	Selecting all value nodes regardless of their position in the XML tree.
<?xml version="1.0" ?> <root> |<result time="200"> ||<value type="length">33.3</value> ||<value type="score">1.00</value> ||<value status="fail" /> |</result> |<result time="250"> ||<value type="length">10.6</value> ||<value type="score">0.30</value> ||<value status="pass" /> |</result> </root>	**//*[@status]**	Selecting all nodes which have 'status' attribute name.
<?xml version="1.0" ?> <root> |<result time="200"> ||<value type="length">33.3</value> ||<value type="score">1.00</value> ||<value status="fail" /> |</result> |<result time="250"> ||<value type="length">10.6</value> ||<value type="score">0.30</value> ||<value status="pass" /> |</result> </root>	**//*[@status='fail']**	Selecting all nodes which have 'status' of value 'fail'.
<?xml version="1.0" ?> <root> |<result time="200"> ||<value type="length">33.3</value> ||<value type="score">1.00</value> ||<value status="fail" /> |</result> |<result time="250"> ||<value type="length">10.6</value> ||<value type="score">0.30</value> ||<value status="pass" /> |</result> </root>	**//*[starts-with(name(),'res') and (@time > 200)]**	Selecting all nodes with names starting with 'res' and having attribute 'time' greater than '200'.
<?xml version="1.0" ?> <root> |<result time="200"> ||<value type="length">33.3</value> ||<value type="score">1.00</value> ||<value status="fail" /> |</result> |<result time="250"> ||<value type="length">10.6</value> ||<value type="score">0.30</value> ||<value status="pass" /> |</result> </root>	**//*[contains(text(),'10')]**	Selecting all nodes whose text (value) contains '10'.

Notice: All indexes are counted starting from [1].

The table below shows basic examples of XPath usage of Xml_SelectMultipleNodeValues_AsStrings filter:

Example	XPath	Description
<?xml version="1.0" ?> <root> I<result time="200"> II<value type="length">33.3</value> II<value type="score">1.00</value> II<value status="fail" /> I</result> I<result time="250"> II<value type="length">10.6</value> II<value type="score">0.30</value> II<value status="pass" /> I</result> </root>	**/root[1]/result[2]/value[2]**	Selecting value from a specified node.
<?xml version="1.0" ?> <root> I<result time="200"> II<value type="length">33.3</value> II<value type="score">1.00</value> II<value status="fail"> </value> I</result> I<result time="250"> II<value type="length">10.6</value> II<value type="score">0.30</value> II<value status="pass" /> I</result> </root>	**/root[1]/result[1]/value**	Get all node values from a specific node. *Notice that empty XML nodes empty contains a default type value.*
<?xml version="1.0" ?> <root> I<result time="200"> II<value type="length">33.3</value> II<value type="score">1.00</value> II<value status="fail"> </value> I</result> I<result time="250"> II<value type="length">10.6</value> II<value type="score">0.30</value> II<value status="pass" /> I</result> </root>	**/root[1]/result/value[@type='length']**	Get values of all nodes with attribute **length**.
<?xml version="1.0" ?> <root> I<result time="200"> II<value type="length">33.3</value> II<value type="score">1.00</value> II<value status="fail"> </value> I</result> I<result time="250"> II<value type="length">10.6</value> II<value type="score">0.30</value> II<value status="pass" /> I</result> </root>	**/root[1]/result/value[@type='length']|/root[1]/result/value[@type='score']**	Get values of all nodes with attribute **length** and nodes with **score** attribute.

The table below shows basic examples of XPath usage of Xml_SelectMultipleAttributes_AsStrings filter:

Example	XPath	Description
<?xml version="1.0" ?> <root> I<result time="200"> II<value type="length">33.3</value> II<value type="score">1.00</value> II<value status="fail" /> I</result> I<result time="250"> II<value type="length">10.6</value> II<value type="score">0.30</value> II<value status="pass" /> I</result> </root>	**//*/@type**	Selecting all attributes named 'type' .
<?xml version="1.0" ?> <root> I<result time="200"> II<value type="length">33.3</value> II<value type="score">1.00</value> II<value status="fail" /> I</result> I<result time="250"> II<value type="length">10.6</value> II<value type="score">0.30</value> II<value status="pass" /> I</result> </root>	**//*[./*[@status='fail']]/@time**	Selecting all attributes 'time' of nodes whose one of the children has status fail.

Common Practices (tips)

- Creating an XML tree by appending new nodes is not always necessary. If the tree structure is constant, the whole XML tree can be stored in a Global Parameter and when it is necessary new values of attributes or node text can be set using Xml_SetAttributes and Xml_SetNodeValues filters.
- XML files are very handy to store program settings.
- Most of Aurora Vision Studio objects can be serialized to text and stored in an XML file.
- XmlNodeToString filter can be used to send XML via TCP/IP or Serial Port.

Zebra
Aurora™ Vision

This article is valid for version 5.7.5

Legal Terms of Use Privacy Policy

ZEBRA and the stylized Zebra head are trademarks of Zebra Technologies Corp., registered in many jurisdictions worldwide. All other trademarks are the property of their respective owners. ©2026 Zebra Technologies Corp. and/or its affiliates.