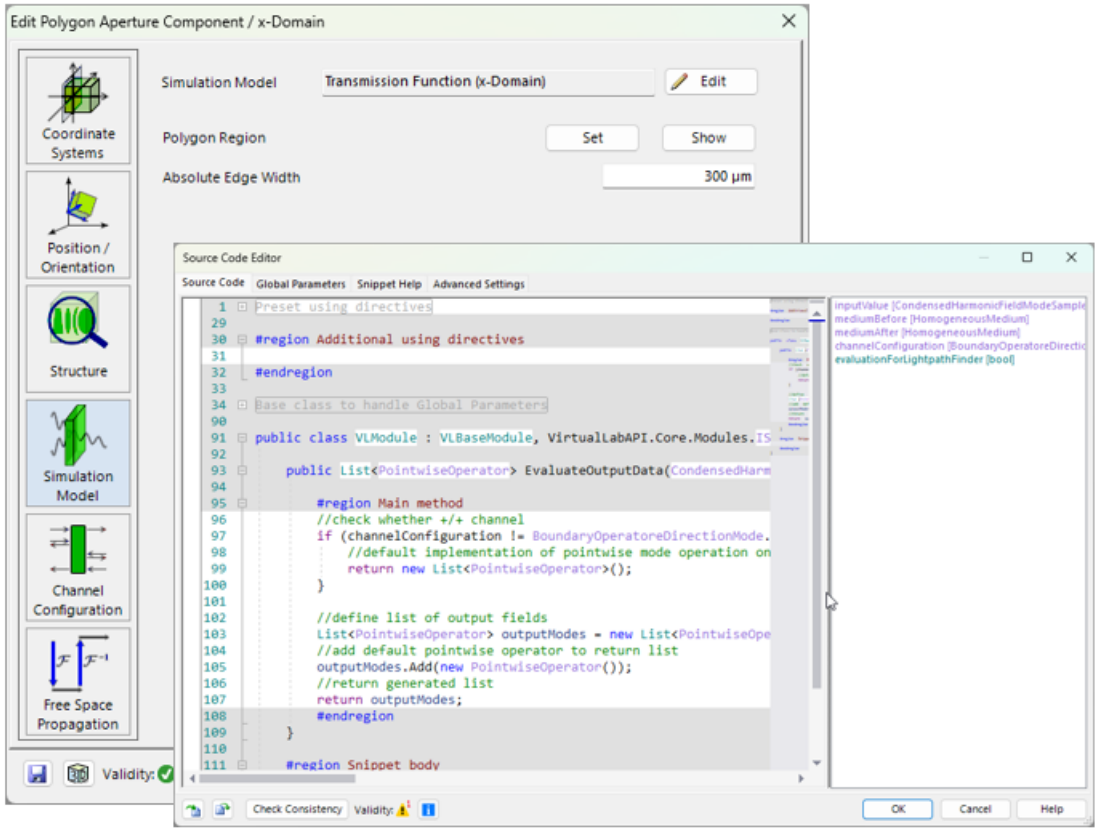
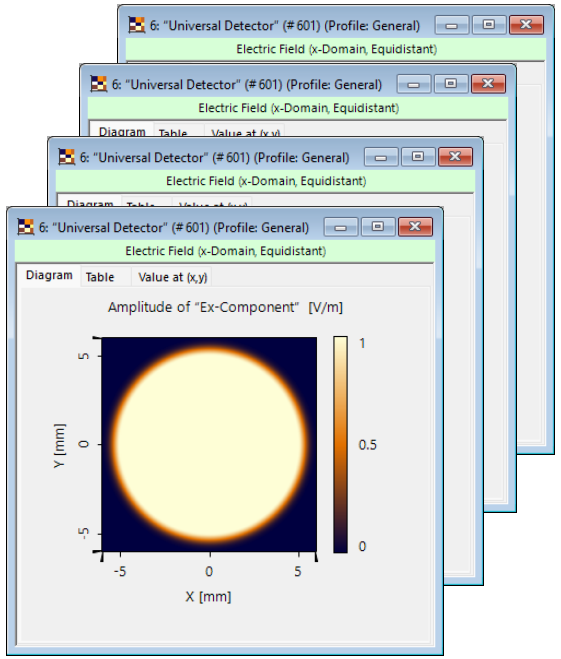
Plug-In Component

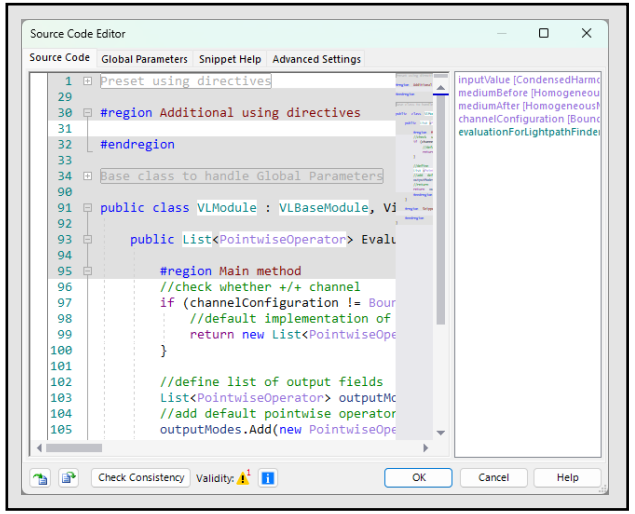
Abstract



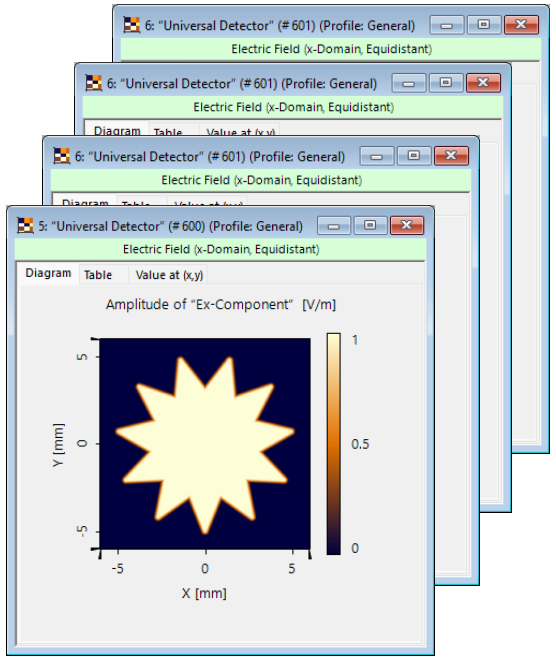
Certain specialized systems or simulation tasks may require the use of a user-defined solver. This tutorial introduces the Plug-in component, which enables users to develop and implement a fully customized solver in both the x-domain and k-domain.

Principle of the Plug-In Component





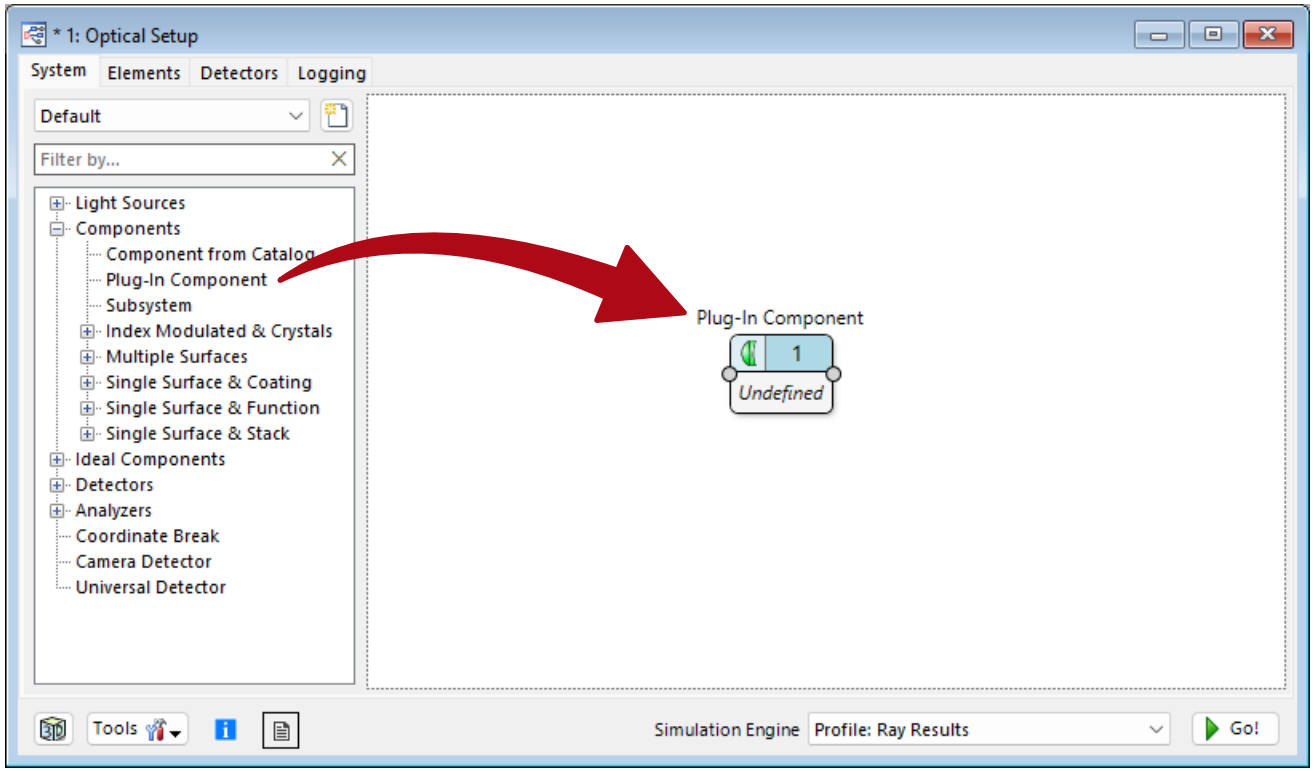
Programmable operator



Input modes (e.g. different wavelengths, incoherent/coherent modes, …)

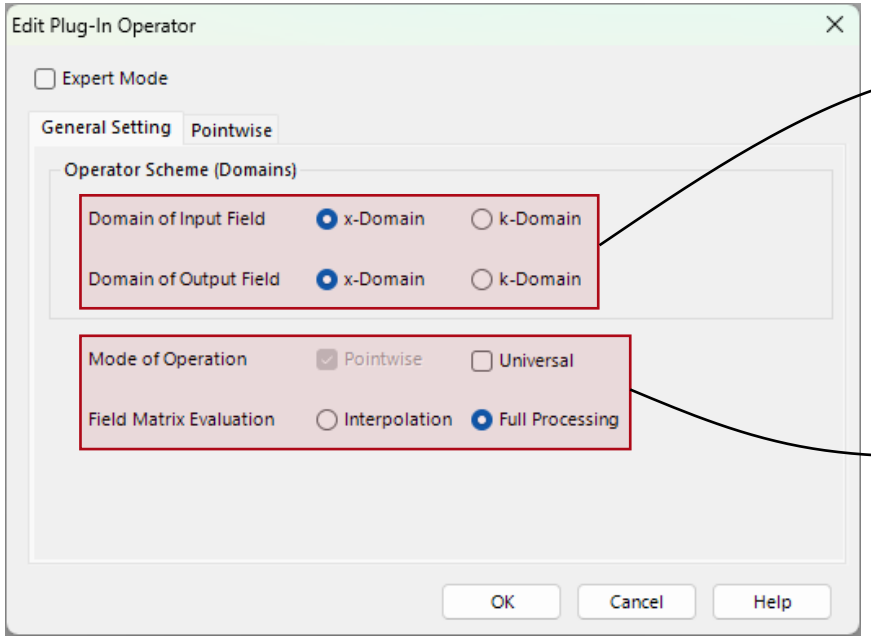
Output modes (adjusted amplitude & phase per mode)

Where to Find the Component



The *Plug-in Component* can be found under Components/Plug-In Component in *the Optical Setup* document.

Plug-In Component Options

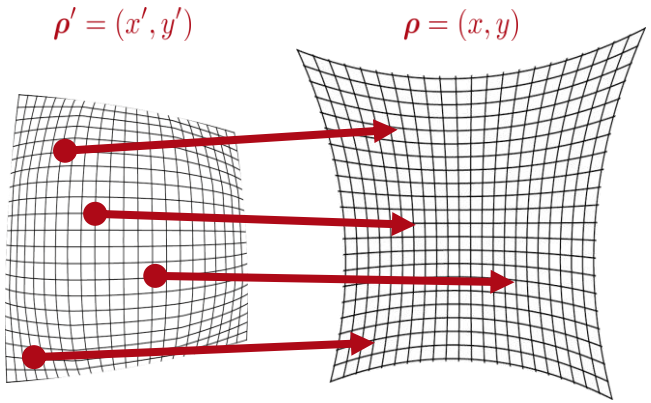


The *Plug-in* component accepts input modes in both the x-domain and k-domain and can produce output in either domain as well. It also supports mixed configurations—for example, a component can be designed to accept x-domain input modes and generate k-domain output modes.

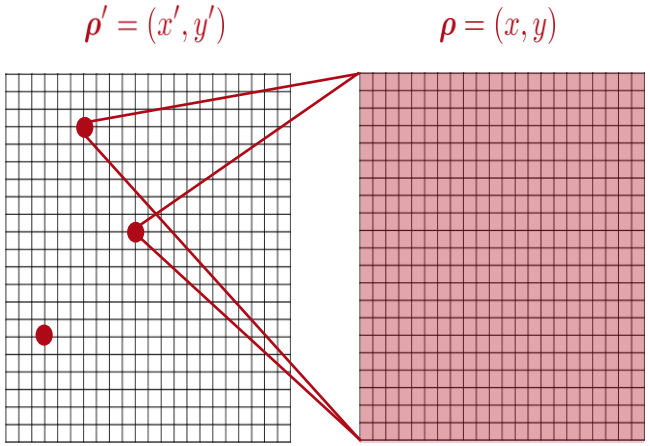
Additionally, users can specify whether the custom operator acts pointwise or globally. Physically, this defines whether each input point affects only a single output point or contributes to all outputs. This choice primarily influences how internal code snippets are structured. Note that a single component may combine both types—for example, the Light Path Finder always utilizes pointwise operators, even if the solver itself is universal.

Note: As a rule of thumb pointwise operators are used when diffraction effects are ignored and universal when they are included.

See here for more information:
Free Space Propagation Settings

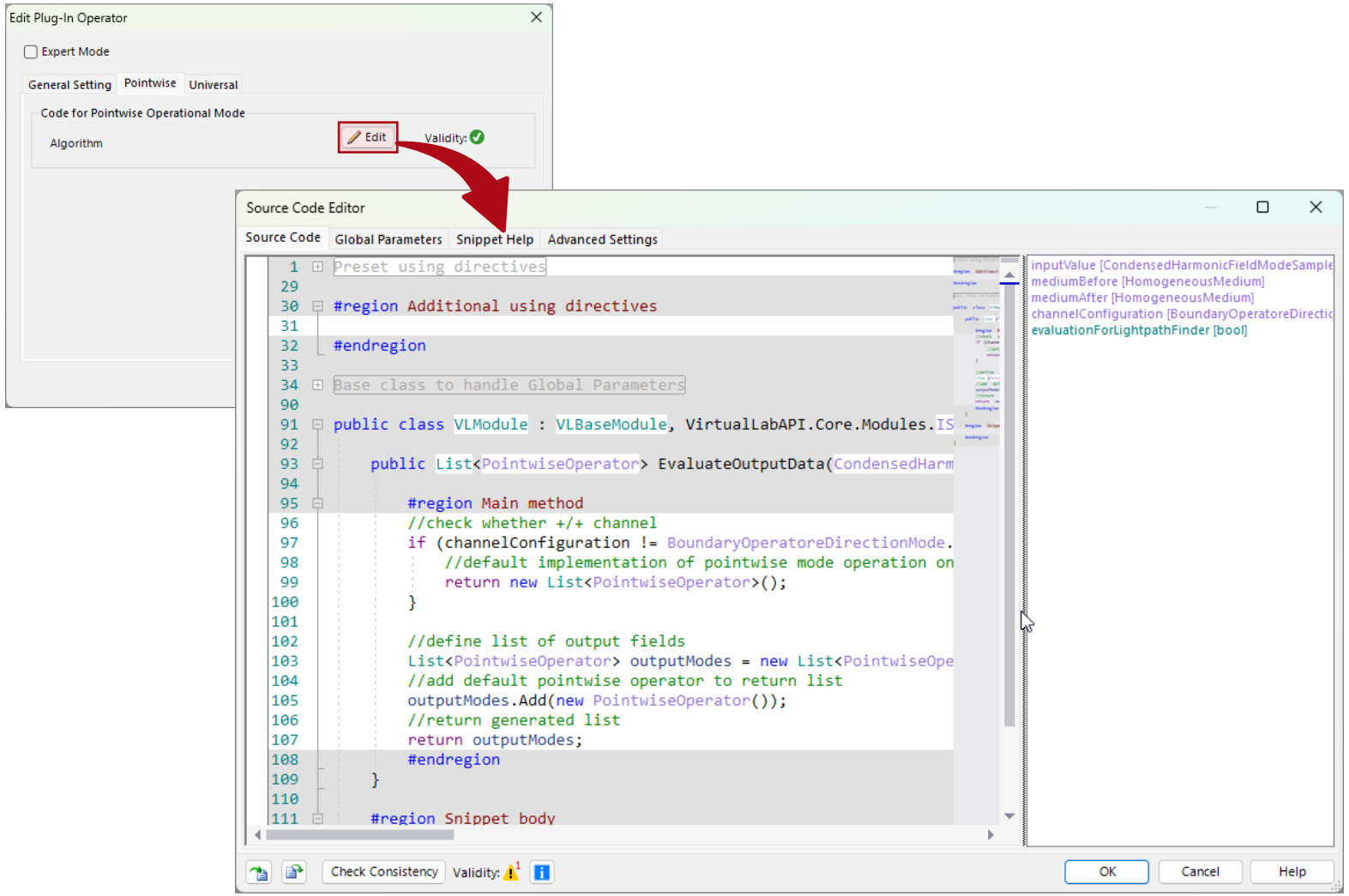


Pointwise operator



Universal operator

Programming the Operator



Access to the relevant snippet is available under the *Pointwise* or *Universal* tab, which appear only when the corresponding option is enabled.

For a detailed tutorial on working with VirtualLab Fusion snippets, please refer to:

Programming Detector Add-ons

The following pages provide a brief overview of the key features and specific capabilities of this component.

Snippet for a Pointwise Operator

Source Code Editor

Source Code | Global Parameters | Snippet Help | Advanced Settings

```
1    Preset using directives
29
30   #region Additional using directives
31
32   #endregion
33
34   Base class to handle Global Parameters
90
91   public class VLModule : VLBaseModule, VirtualLabAPI.Core.Modules.ISnippetListPointwiseOperator_Condens
92
93       public List<PointwiseOperator> EvaluateOutputData(CondensedHarmonicFieldModeSample inputValue, Hor
94
95           #region Main method
96           //check whether +/+ channel
97           if (channelConfiguration != BoundaryOperatoreDirectionMode.PlusPlus) {
98               //default implementation of pointwise mode operation only supports +/+ channel, so we retu
99               return new List<PointwiseOperator>();
100          }
101
102          //define list of output fields
103          List<PointwiseOperator> outputModes = new List<PointwiseOperator>();
104          //add default pointwise operator to return list
105          outputModes.Add(new PointwiseOperator());
106          //return generated list
107          return outputModes;
108          #endregion
109      }
110
111      #region Snippet body
112
```

inputValue [CondensedHarmonicFieldModeSample]
mediumBefore [HomogeneousMedium]
mediumAfter [HomogeneousMedium]
channelConfiguration [BoundaryOperatoreDirectionMode]
evaluationForLightpathFinder [bool]

Check Consistency Validity:

Users can also define a special operator that only shall operate on the Light Path Finder.

For pointwise operators, VirtualLab Fusion automatically loops over all points. The user must specify a 2×2 response matrix, which is multiplied onto the input at each point by:

$$\begin{pmatrix} E_x^{out}(x_i, y_i) \\ E_y^{out}(x_i, y_i) \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} E_x^{in}(x_i, y_i) \\ E_y^{in}(x_i, y_i) \end{pmatrix}$$

We demonstrate a concrete example later in this tutorial.

Snippet for a Universal Operator

Source Code Editor

Source Code | Global Parameters | Snippet Help | Advanced Settings

```
1   Preset using directives
29
30  #region Additional using directives
31
32  #endregion
33
34  Base class to handle Global Parameters
90
91  public class VLModule : VLBaseModule, VirtualLabAPI.Core.Modules.ISnippetHarmonicFieldMode_HarmonicFiel
92
93      public List<HarmonicFieldMode> EvaluateOutputModes(HarmonicFieldMode inputMode, HomogeneousMedium m
94
95          #region Main method
96          //check whether +/+ channel
97          if (channelConfiguration != BoundaryOperatoreDirectionMode.PlusPlus) {
98              //default implementation of pointwise mode operation only supports +/+ channel, so we retur
99              return null;
100         }
101
102         //define list of output fields
103         List<HarmonicFieldMode> outputModes = new List<HarmonicFieldMode>();
104         //add input mode to list
105         outputModes.Add(inputMode);
106         //return generated list
107         return outputModes;
108         #endregion
109     }
110
111     #region Snippet body
112
113     #endregion
114 }
```

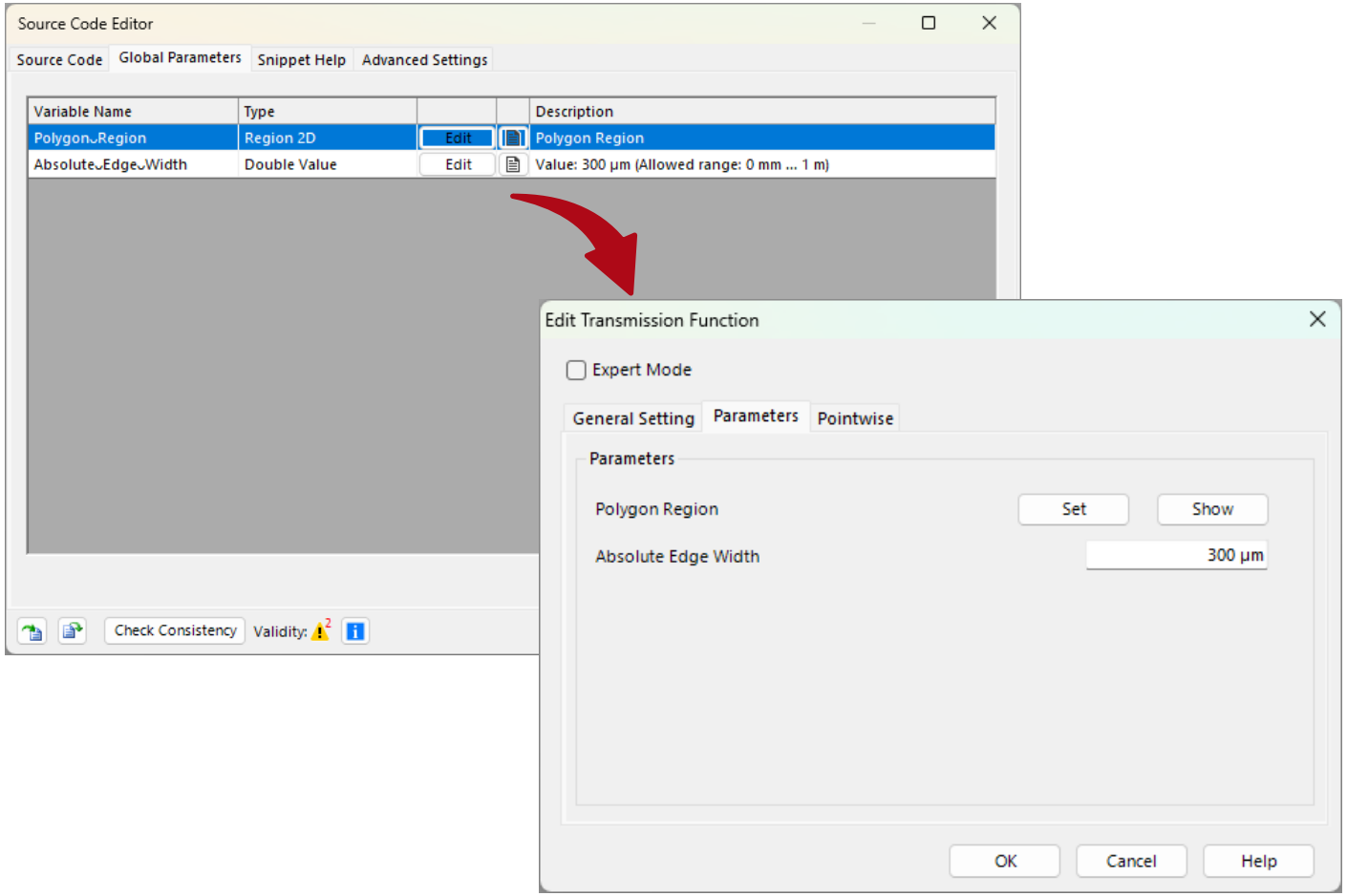
inputMode [HarmonicFieldMode]
mediumBefore [HomogeneousMedium]
mediumAfter [HomogeneousMedium]
channelConfiguration [BoundaryOperatoreDirectionMode]

Check Consistency Validity:

For Universal operaters, input and output are given as *HarmonicFieldMode*, which represent complete fields.

We demonstrate a concrete example later in this tutorial.

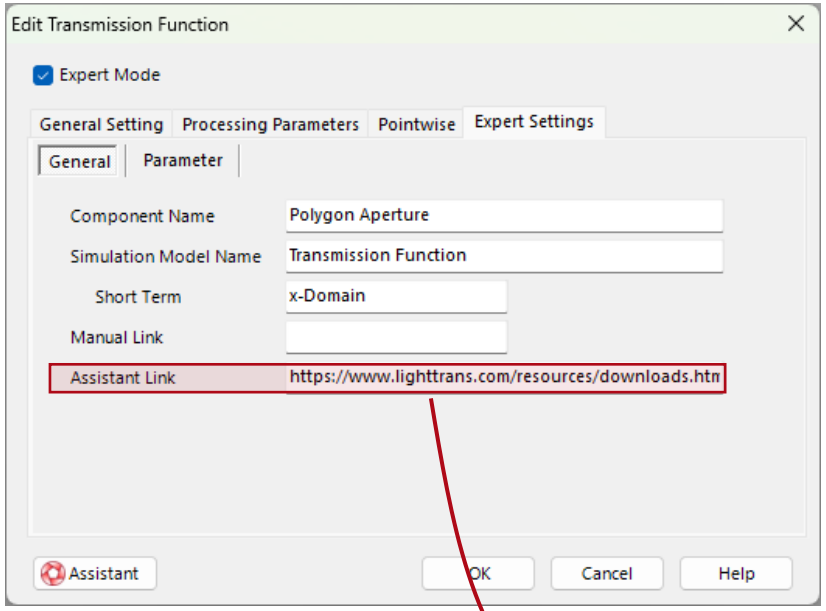
Defining Parameter



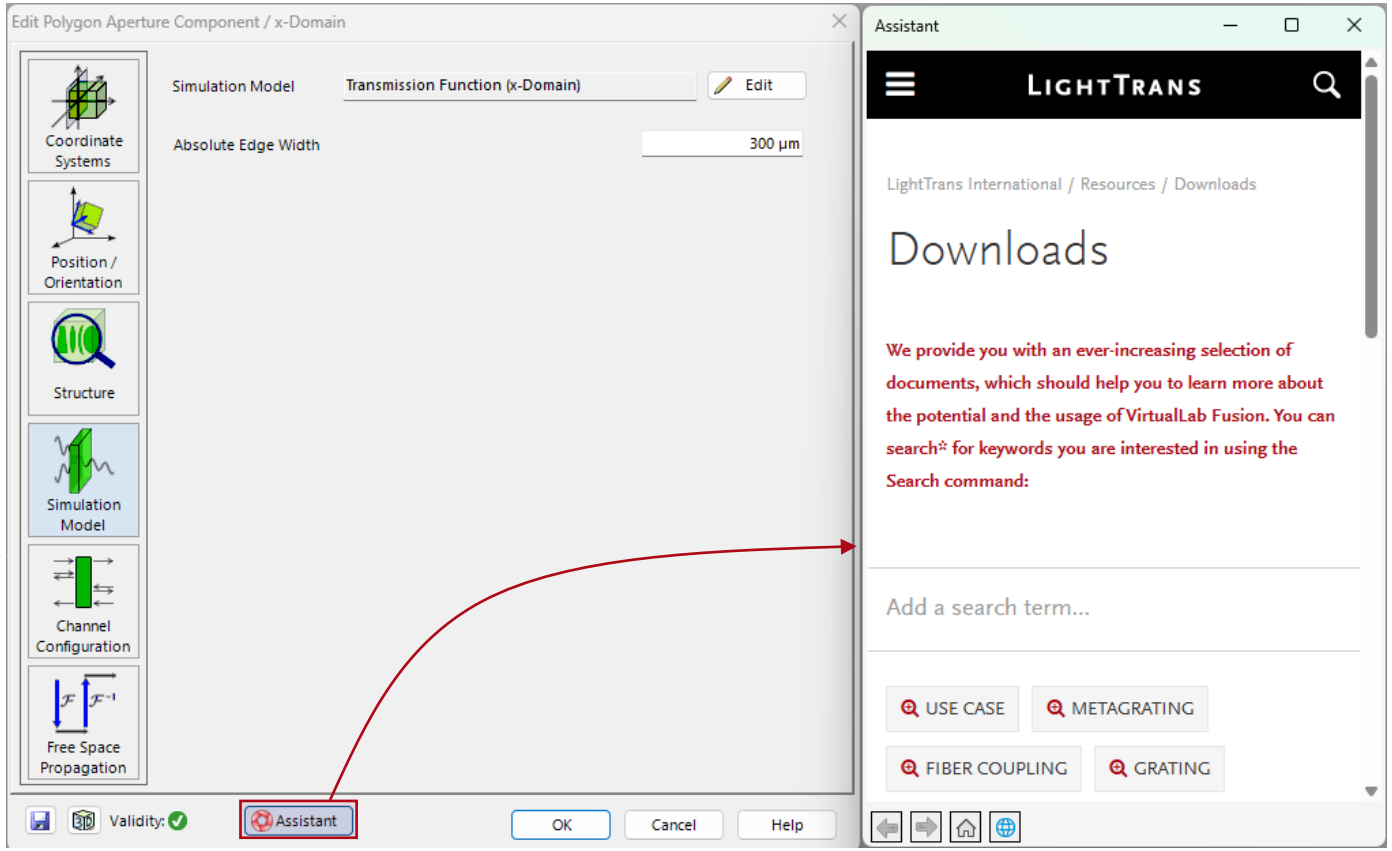
As with other programmable components, custom parameters can be defined within the programmable snippet.

In the *Plug-in* component, these parameters automatically appear under the *Parameters* tab, which becomes visible only when at least one custom parameter is present.

Expert Modus

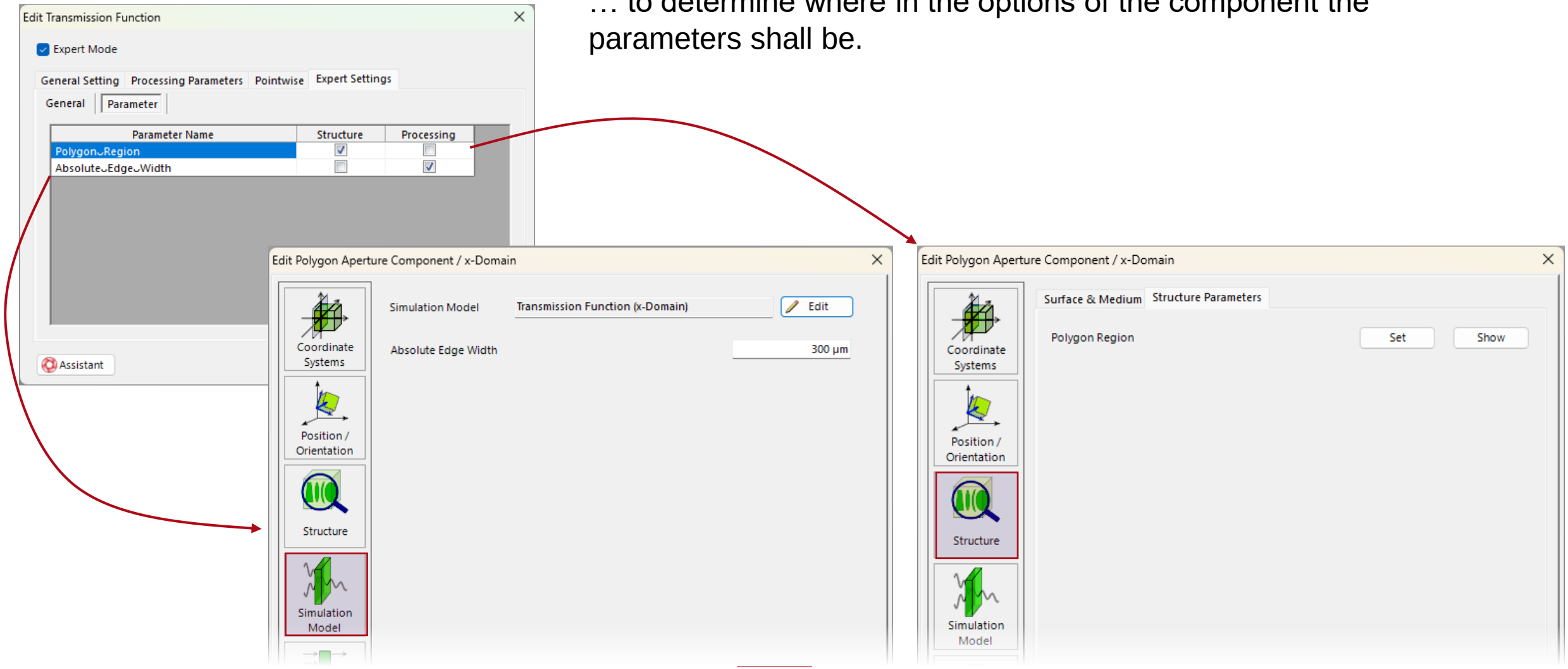


Activate the Export Mode for additional convenience options such a the possibility to a website with further explanations or ....

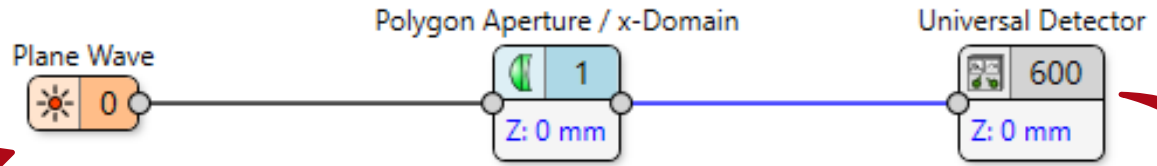


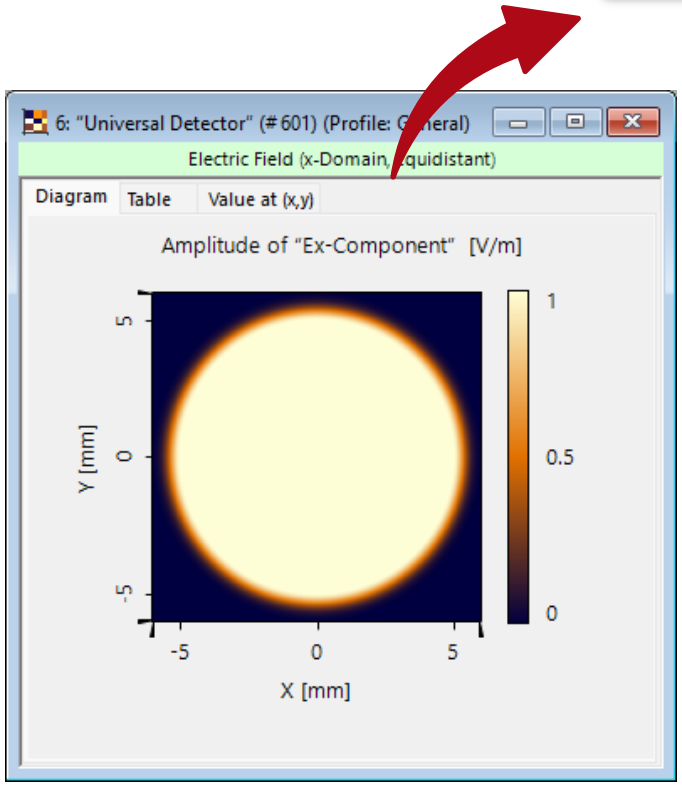
Expert Modus - Parameter

… to determine where in the options of the component the parameters shall be.

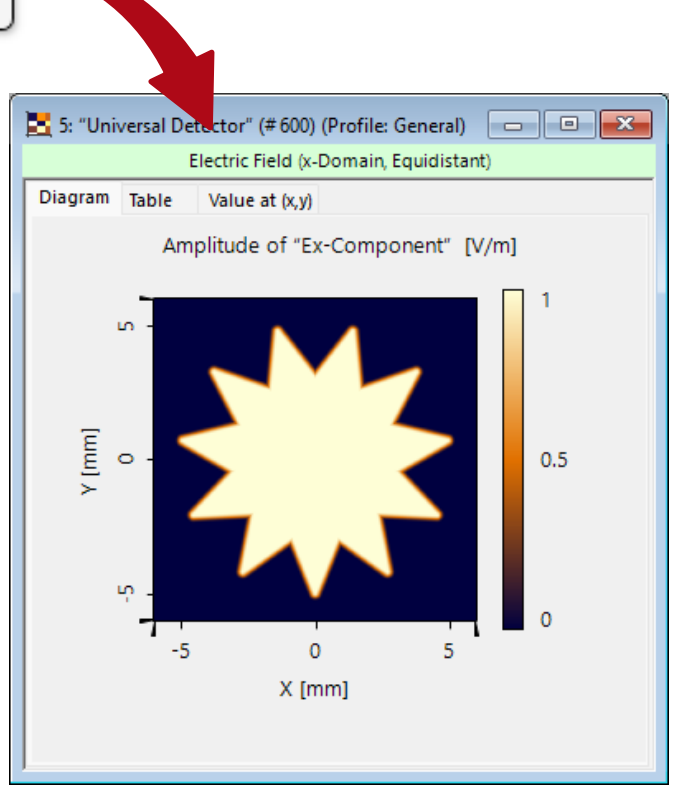


Example: Custom Aperture

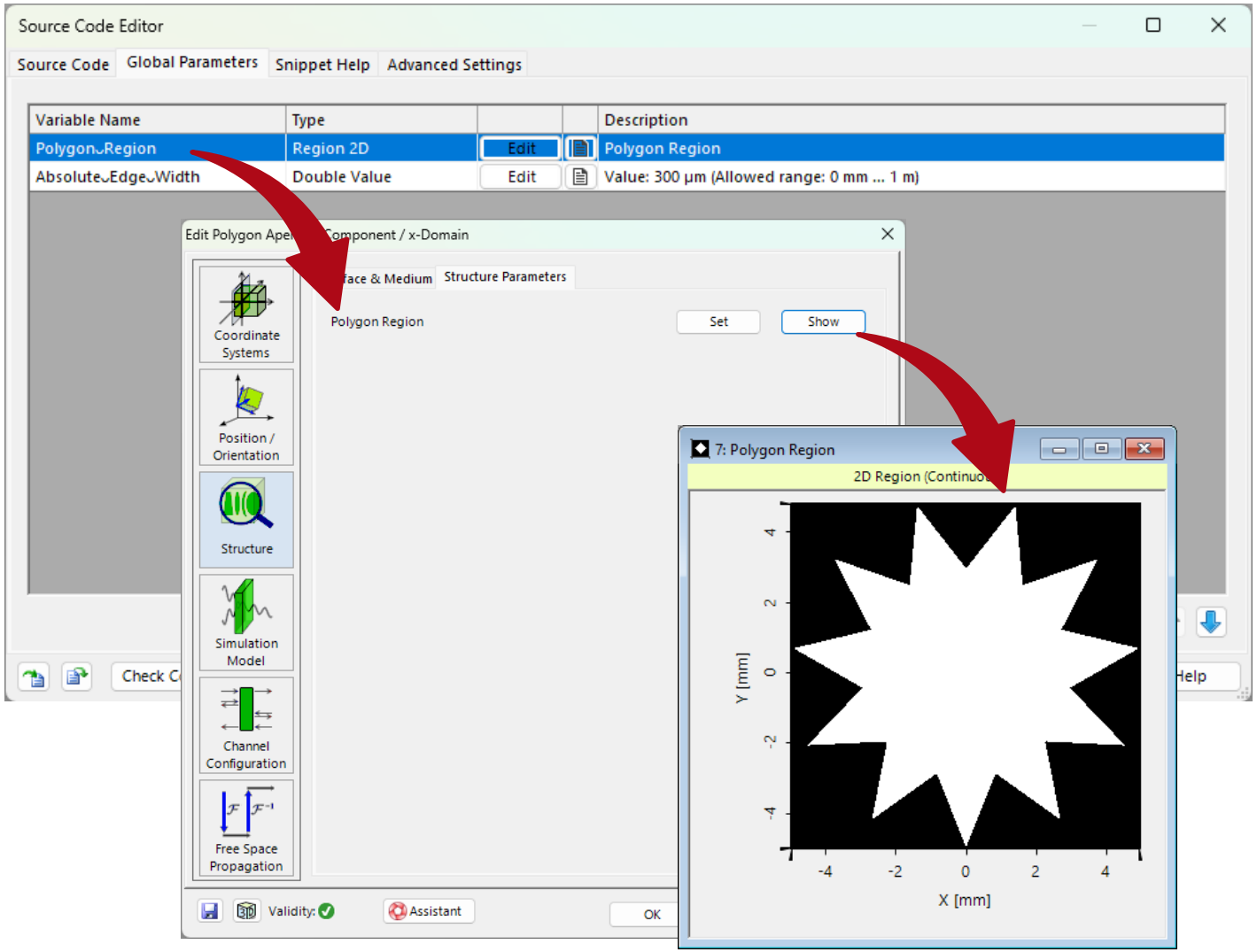




As an example we want to present a customizable aperture, e.g. a star-shaped one.



Custom Aperture: Parameter



We set up the aperture shape as a parameter. For our case we import a polygon-region. We also include a parameter to control the soft edge.

Star-Shaped Aperture: Code for Pointwise Operator

```
#region Main method
//check whether +/+ channel
if (channelConfiguration != BoundaryOperatoreDirectionMode.PlusPlus) {
    //default implementation of pointwise mode operation only supports +/+ channel, so we return empty list
    return new List<PointwiseOperator>();
}

//define list of output fields
List<PointwiseOperator> outputModes = new List<PointwiseOperator>();
//define new pointwise operator
PointwiseOperator apertureOperator = new PointwiseOperator();

//evaluate aperture function with smooth cosine edge
double apertureValueAtCurrentPosition =
VirtualLabAPI.Core.BasicFunctions.ApertureFunctions.ApertureFactorCosine(new VectorD(inputValue.Coordinate.X, inputValue.Coordinate.Y),
                                                                          PolygonRegion as SimplePolygon,
                                                                          AbsoluteEdgeWidth);

//define field matrix for aperture operator
apertureOperator.FieldMatrix = new Matrix2x2C(apertureValueAtCurrentPosition, 0, 0, apertureValueAtCurrentPosition);

//add default pointwise operator to return list
outputModes.Add(apertureOperator);
//return generated list
return outputModes;
#endregion
```

These line determine which channels the operator shall work on. In our case only transmission (+/+) is of importance so we set the rest to zero.

Initialize output container

Perform calculations. In our case we simply call a function that generates a aperture with a given shape and soft edges..

Set FieldMatrix. This is the response matrix per point of the field.

Star-Shaped Aperture: Code for Universal Operator

```
#region Main method
//check whether +/+ channel
if (channelConfiguration != BoundaryOperatoreDirectionMode.PlusPlus) {
    //default implementation of pointwise mode operation only supports +/+ channel, so we return empty list
    return null;
}
```

These line determine which channels the operator shall work on. In our case only transmission (+/+) is of importance so we set the rest to zero.

```
//define list of output fields
List<HarmonicFieldMode> outputModes = new List<HarmonicFieldMode>();
//create container for output field
HarmonicFieldMode outputMode = new HarmonicFieldMode(inputMode);
//extract components
ComplexField modeToEvaluateEx = inputMode.GetEquidistantExData();
ComplexField modeToEvaluateEy = inputMode.GetEquidistantEyData();
```

Initialize output container.

Extract fields.

```
//sampling parameters
double firstDataPointX = inputMode.CenterOfFieldData.X - inputMode.SamplingDistanceFieldData.X * inputMode.NumberOfSamplingPointsFieldData.X / 2;
double firstDataPointY = inputMode.CenterOfFieldData.Y - inputMode.SamplingDistanceFieldData.Y * inputMode.NumberOfSamplingPointsFieldData.Y / 2;
```

Calculate help variables for sampling.

```
//loop over all points
for (int x = 0; x < inputMode.NumberOfSamplingPointsFieldData.X; x++)
    for (int y = 0; y < inputMode.NumberOfSamplingPointsFieldData.Y; y++) {
        //calculate position
        VectorD position = new VectorD(firstDataPointX + inputMode.SamplingDistanceFieldData.X * x, firstDataPointY + inputMode.SamplingDistanceFieldData.Y * y);
        //evaluate aperture function with smooth cosine edge
        double apertureValueAtCurrentPosition = VirtualLabAPI.Core.BasicFunctions.ApertureFunctions.ApertureFactorCosine(position,
                                                                                                                        PolygonRegion as SimplePolygon,
                                                                                                                        AbsoluteEdgeWidth);

        //multiply aperture function on field
        modeToEvaluateEx[x, y] *= apertureValueAtCurrentPosition;
        modeToEvaluateEy[x, y] *= apertureValueAtCurrentPosition;
    }
```

Loop over all points, the aperture function is the same as in the pointwise example.

Multiply input field with aperture.

```
//overwrite field in output
outputMode.SetEquidistantExData(modeToEvaluateEx);
outputMode.SetEquidistantEyData(modeToEvaluateEy);
```

Overwrite field in the output with the newly calculated one.

```
//add input mode to list
outputModes.Add(outputMode);
//return generated list
return outputModes;
#endregion
```

Return field mode.

Document Information

Title	Plug-In Component
Document code	TUT.0458
Publication date	08.07.2025
Required packages	-
Software version	2025.1 (Build 1.172)*
Category	Tutorial
Further reading	- Programming Detector Add-ons - Free Space Propagation Settings

* *The files attached to this document require the specific version or later.*