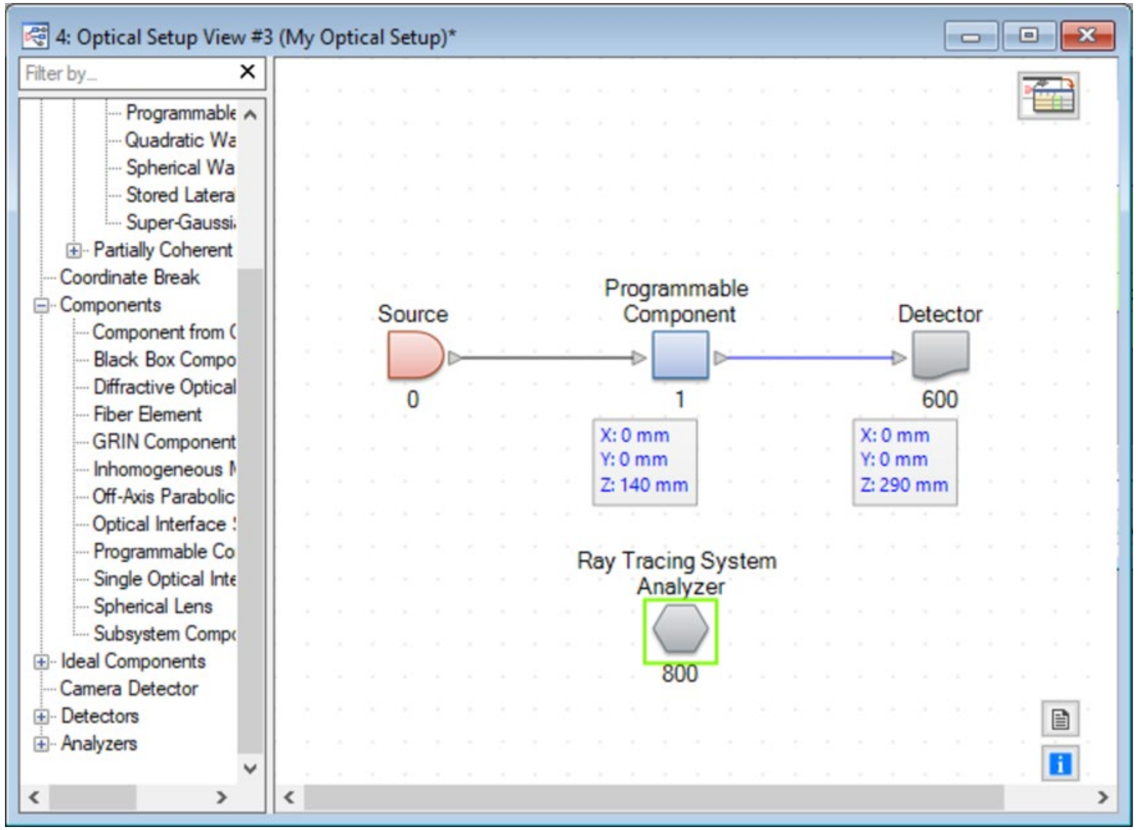
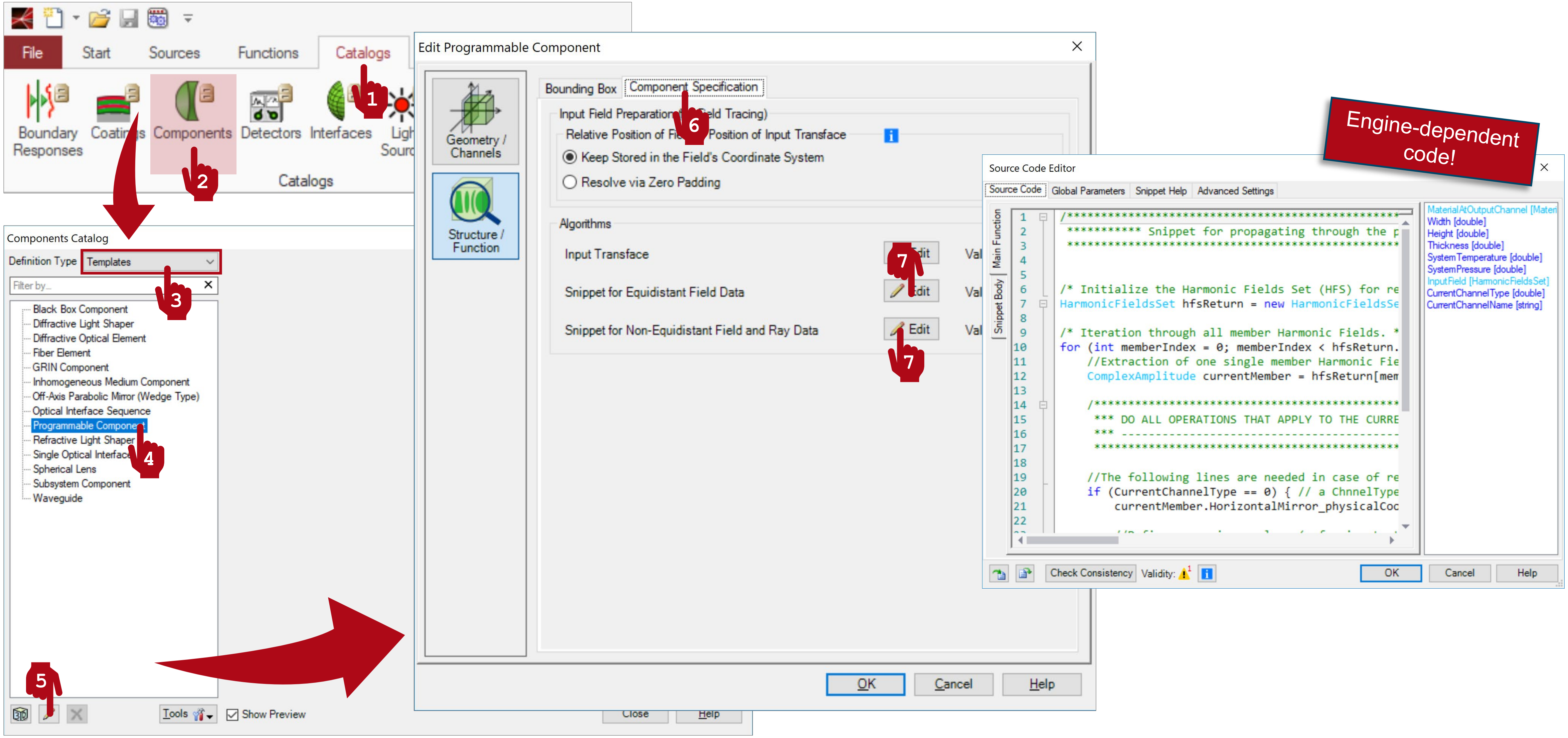
How to Work with the Programmable Component and Example (Ideal Grating)

Abstract

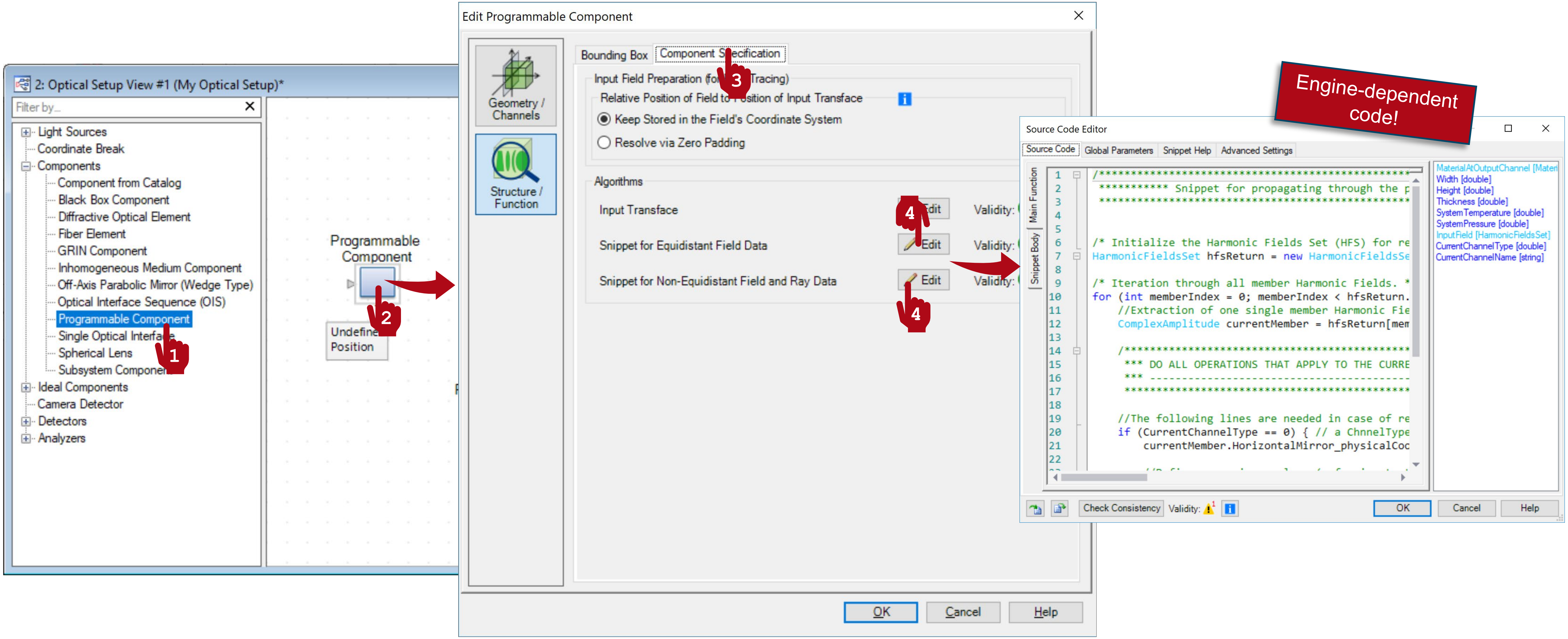


Providing maximum versality for your optical simulations is one of our most fundamental objectives. One of the most flexible representatives of this potential for customization in VirtualLab Fusion is the Programmable Component: a feature that allows you to freely transform the incoming light according to whatever model may be relevant for your application. You can subsequently, of course, combine your custom component with all the other capabilities already available in VirtualLab in order to construct a full optical system according to your requirements.

Where to Find the Programmable Component: Catalog

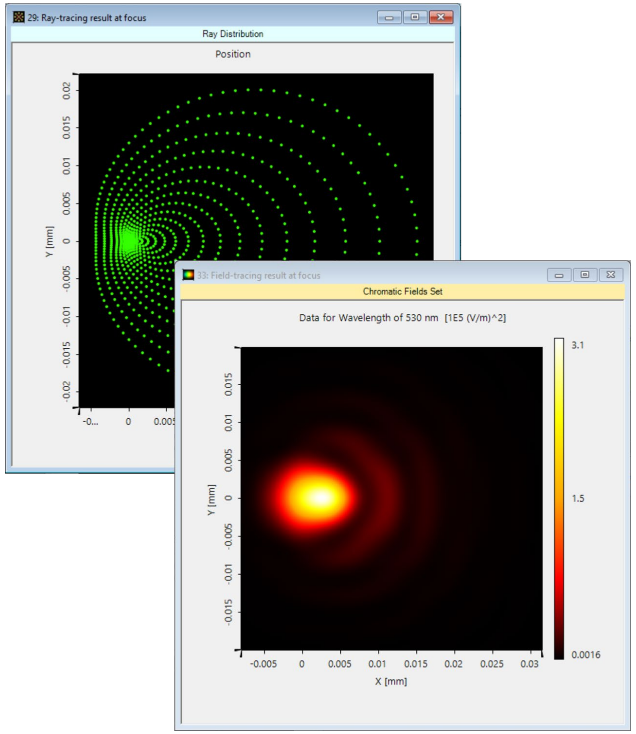


Where to Find the Programmable Component: Optical Setup

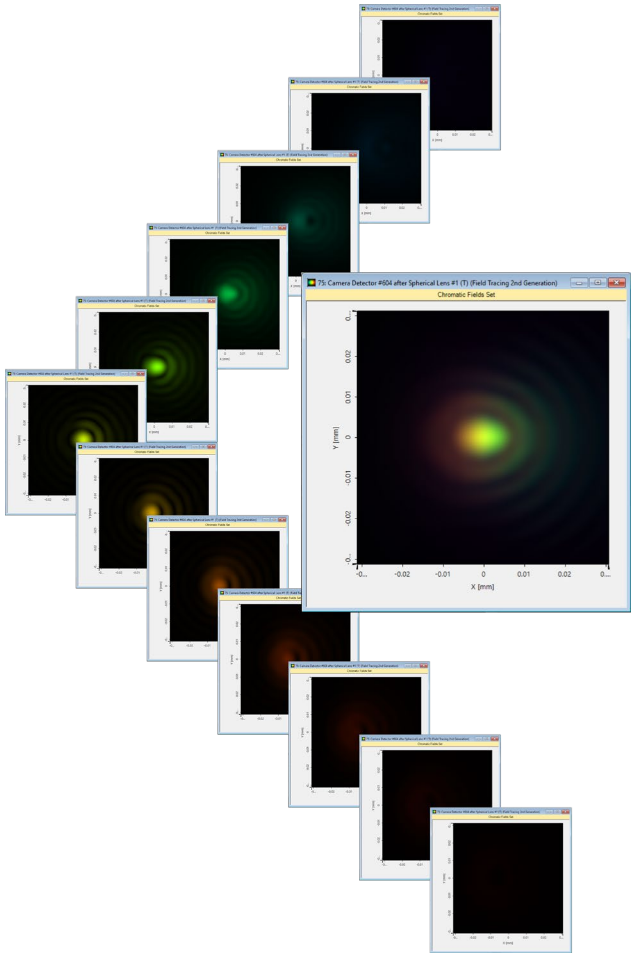


A Note on the Light Representation

- The vector electromagnetic field that represents light in physical optics is always fully accessible in VirtualLab Fusion as it is traced through the system.
- For this approach to be practical from the point of view of computational efficiency, it is paramount to have at our disposal a diverse set of mathematical techniques (efficient Fourier transform algorithms, interpolation and fitting methods, heterogeneous sampling mechanisms, among others).
- In the current version of VirtualLab Fusion, this translates into the coexistence of several simulation engines:
  - Ray tracing: pure ray tracing, yielding both 2 and 3D results
  - Classic Field Tracing: handles equidistantly sampled EM field data
  - 2nd Generation Field Tracing: is also able to handle non-equidistant EM field data
- This is relevant to the Programmable Component: a good implementation of your algorithm needs to take into account how light is represented in the different engines!

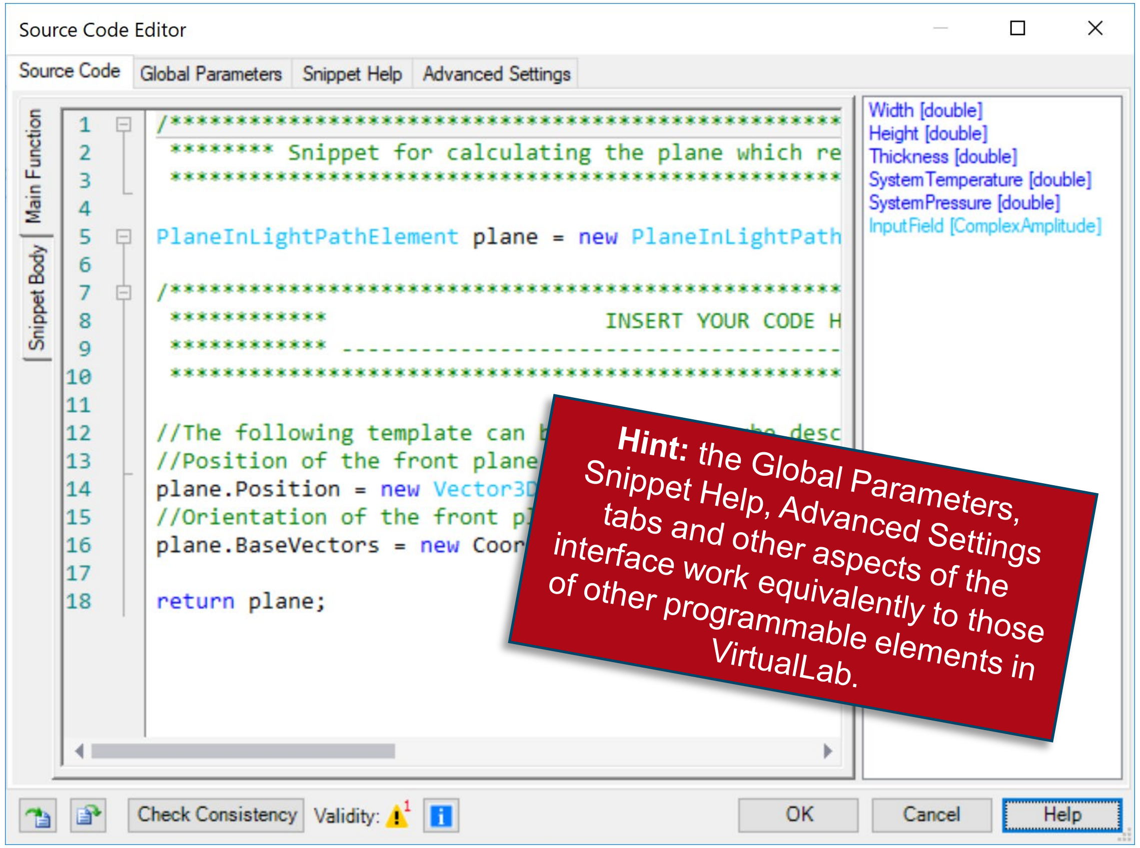


A Note on the Light Representation



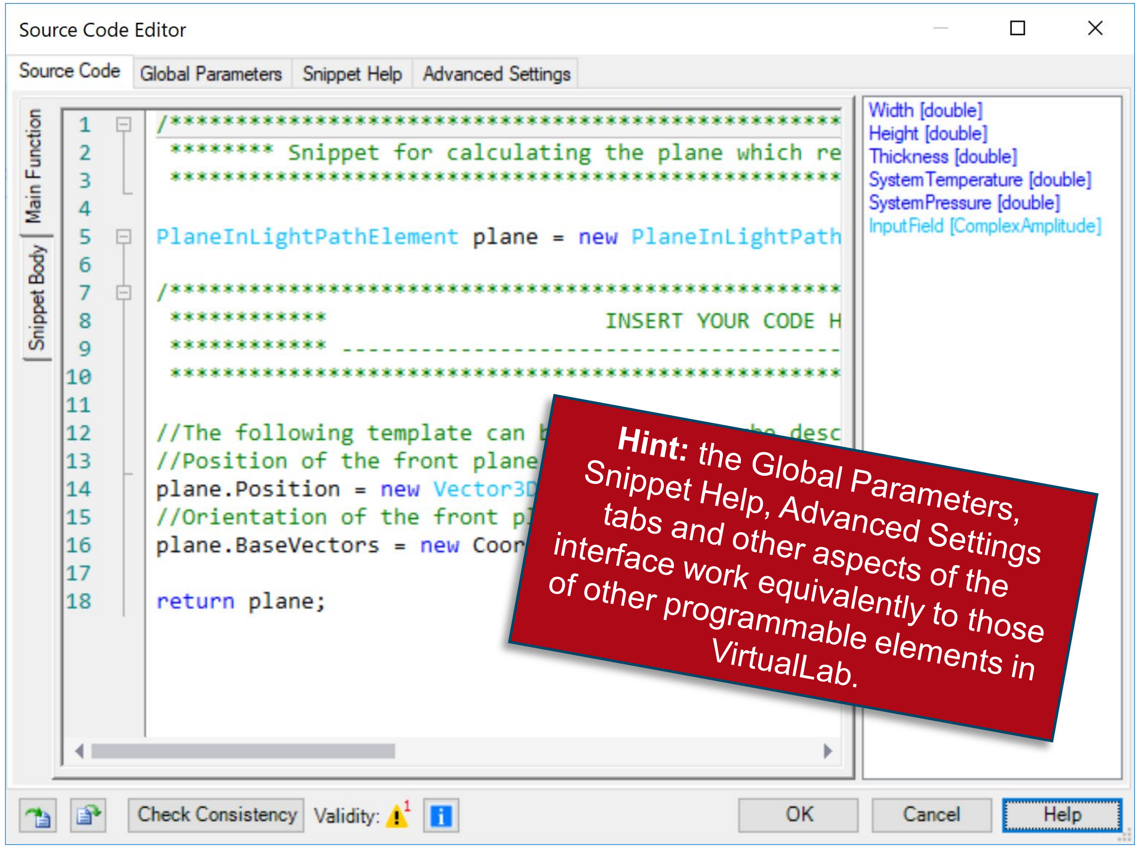
- Additionally, in order to replicate a series of important physical properties of light (partial coherence, for instance, whether temporal or spatial) VirtualLab uses a mode decomposition.
- The different modes are accessible in the Programmable Component via a series of indices.
- Taking the different modes into account is also fundamental if a Programmable Component is to exhibit the correct desired physical behaviour!

Writing the Code: Input Transface



- There are three customizable snippets in the Programmable Component, the first of which is the Input Transface: the plane where the field is retrieved from the previous free-space propagation and imported into the component.
- In other real components, which are constructed from surfaces and media, the geometry is accessible to the VirtualLab code and, therefore, the software can automatically determine a suitable Input Transface for the component. Not so in the Programmable Component, where the full functionality is defined by the user.
- It falls then upon the user too to provide a suitable Input Transface plane for their component.

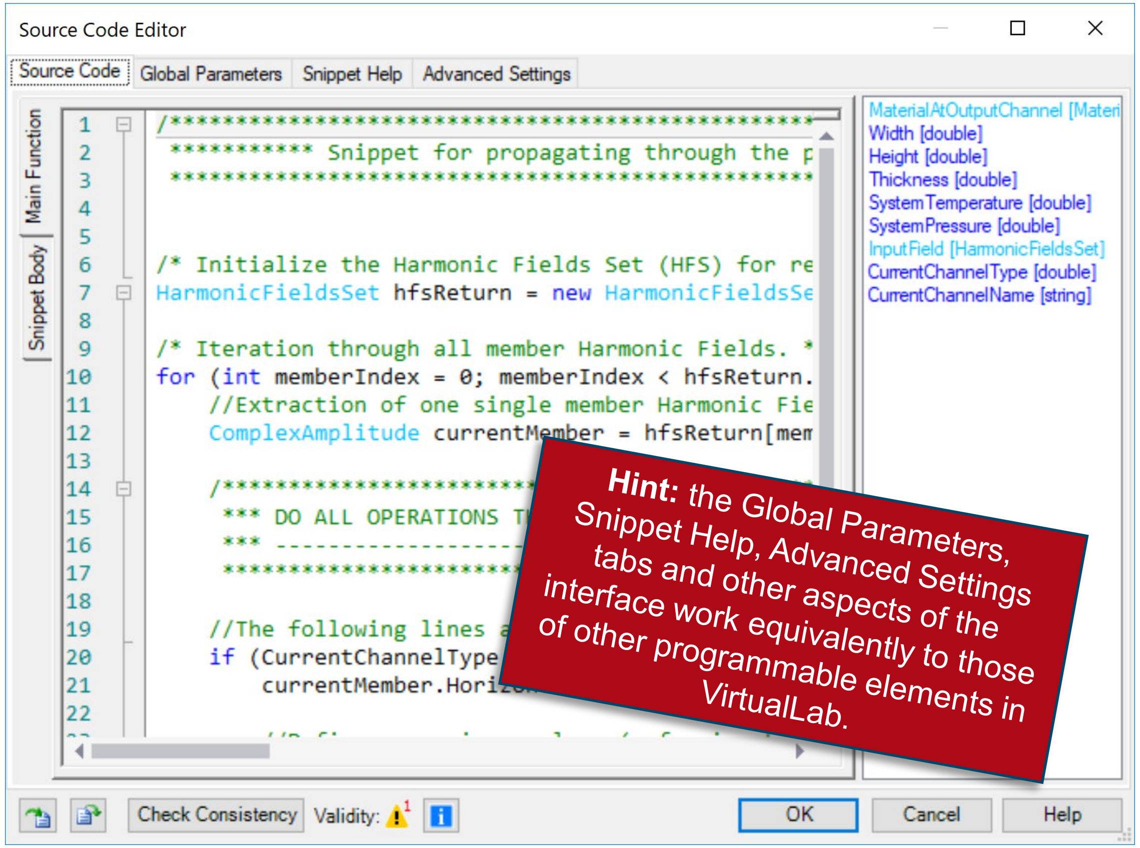
Writing the Code: Input Transface



- Some global parameters are available by default in the snippet for the Input Transface.
- `Width`, `Height` and `Thickness` give the dimensions of the bounding box (the three dimensional cuboid which encompasses at least the entire volume of the component). The values of these parameters are input by the user in the configuration dialog for the component.
- `SystemTemperature` and `SystemPressure` are parameters of the Optical Setup in which the component is included.
- `InputField` refers to the individual field modes which reach the Programmable Component.

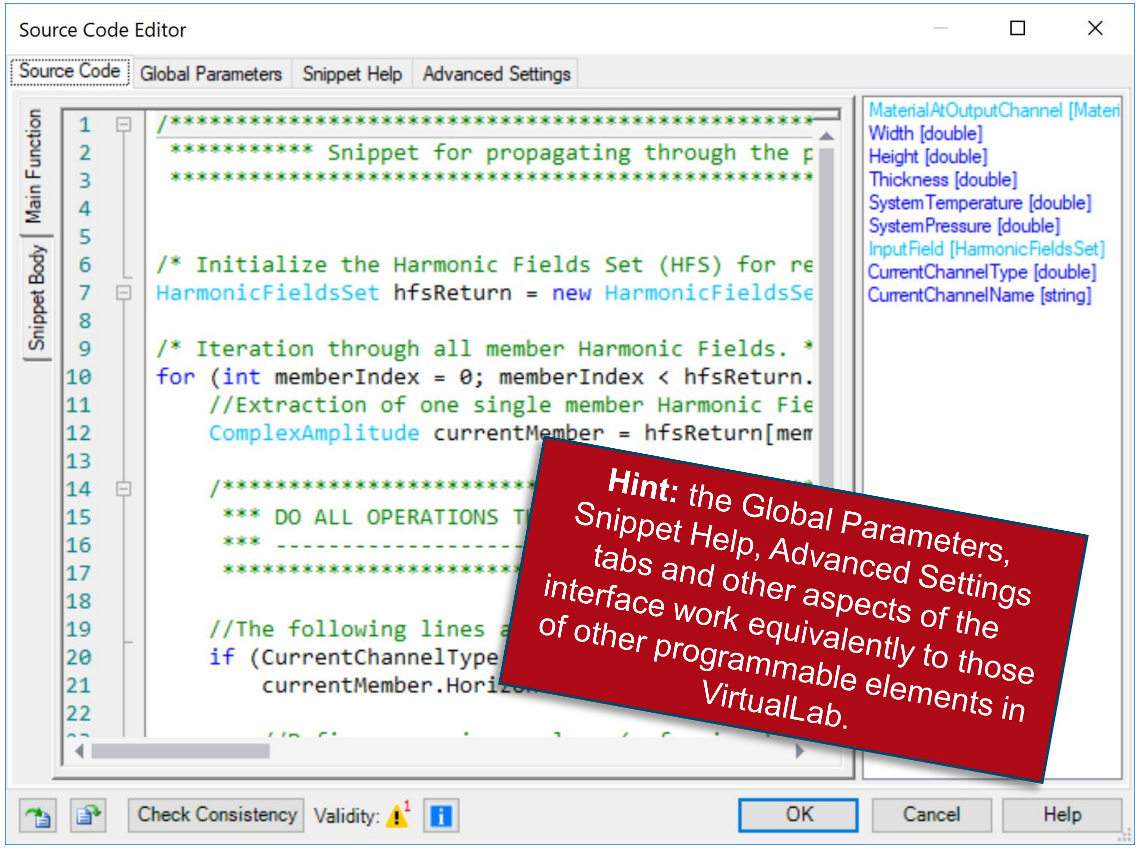
Writing the Code: Equidistant Field Data

- The Programmable Component provides two different programming dialogs for light propagation through the component in question. These are related to the simulation engines. The first, titled Snippet for Equidistant Field Data, handles electromagnetic field objects sampled on an equidistant, rectangular *x, y* grid.
- It is a direct result of Maxwell's equations that in homogeneous media only two of the six electromagnetic components are independent; consequently, the fields reaching the component consist only of *Ex* and *Ey* components, all the other four being thus unequivocally determined and possible to calculate on demand if so required.
- Depending on the polarization characteristics of the incoming field, *Ex* and *Ey* can be two independent functions (local polarization) or obtained from a single field function *U* via a constant Jones' vector (constant in *x* and *y*), so that *Ex* = *Jx* * *U* and *Ey* = *Jy* * *U*.



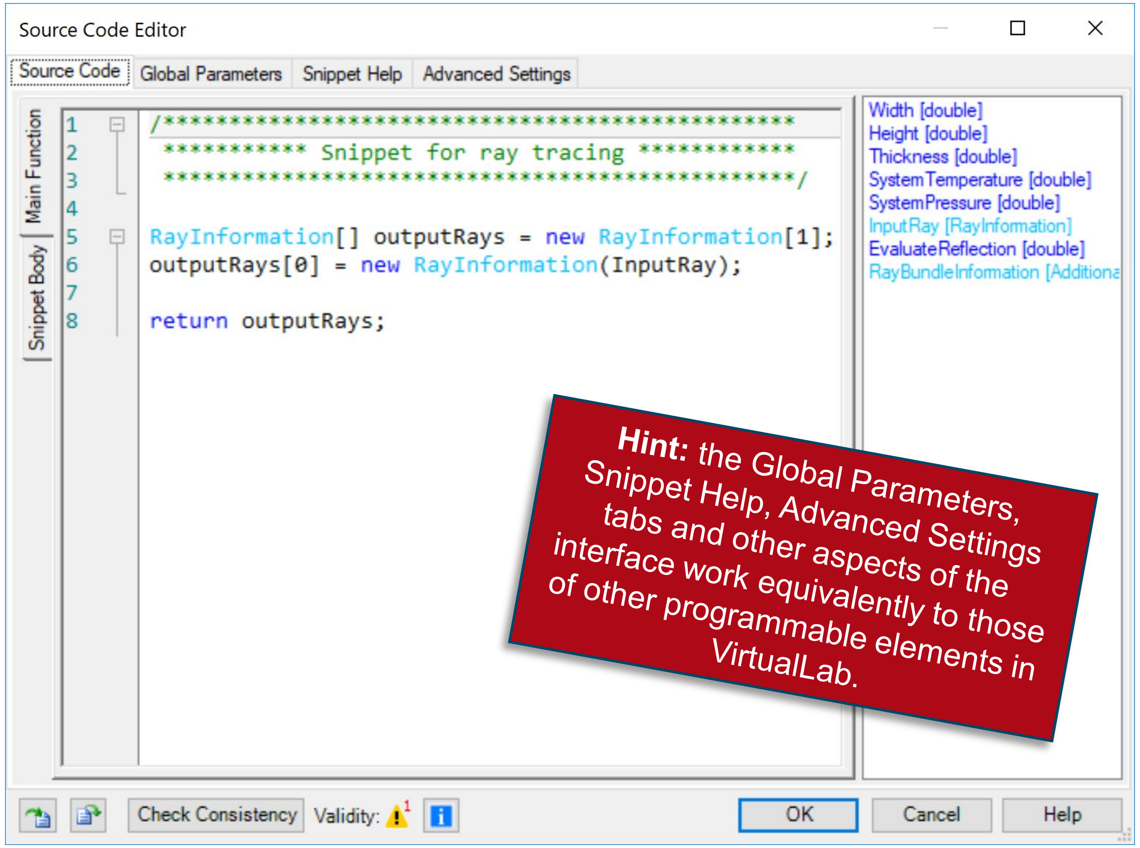
Writing the Code: Equidistant Field Data

- The panel on the right shows a list of available independent parameters.
- `MaterialAtOutputChannel` contains information about the material which has been defined in the system at the output channel of the component. Depending on the nature of the channel (reflection or transmission) this material can coincide with the input one, or be a different one. The properties of this object allow the user to access, among others, the corresponding refractive index.
- `Width`, `Height` and `Thickness` give the dimensions of the bounding box, which coincide with those for the Input Transface.
- `SystemTemperature` and `SystemPressure` are parameters of the whole system, whose value can be used in the code to implement temperature- and pressure-dependent responses.
- `InputField` contains the full set of modes which represent the light entering the component at the Input Transface. In this snippet they are equidistantly sampled on an *x*, *y* grid.
- `CurrentChannelType` encodes information related to the nature of the channel: its value is 1 for channels working in transmission, and 0 for reflection.
- `CurrentChannelName` is a string with, as the name of the variable itself suggests, the name of the channel.



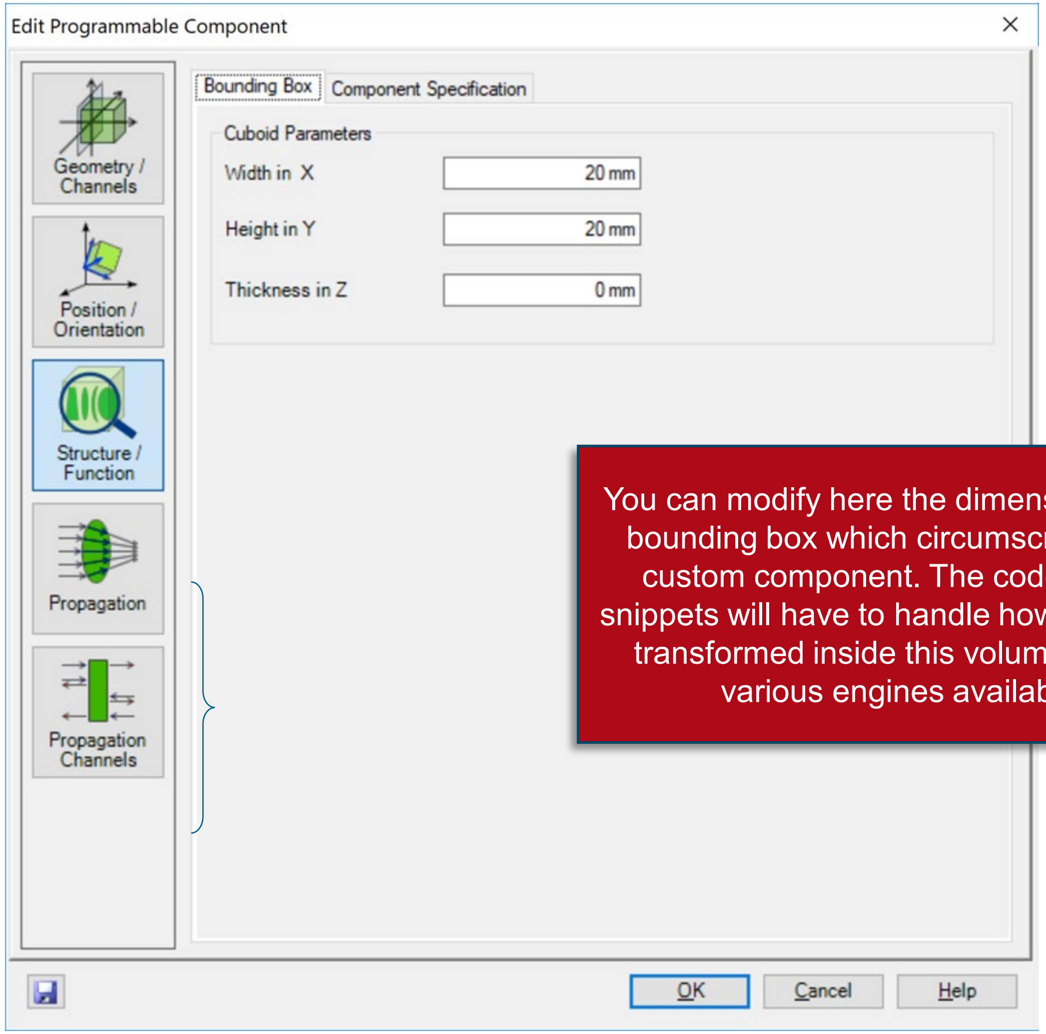
Hint: the Global Parameters, Snippet Help, Advanced Settings tabs and other aspects of the interface work equivalently to those of other programmable elements in VirtualLab.

Writing the Code: Non-Equidistant Field and Ray Data

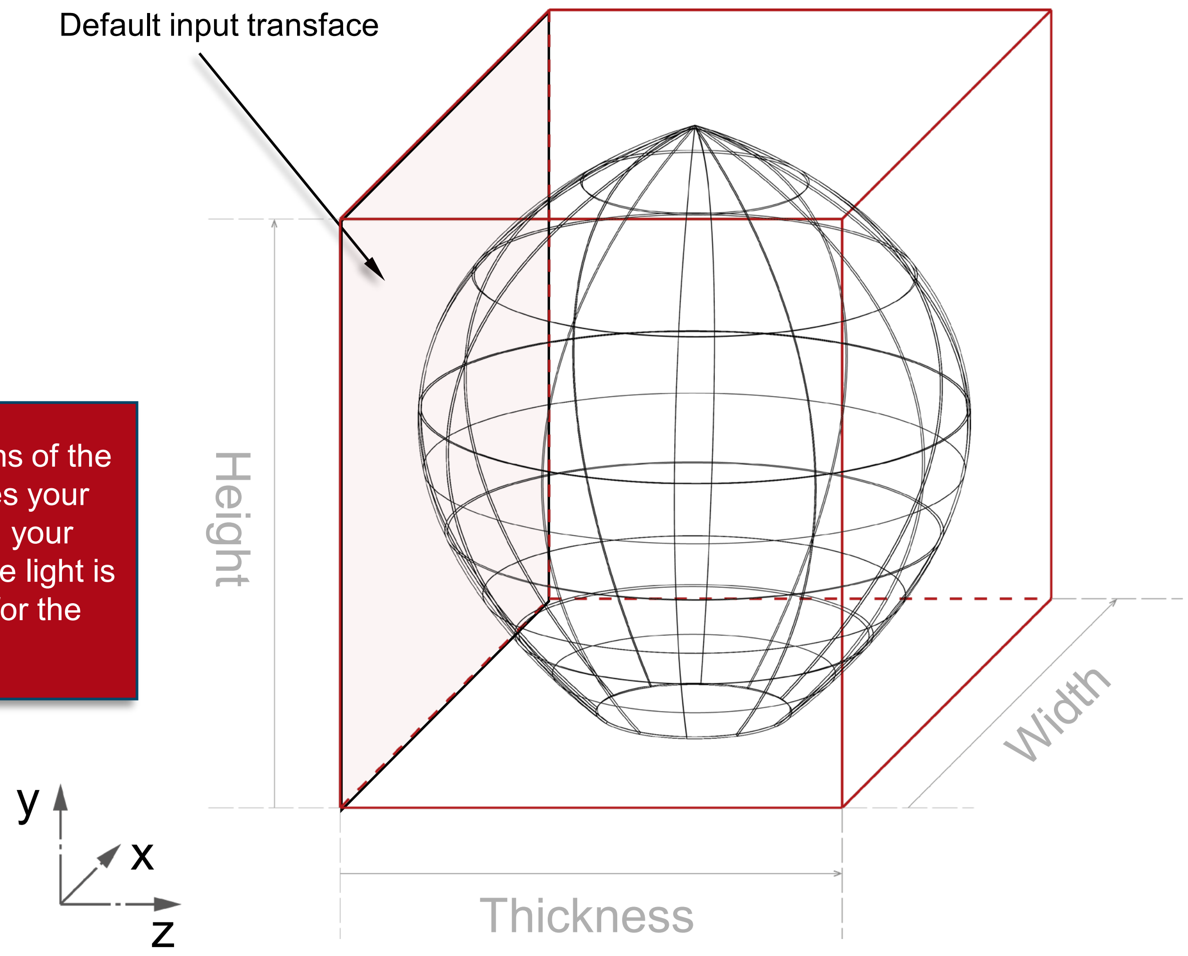


- The last programming dialog in the Programmable Component handles non-equidistantly sampled field data and rays.
- For non-equidistant fields, the vector field samples may coincide with the ray samples. This snippet can therefore return both ray information—if the simulation is run with the Ray Tracing Engine—and physical optics results—when the chosen engine is 2nd Generation Field Tracing. It is the programmer's responsibility to account for both instances.
- The panel on the right shows, again, a list of available independent parameters. The first items on the list coincide with those in the other snippet.
- `InputRay` refers to each of the individual rays or field samples (depending on the engine) that reach the component.
- `EvaluateReflection` works in a similar way as `CurrentChannelType` in the snippet for equidistantly sampled fields.
- `RayBundleInformation` contains information about the ray or field-sample bundle which contains the currently handled instance.
- The code in this snippet is then implemented per ray or field sample. The same code is then iterated by VirtualLab when the simulation is run for each of the rays/field samples present.
- Do not let the names `InputRay` and `RayBundleInformation` fool you! This nomenclature is obsolete and will be phased out in future versions.

Bounding Box Configuration

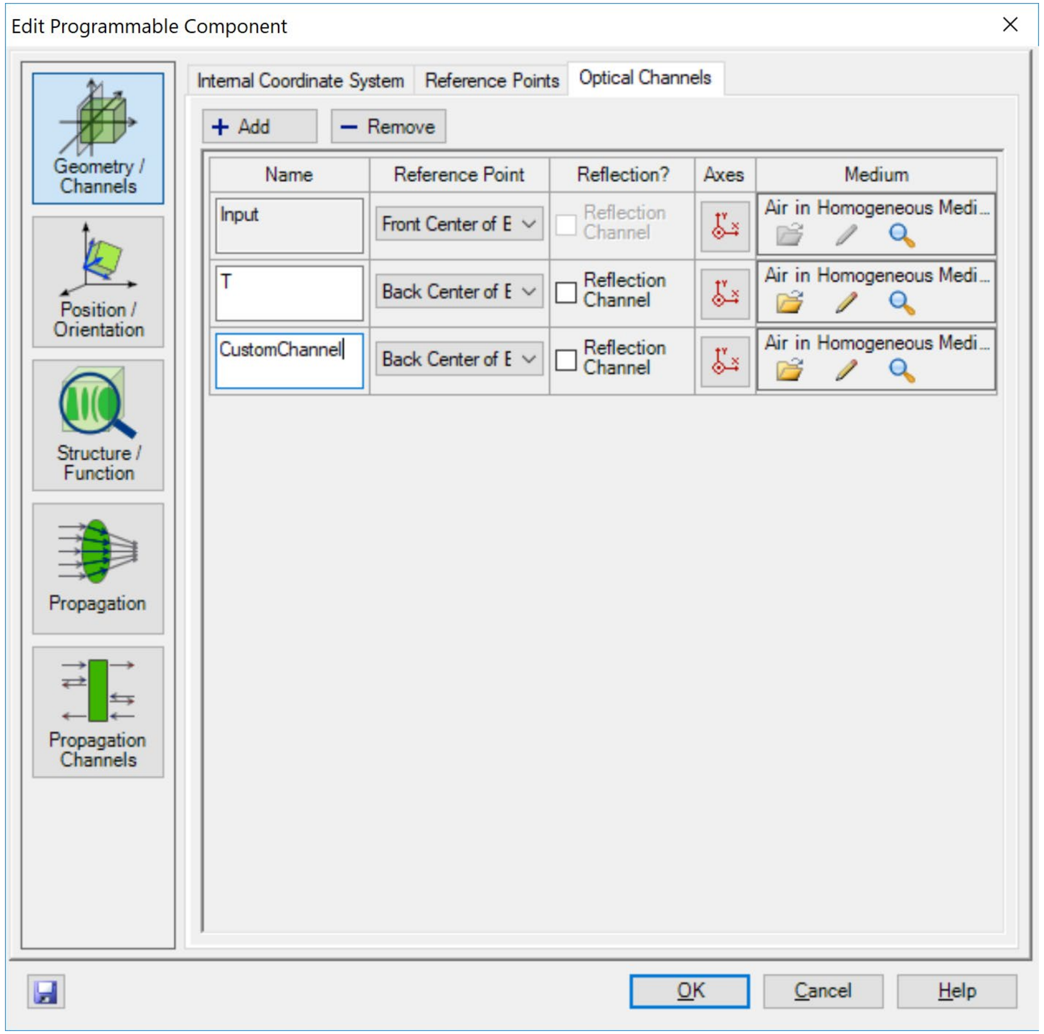


You can modify here the dimensions of the bounding box which circumscribes your custom component. The code in your snippets will have to handle how the light is transformed inside this volume, for the various engines available.



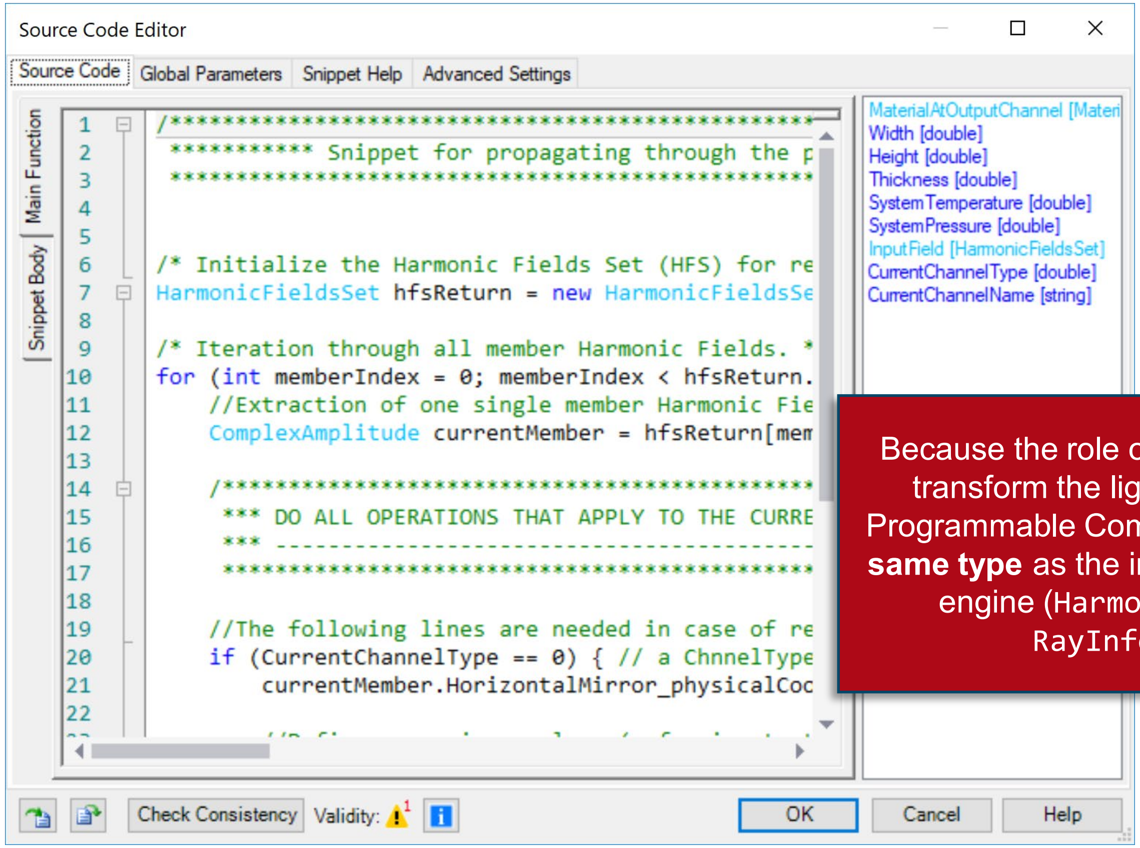
Channel Definition

- You can add and delete different output channels in your programmable component.
- Information regarding their name (also user-defined) and whether they are a reflection channel or not can then be accessed in the snippet code.
- The most obvious configuration is one where there are two possible channels (transmission and reflection) but more complex configurations are possible (for instance, a grating with different propagating orders).
- When a channel is marked as reflection the output medium is adjusted accordingly. However, it is the programmer’s responsibility to make sure that the coordinate systems are defined properly in the code!

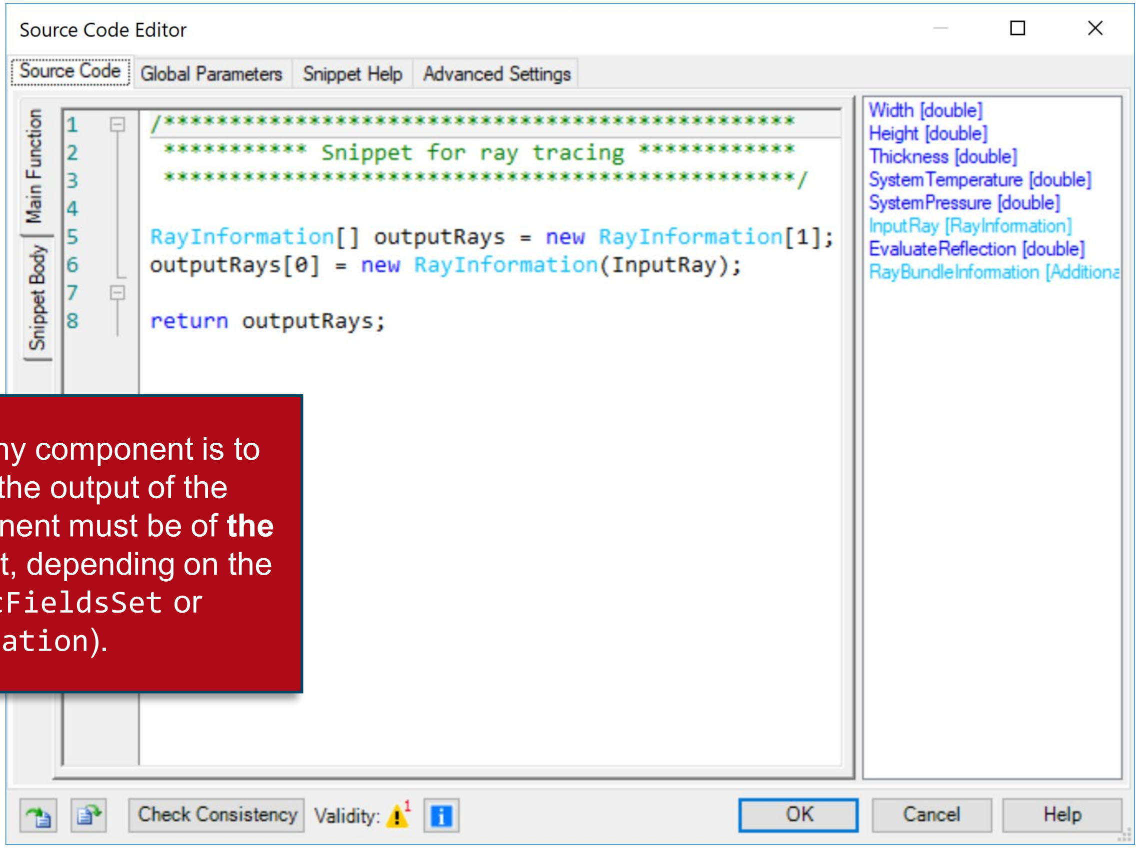


Output

Equidistantly sampled fields:



Non-equidistantly sampled fields and rays:



Because the role of any component is to transform the light, the output of the Programmable Component must be of **the same type** as the input, depending on the engine (`HarmonicFieldsSet` or `RayInformation`).

Programming an Ideal Grating

Ideal Grating

The objective of this example is to create a custom component that imitates the behaviour of an ideal grating: for given incident direction and input and output media, it should compute the outgoing direction of a certain diffraction order (in transmission), with the desired order and the corresponding scalar diffraction efficiency preliminary user-defined parameters. The main formula which shall be employed in this example is the grating equation:

$$\boldsymbol{\kappa}_m^{\text{out}} = \boldsymbol{\kappa}^{\text{in}} + \Delta\boldsymbol{\kappa}_m \qquad (1)$$

$$\boldsymbol{\kappa} = (k_x, k_y)$$

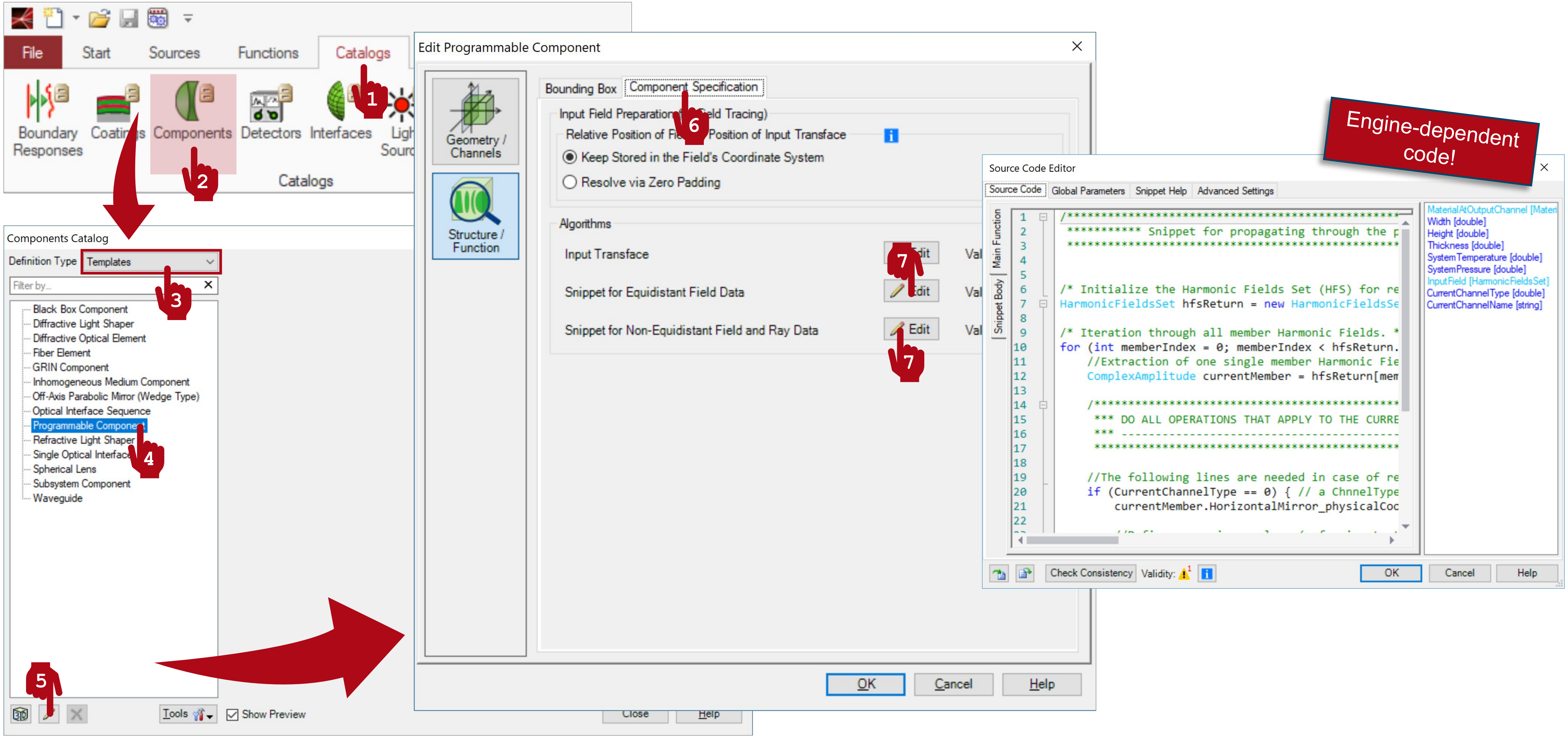
$$\Delta\boldsymbol{\kappa}_m = n^{\text{out}} \frac{2\pi}{d} m$$

$m \rightarrow$ diffraction order

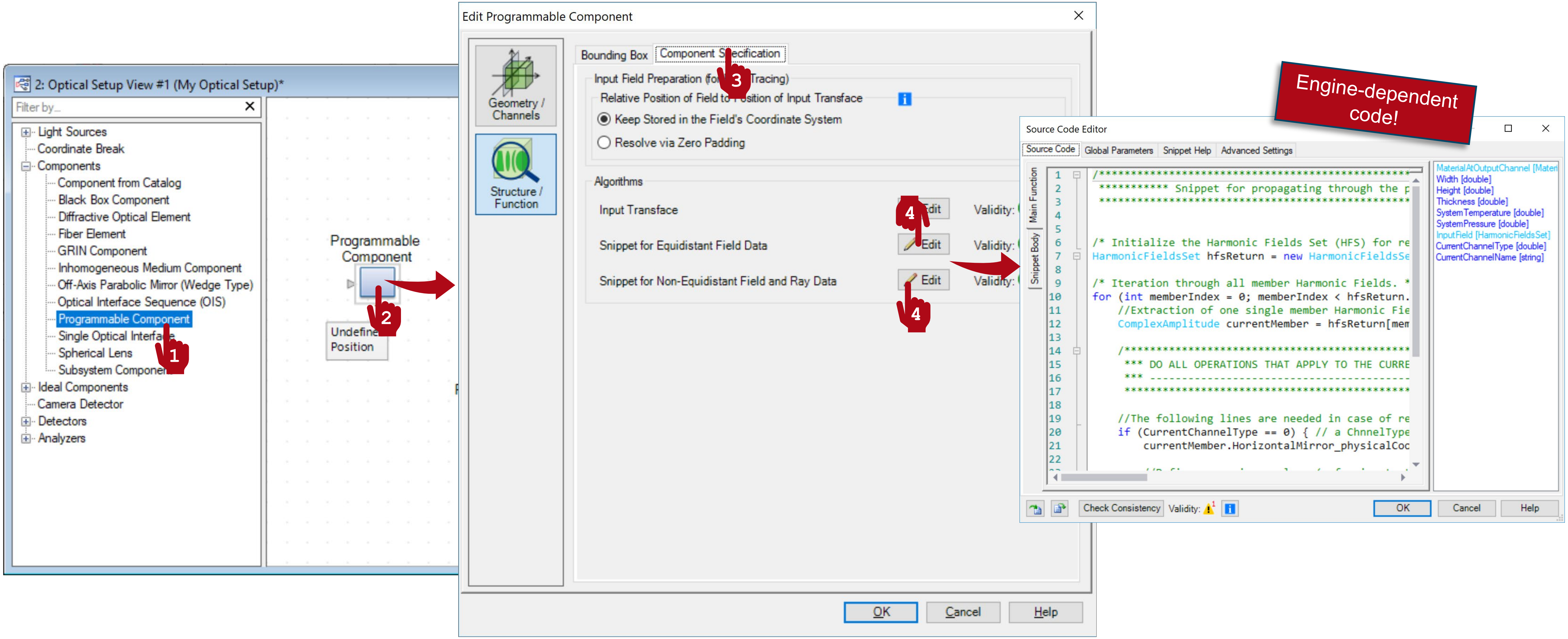
$d \rightarrow$ grating period

For a single embedding medium, Eq. (1) reduces to the well-known $d\left[\sin\left(\theta_m^{\text{out}}\right) - \sin\left(\theta^{\text{in}}\right)\right] = m\lambda$

Where to Find the Programmable Component: Catalog



Where to Find the Programmable Component: Optical Setup

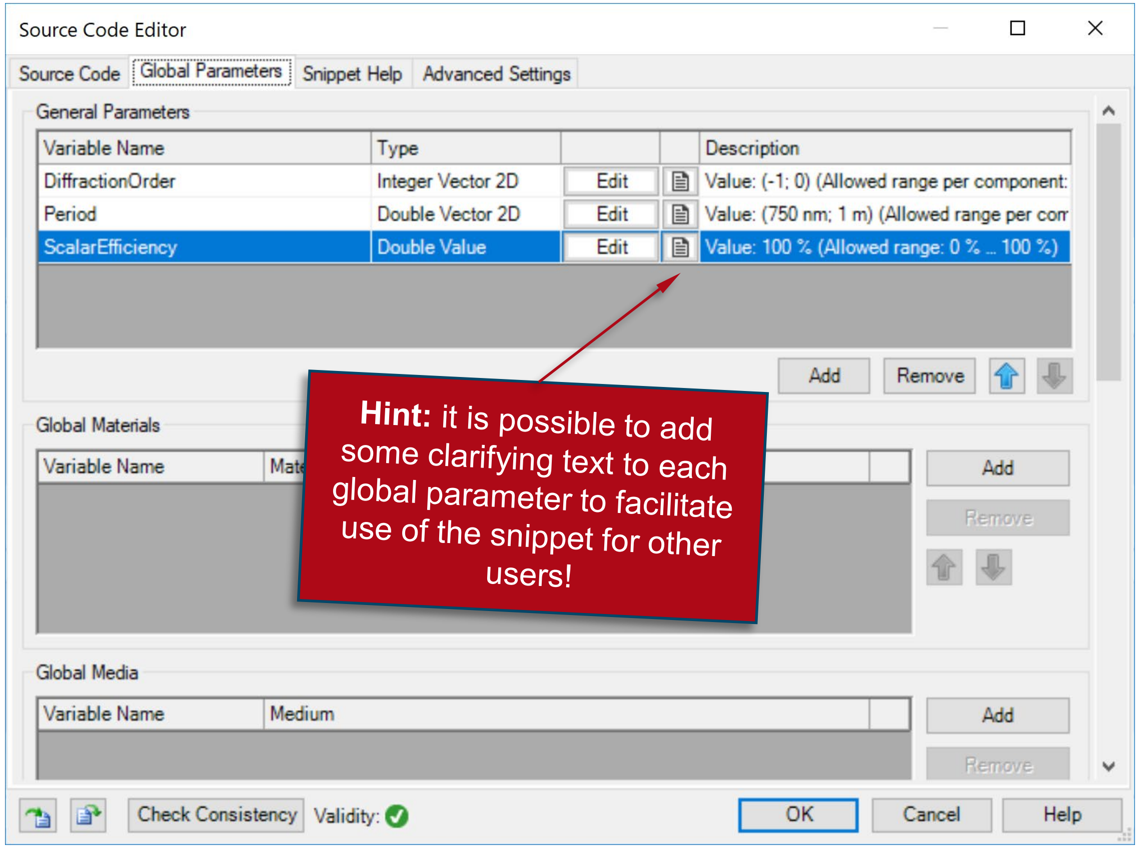


Programmable Component: Global Parameters

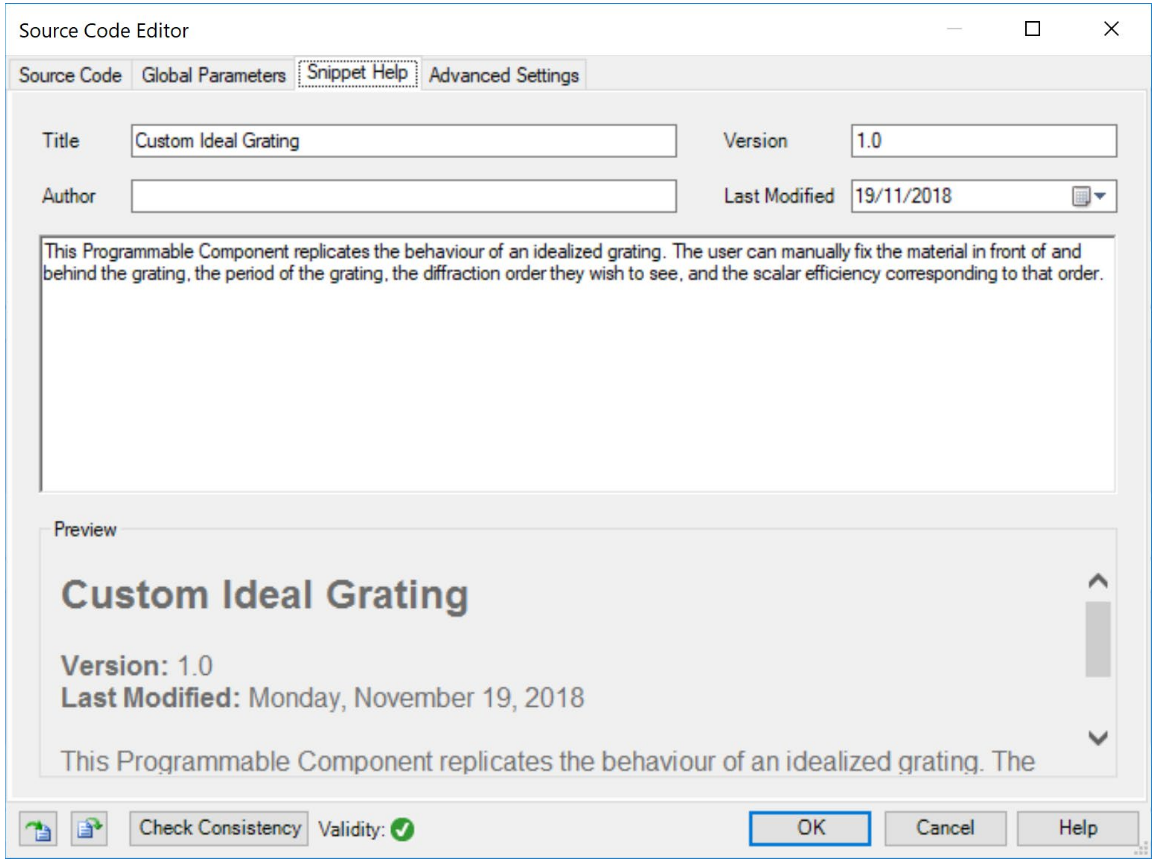
- Once you have triggered open the Edit dialog (Source Code Editor), go to the Global Parameters tab.

- There, Add and Edit three parameters:
  - `Vector DiffractionOrder = (-1, 0), (per component -1000, 1000)`: the index, in *x* and *y*, of the desired diffraction order.
  - `VectorD Period = (750 nm, 1 m), (per component 0 m, 1 m)`: the period of the grating, in *x* and *y*.
  - `double ScalarEfficiency = 100 % (0 %, 100 %)`: the efficiency of the diffraction order.

- Use the button with the small “notes” icon to add some explanation to your custom global parameters.

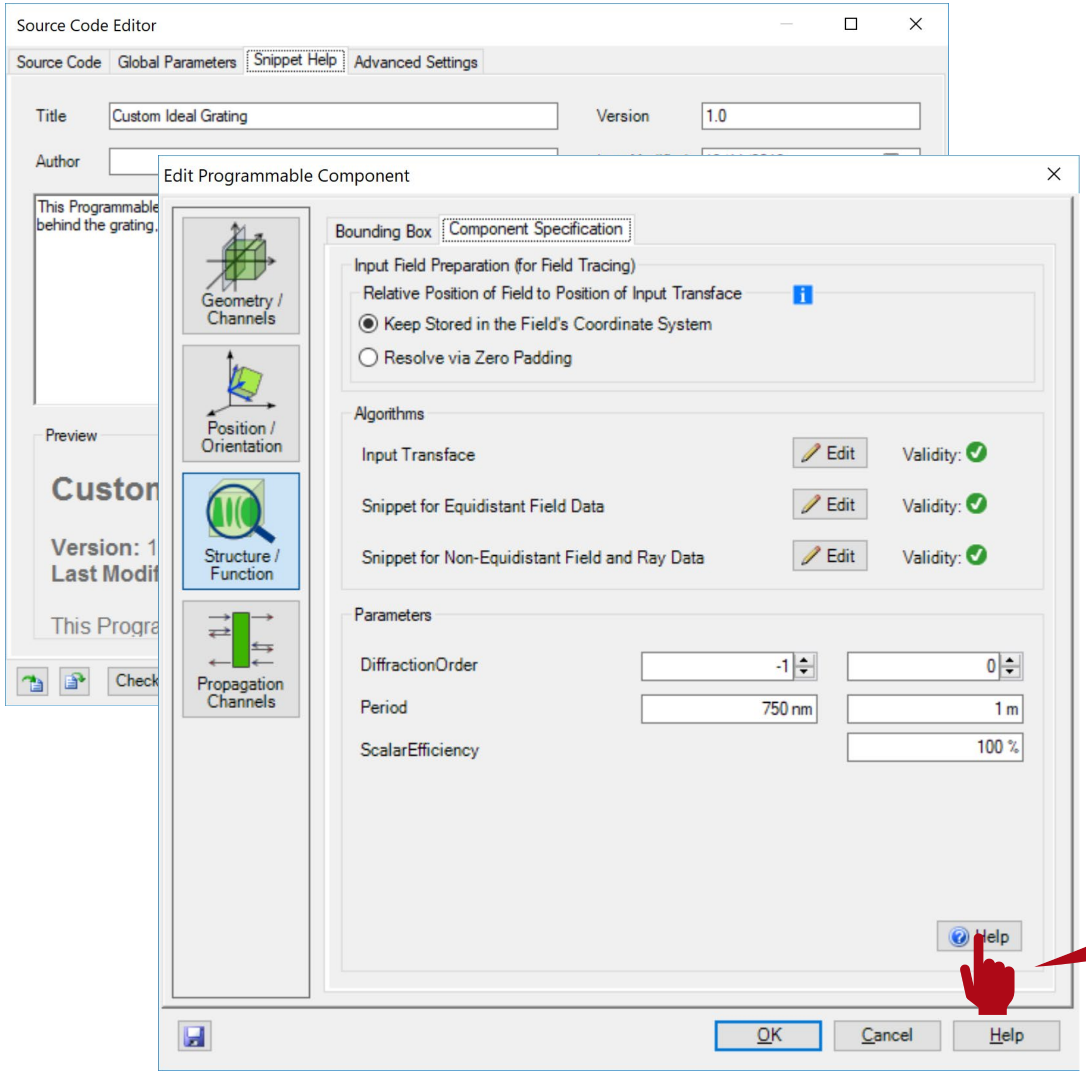


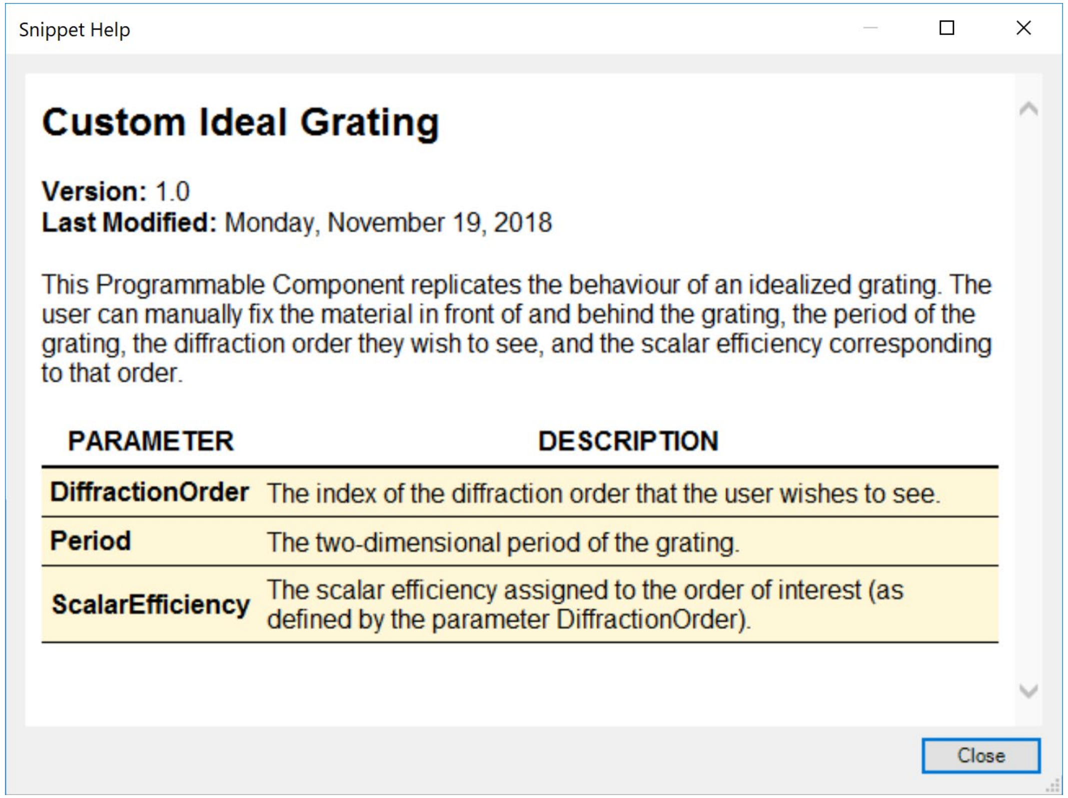
Programmable Component: Snippet Help



- **Optional:** you can use the Snippet Help tab to write instructions, clarifications, and some additional data associated to your snippet.
- This option is very helpful to keep track of your progress with a programmable element.
- It is especially useful when the programmable element is later disseminated to be handled by other users!

Programmable Component: Snippet Help





Programmable Component: Writing the Code (1)

Snippet for equidistantly sampled field results!

Source Code Editor

Source Code | Global Parameters | Snippet Help | Advanced Settings

Main Function | Snippet Body

```
HarmonicFieldsSet hfsReturn = new HarmonicFieldsSet(InputField);

for (int memberIndex = 0; memberIndex < hfsReturn.Count; memberIndex++)
{
    ComplexAmplitude currentMember = hfsReturn[memberIndex];

    double wavelengthOfMember = currentMember.Wavelength;

    Vector3D inputDirectionOfCurrentMember = currentMember.CentralDirection;

    Complex refractiveIndexInFrontOfGrating = currentMember.EmbeddingMedium.BaseMaterial.GetComplexRefractiveIndex(
        wavelengthOfMember,
        SystemTemperature,
        SystemPressure);
    Complex refractiveIndexBehindGrating = MaterialAtOutputChannel.GetComplexRefractiveIndex(
        wavelengthOfMember,
        SystemTemperature,
        SystemPressure);

    Vector3D diffractedDirection = VL_Propagations.CalculateDiffractedDirectionTransmission(
        inputDirectionOfCurrentMember,
        refractiveIndexInFrontOfGrating,
        refractiveIndexBehindGrating,
        DiffractionOrder,
        Period,
        wavelengthOfMember);

    currentMember.CentralDirection = diffractedDirection;

    currentMember *= Math.Sqrt(ScalarEfficiency);

    hfsReturn[memberIndex] = currentMember;
}

return hfsReturn;
```

Material... [Mater...
Width [do...
Height [double]
Thickness [double]
SystemTemperature [double]
SystemPressure [double]
InputField [HarmonicFieldsSet]
CurrentChannelType [double]
CurrentChannelName [string]
DiffractionOrder [Vector]
Period [VectorD]
ScalarEfficiency [double]

Check Consistency | Validity: ✓ | OK | Cancel | Help

- Declare and assign output
- Extract a single mode
- Read in wavelength and incoming direction of mode
- Read in refractive indices (input and output)
- Modify outgoing direction of member
- Export Snippet to save your work!
- Loop through all the incoming modes
- Compute outgoing direction of grating order according to Eq. (1)
- Apply efficiency
- Are there errors in your code?
- Default global parameters/variables
- Global parameters defined by user in Global Parameters tab

Programmable Component: Writing the Code (2)

Snippet for ray & non-equidistantly sampled field results!

Source Code Editor

Source Code | Global Parameters | Snippet Help | Advanced Settings

Main Function | Snippet Body

Declare and assign output

Read in wavelength, position and incoming direction of individual sample

```
List<RayInformation> outputRays = new List<RayInformation>();
double currentWavelength = InputRay.Wavelength;
Vector3D positionOfInputSample = InputRay.Position;
Vector3D incomingDirection = InputRay.Direction;

Complex refractiveIndexInFrontOfGrating =
    RayBundleInformation.MediumOfBundle.BaseMaterial.GetComplexRefractiveIndex(
    InputRay.Wavelength,
    SystemTemperature,
    SystemPressure);

Complex refractiveIndexBehindGrating = refractiveIndexInFrontOfGrating;

Vector3D diffractedDirection = VL_Propagations.CalculateDiffractedDirectionTransmission(
    incomingDirection,
    refractiveIndexInFrontOfGrating,
    refractiveIndexBehindGrating,
    DiffractionOrder,
    Period,
    currentWavelength);

outputRays.Add(new RayInformation(InputRay));
outputRays[outputRays.Count - 1].Direction = diffractedDirection;

VectorD DeltaKappa = new VectorD(0, 0);
DeltaKappa.X = 2.0 * Math.PI * DiffractionOrder.X / Period.X;
DeltaKappa.Y = 2.0 * Math.PI * DiffractionOrder.Y / Period.Y;
outputRays[outputRays.Count - 1].AbsolutePhase +=
    new VectorD(positionOfInputSample.X, positionOfInputSample.Y) | DeltaKappa;

double scalarSurfaceResponse = Math.Sqrt(ScalarEfficiency *
    (refractiveIndexInFrontOfGrating.Re * InputRay.Direction.Z) /
    (refractiveIndexBehindGrating.Re * diffractedDirection.Z));
outputRays[outputRays.Count - 1].EnergyConservationFactor *= scalarSurfaceResponse;

return outputRays.ToArray();
```

Loop through all the incoming modes

Read in refractive indices (output = input)

Compute outgoing direction of grating order according to Eq. (1)

Generate output sample from input and modify its direction

Compute and apply shift

Energy conservation

Width [doubl
Height [double
Thickness [double]
SystemTemperature [double]
SystemPressure [double]
InputRay [RayInformation]
EvaluateReflection [double]
RayBundleInformation [Additiona
DiffractionOrder [Vector]
Period [VectorD]
ScalarEfficiency [double]

Default global parameters/variables

Global parameters defined by user in Global Parameters tab

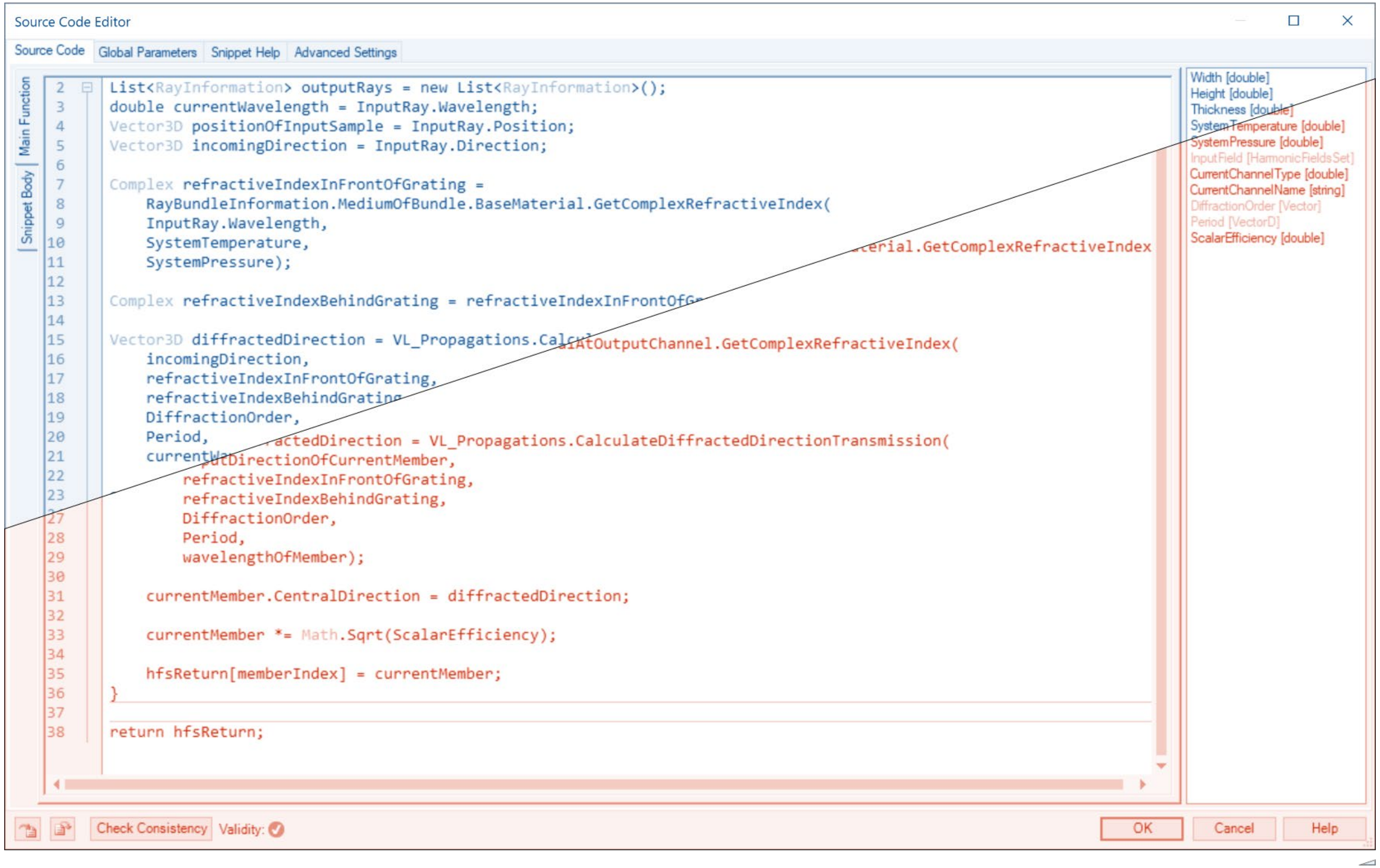
Are there errors in your code?

Export Snippet to save your work!

Check Consistency | Validity:

OK | Cancel | Help

Programmable Component: Comparing the Snippets



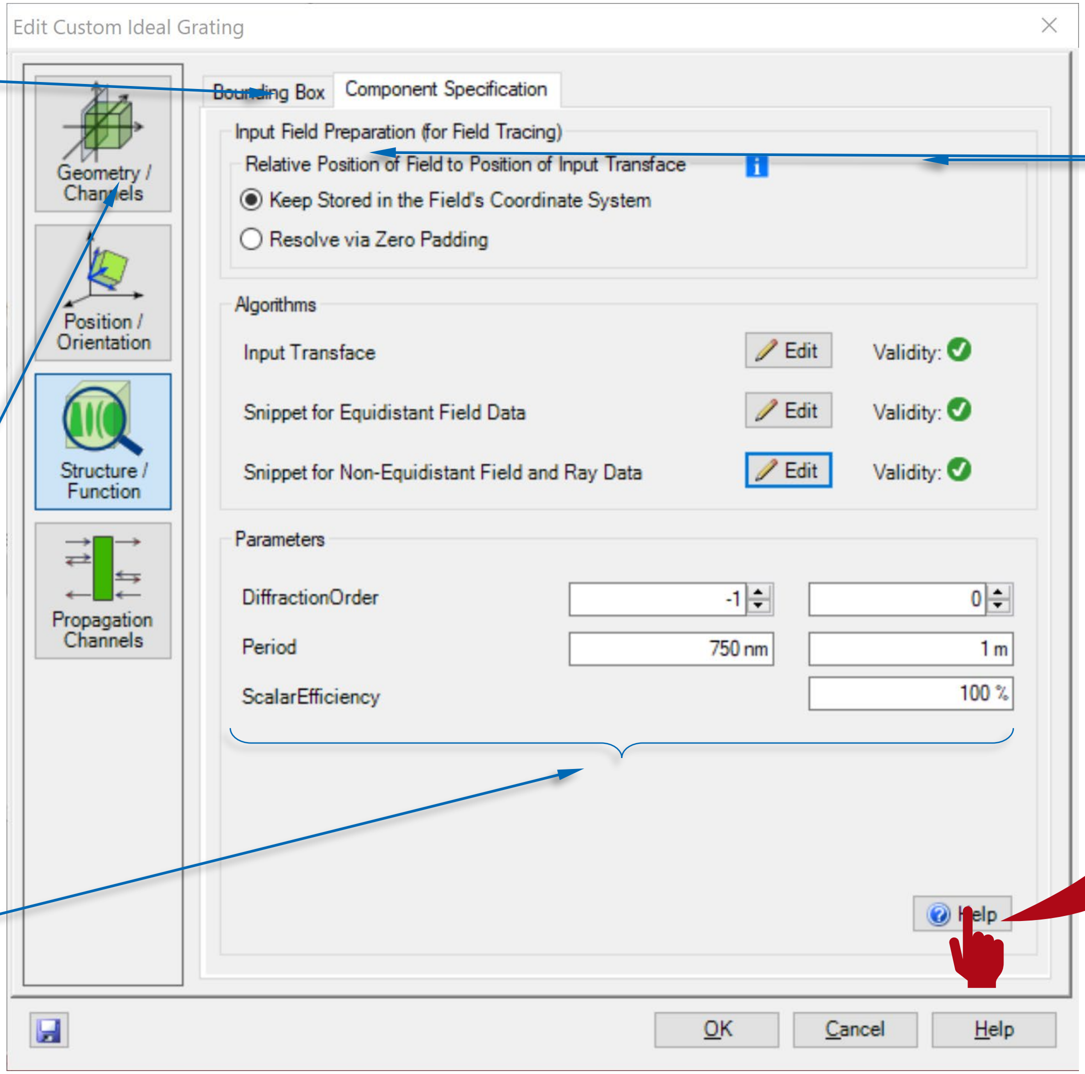
- Variables need to be declared separately and independently in both snippets.
- It would even be possible to use different nomenclature!
- It is the programmer’s responsibility to ensure that the code functions in an equivalent manner in both snippets.
- Of all the global parameters (including those defined by the user) only one is snippet-dependent: the one corresponding to light representation (InputField ←→ RayTracing Result)

Programmable Component: Using Your Snippet

In the Bounding Box tab you can modify the dimensions of said Bounding Box

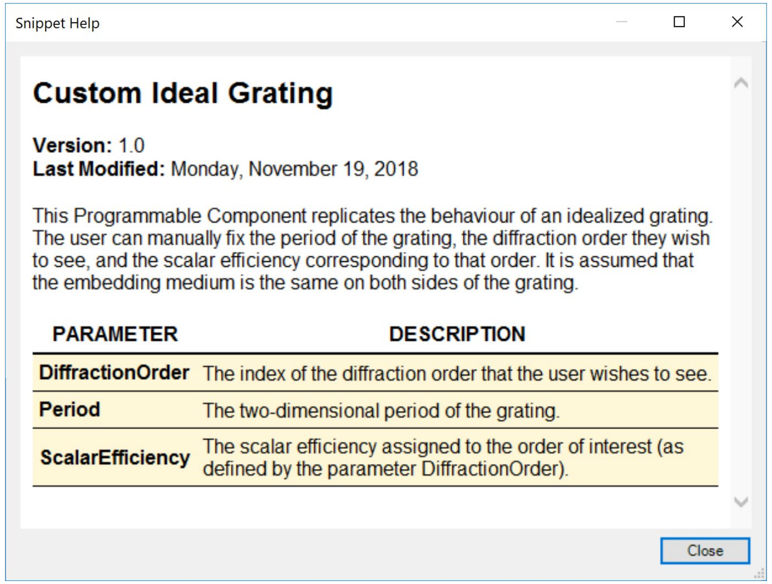
In the Geometry/Channels menu you can control some aspects of the coordinate system, and include additional output channels for your implementation

You can modify the value of the global parameters you defined here

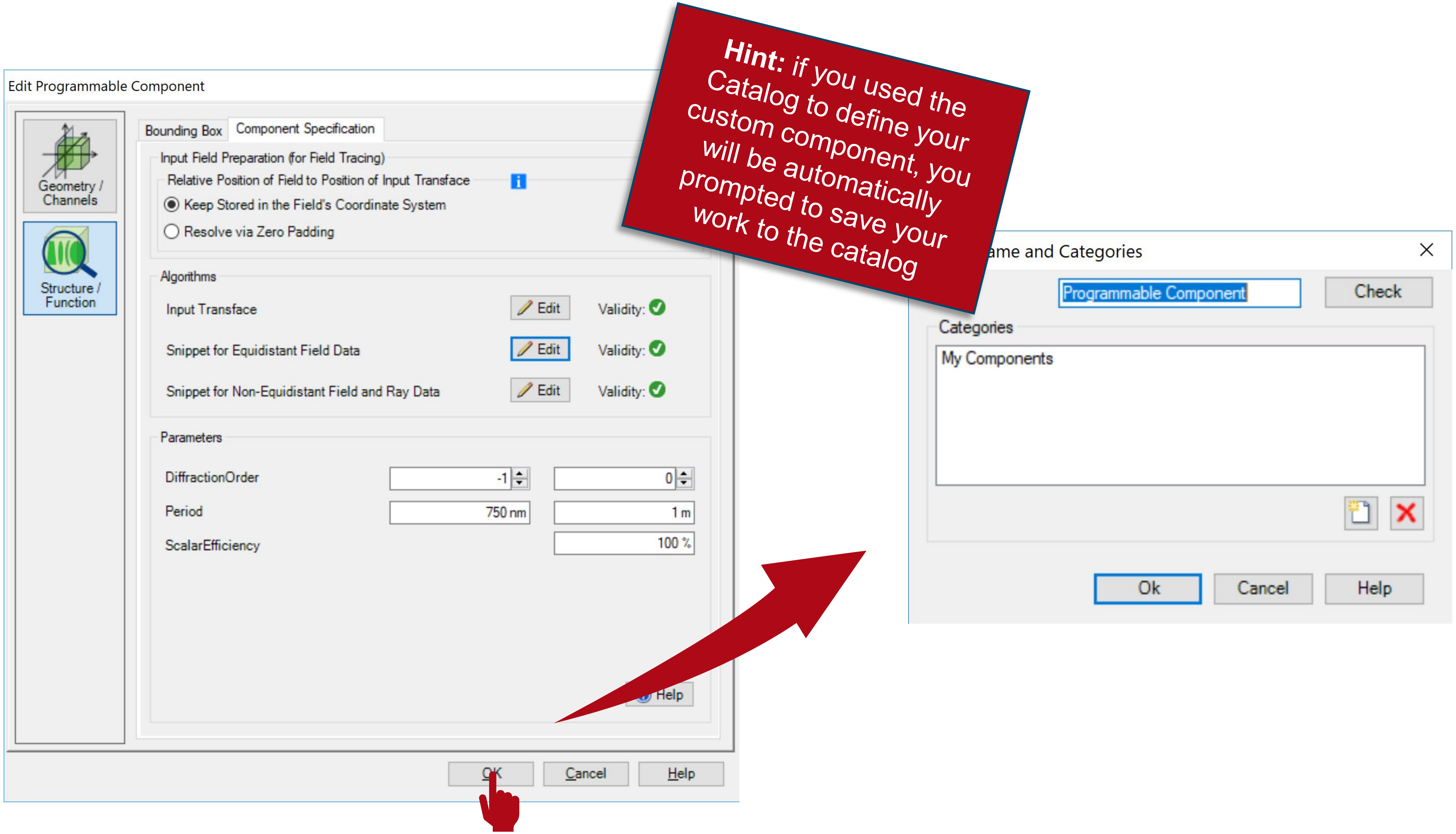


For the equidistant field snippet, this controls with what mechanism any eventual overall position shift of the field will be handled: via padding (increases sampling effort) or via coordinate system.

Modify your snippets by clicking on Edit



Save the Custom Component to the Catalog



Test the Code!

Main Function (Equidistant)

```
// Generate output (by copying input):
HarmonicFieldsSet hfsReturn = new HarmonicFieldsSet(InputField);

// Run loop through all the members of the input:
for (int memberIndex = 0; memberIndex < hfsReturn.Count; memberIndex++)
{
    // Extract the individual member (Complex Amplitude):
    ComplexAmplitude currentMember = hfsReturn[memberIndex];

    // Read in the wavelength of the member:
    double wavelengthOfMember = currentMember.Wavelength;

    // Read in the incoming direction of the member:
    Vector3D inputDirectionOfCurrentMember = currentMember.CentralDirection;

    // Read in the refractive index of the medium in front of the grating:
    Complex refractiveIndexInFrontOfGrating = currentMember.EmbeddingMedium.BaseMaterial.GetComplexRefractiveIndex(
        wavelengthOfMember,
        SystemTemperature,
        SystemPressure);

// Continued in next slide.
```

Test the Code!

Main Function (Equidistant)

```
// Continued from last slide.

    // Read in the refractive index in the medium behind the grating:
    Complex refractiveIndexBehindGrating = MaterialAtOutputChannel.GetComplexRefractiveIndex(
        wavelengthOfMember,
        SystemTemperature,
        SystemPressure);

    // Compute the outgoing direction of the diffracted order in question:
    Vector3D diffractedDirection = VL_Propagations.CalculateDiffractedDirectionTransmission(
        inputDirectionOfCurrentMember,
        refractiveIndexInFrontOfGrating,
        refractiveIndexBehindGrating,
        DiffractionOrder,
        Period,
        wavelengthOfMember);

    // Assign direction to corresponding member:
    currentMember.CentralDirection = diffractedDirection;

    // Apply efficiency, as provided by user in global parameters:
    currentMember *= Math.Sqrt(ScalarEfficiency);

// Continued in next slide.
```

Test the Code!

Main Function (Equidistant)

```
// Continued from last slide.

    // Re-insert member in alloted place:
    hfsReturn[memberIndex] = currentMember;
}

// Deliver result:
return hfsReturn;
```

Test the Code!

Main Function (Non-Equidistant & Rays)

```
// Declare output:
List<RayInformation> outputRays = new List<RayInformation>();

// Read in wavelength of current sample:
double currentWavelength = InputRay.Wavelength;

// Read in position and direction of current incoming sample:
Vector3D positionOfInputSample = InputRay.Position;
Vector3D incomingDirection = InputRay.Direction;

// Read refractive index in front of grating:
Complex refractiveIndexInFrontOfGrating =
    RayBundleInformation.MediumOfBundle.BaseMaterial.GetComplexRefractiveIndex(
    InputRay.Wavelength,
    SystemTemperature,
    SystemPressure);

// Read refractive index behind grating (single embedding medium):
Complex refractiveIndexBehindGrating = refractiveIndexInFrontOfGrating;

// Continued in next slide.
```

Test the Code!

Main Function (Non-Equidistant & Rays)

```
// Continued from last slide.

// Compute the outgoing direction of the diffraction order:
Vector3D diffractedDirection = VL_Propagations.CalculateDiffractedDirectionTransmission(
    incomingDirection,
    refractiveIndexInFrontOfGrating,
    refractiveIndexBehindGrating,
    DiffractionOrder,
    Period,
    currentWavelength);

// Include sample in output and assign direction to corresponding member:
outputRays.Add(new RayInformation(InputRay));
outputRays[outputRays.Count - 1].Direction = diffractedDirection;

// Adjust phase:
VectorD DeltaKappa = new VectorD(0, 0);
DeltaKappa.X = 2.0 * Math.PI * DiffractionOrder.X / Period.X;
DeltaKappa.Y = 2.0 * Math.PI * DiffractionOrder.Y / Period.Y;
outputRays[outputRays.Count - 1].AbsolutePhase +=
    new VectorD(positionOfInputSample.X, positionOfInputSample.Y) | DeltaKappa;

// Continued in next slide.
```

Test the Code!

Main Function (Non-Equidistant & Rays)

```
// Continued from last slide.

// Account for energy conservation:
double scalarSurfaceResponse = Math.Sqrt(ScalarEfficiency *
    (refractiveIndexInFrontOfGrating.Re * InputRay.Direction.Z) /
    (refractiveIndexBehindGrating.Re * diffractedDirection.Z));
outputRays[outputRays.Count - 1].EnergyConservationFactor *= scalarSurfaceResponse;

// Deliver result:
return outputRays.ToArray();
```

Document Information

title	How to Work with the Programmable Component and Example (Ideal Grating)
document code	CZT.0102
version	1.0
toolbox(es)	Starter Toolbox
VL version used for simulations	7.4.0.49
category	Feature Use Case
further reading	