Created in COMSOL Multiphysics 6.2

Petzval Lens Geometric Modulation Transfer Function

Introduction

In this tutorial we demonstrate how to use an Application Method to compute the modulation transfer function (MTF) of a lens. The optical transfer function (OTF) is a measure of an optical systems ability to resolve an object at a given spatial frequency. The OTF is defined as

$$\text{OTF} = \frac{\text{Image Contrast}}{\text{Object Contrast}} , \tag{1}$$

where the contrast is defined in terms of the intensity as

$$\text{Contrast} = \frac{I_{\text{max}} - I_{\text{min}}}{I_{\text{max}} + I_{\text{min}}} . \tag{2}$$

The OTF is a vector quantity including a phase term. When referring to the modulation transfer function, only the amplitude is considered.

The modulation transfer function at a given spatial frequency ν can be computed from the line spread function (LSF). In the context of an image of a point source (or, a collimated source at infinity), the LSF is an integration in a single direction of the point spread function (PSF). The MTF is then given by (Ref. 1)

$$\text{MTF}(\nu) = \sqrt{L_{\text{c}}^2(\nu) + L_{\text{s}}^2(\nu)} , \tag{3}$$

in which the expressions for L_{c} and L_{s} are

$$L_{\text{c}} = \frac{\int \text{LSF}(\delta) \cos 2\pi\nu\delta \, d\delta}{\int \text{LSF}(\delta) \, d\delta} , \tag{4}$$

and

$$L_{\text{s}} = \frac{\int \text{LSF}(\delta) \sin 2\pi\nu\delta \, d\delta}{\int \text{LSF}(\delta) \, d\delta} . \tag{5}$$

The LSF is given in terms of δ, the spatial location on the detector plane. The MTF, which is computed in the sagittal (x) or tangential (y) directions, is the response of the LSF to a signal that is spatially periodic at the frequency ν in either of these directions.

GEOMETRIC MODULATION TRANSFER FUNCTION

The MTF can be calculated using the results of a Geometrical Optics ray trace. In this method for computing the geometric MTF, the LSF is generated using the number density of ray intersections on the focal plane (that is, the spot diagram; see Figure 1). That is, for each of the sagittal (x) or tangential (y) directions, the LSF is given by

$$\mathrm{LSF_x} = \sum_{\delta - \frac{\Delta}{2}}^{\delta + \frac{\Delta}{2}} \begin{matrix} 1 \text{ if } \left(\delta - \frac{\Delta}{2} < x \le \delta + \frac{\Delta}{2}\right) \\ 0 \text{ otherwise} \end{matrix} \tag{6}$$

and

$$\mathrm{LSF_y} = \sum_{\delta - \frac{\Delta}{2}}^{\delta + \frac{\Delta}{2}} \begin{matrix} 1 \text{ if } \left(\delta - \frac{\Delta}{2} < y \le \delta + \frac{\Delta}{2}\right) \\ 0 \text{ otherwise} \end{matrix} \tag{7}$$

where δ is the distance relative to the centroid of the spot along either the x or y directions. The LSF of the spot diagram shown in Figure 1 can be seen in Figure 2.

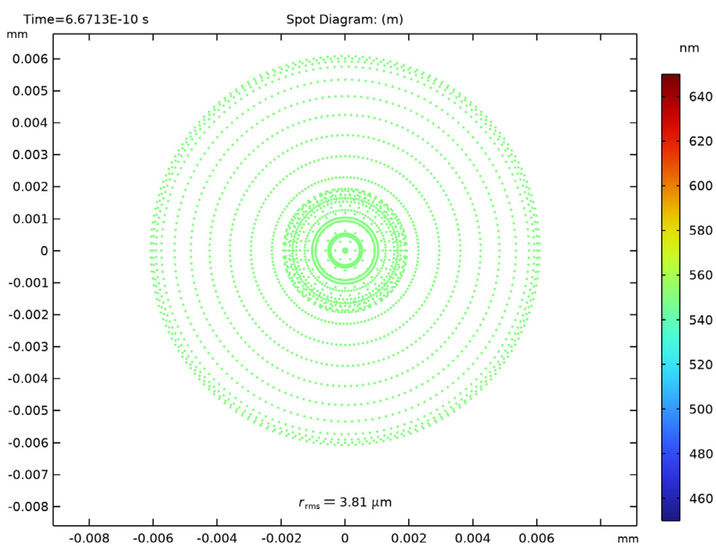

Figure 1: On-axis spot diagram. The line spread function (LSF) of this spot can be seen below.

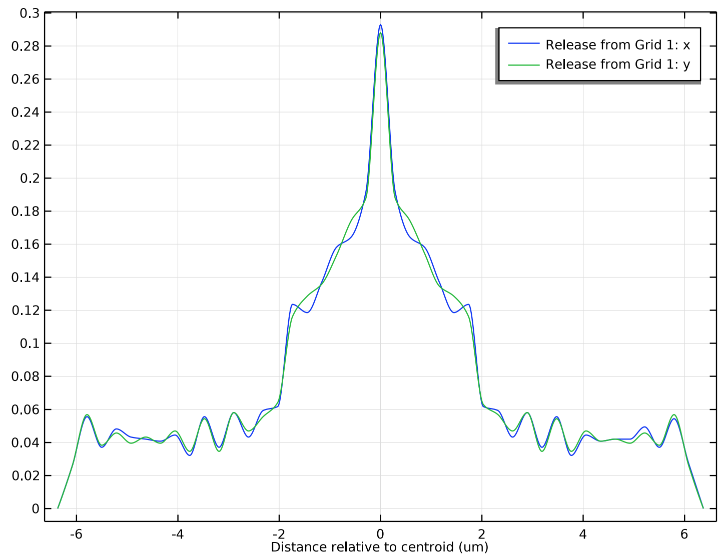

Figure 2: Line spread function (LSF) computed using the spot from the previous figure.

Model Definition

The model used for this example is the Petzval Lens, which can be found in the Ray Optics Module Application Library. The lens has a 100 mm focal length and includes a field flattening lens to provide good image quality over a ±10° field of view (Ref. 2). An overview can be seen in Figure 3.

The **Application Method** `computeMTF` is used to compute and plot the LSF and MTF curves and it can be added to the model using the **Application Builder**[1]. The method `computeMTF` also uses the **Utility Classes** `mtfutil` and `plotutil`. The Java code snippets for all of these methods can be found in the Appendix.

Detailed step-by-step instructions for creating and running the method can be found in the section Modeling Instructions. First the sagittal and tangential LSFs are captured in **Interpolation** functions with piecewise cubic interpolation. Next, a set of **Analytic** functions are created to be used in the integration of the expressions for L_c and L_s. Computing the MTF involves iterating over spatial frequency with a **Global Evaluation** of these expressions. The MTF results are also placed in **Interpolation** functions for plotting.

1. The Application Builder is only available in Windows® versions of COMSOL.

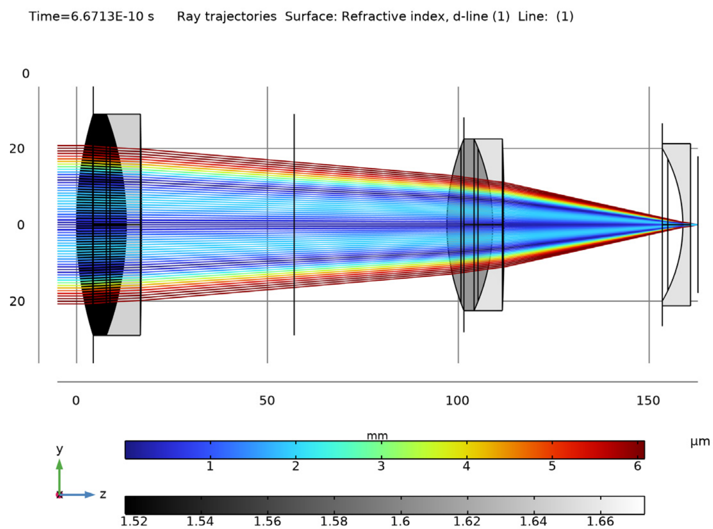

Figure 3: The Petzval lens with an on-axis ray trace.

The LSF and MTF plots each use a **1D Plot Group** with **Line Graphs** pointing to **Grid 1D** datasets to create the curves. Each of the grid datasets uses the corresponding LSF or MTF **Interpolation** functions as a source. The method creates and updates these features as required. The settings interface (see Figure 4) provide the user with control over several options.

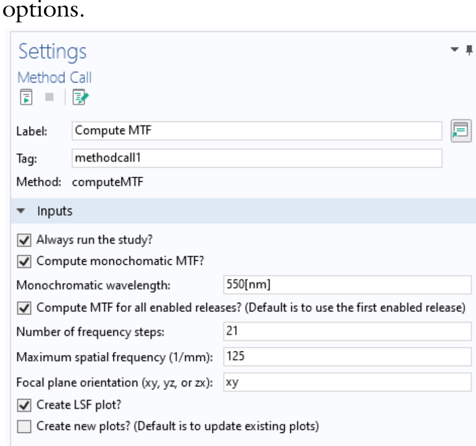

Figure 4: The Settings window for the MTF method call.

Results and Discussion

The default settings for this method will compute the MTF for the first active release feature. The spot diagram for this release was seen in Figure 1 together with the resulting LSF (Figure 2). These were used to compute the MTF seen in Figure 5. Note that the calculations presented here are all monochromatic with λ = 550 nm.

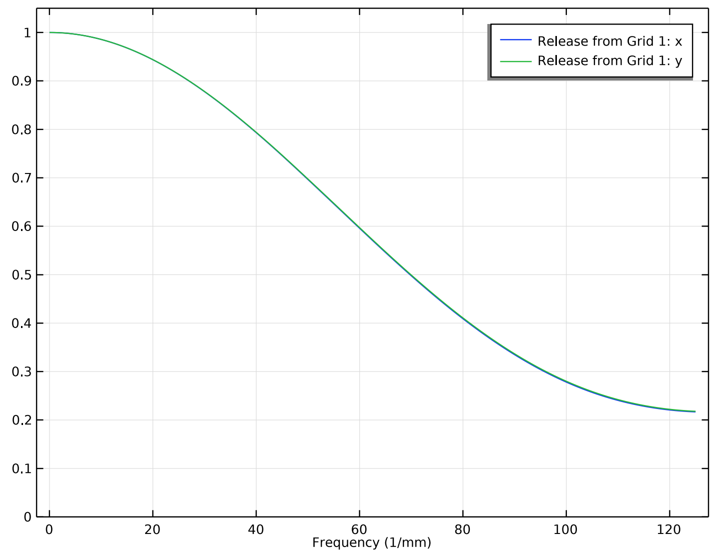

Figure 5: The modulation transfer function (MTF) of the Petzval lens for an on-axis field.

After enabling all three release features, the method can be run again. The spot diagram shown in Figure 6 now includes two off-axis fields. The line spread function for these fields are shown in Figure 7 and the resulting MTF is seen in Figure 8.

These results show how the spatial resolution of the Petzval lens can be analyzed as a function of field of view. The use of an MTF evaluation also allows the impact of the off-axis aberrations on the MTF in sagittal and tangential directions to be displayed.

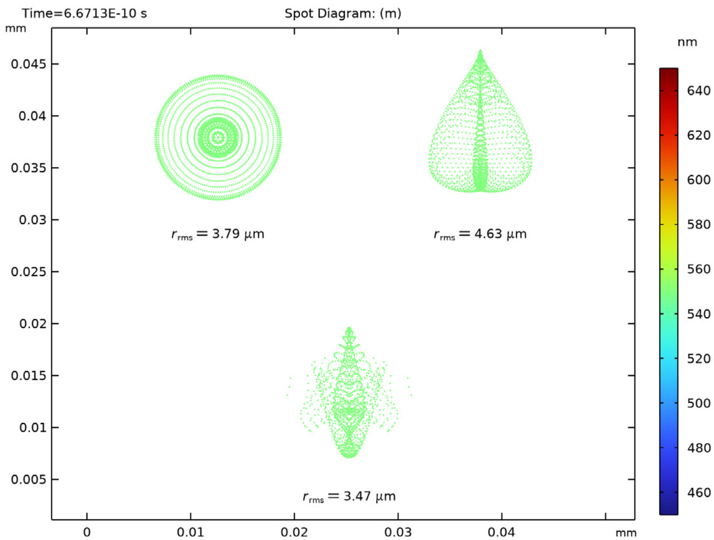

Figure 6: The Petzval lens spot diagram with three fields.

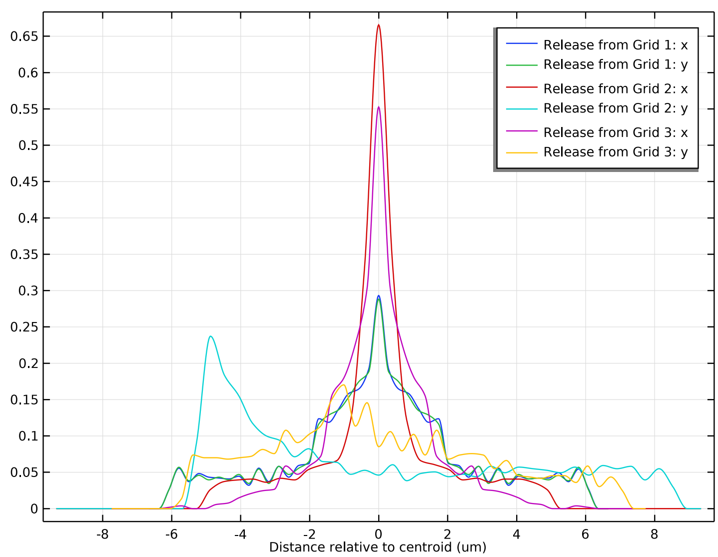

Figure 7: The line spread function (LSF) of the spots shown above.

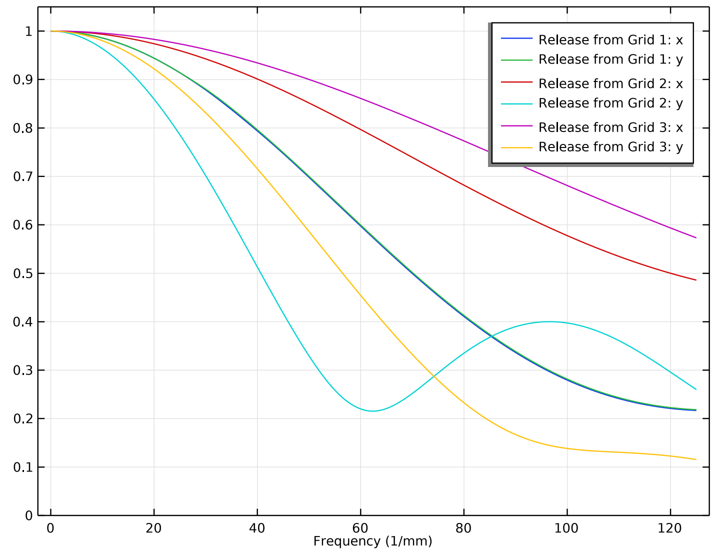

Figure 8: The modulation transfer function (MTF) of all three fields for the Petzval lens.

References

1. W.J. Smith, *Modern lens design*, vol. 2. New York, NY, USA: McGraw Hill, 2005.

2. M.J. Kidger, *Fundamental Optical Design*, SPIE Press, 2001.

Application Library path: `Ray_Optics_Module/Lenses_Cameras_and_Telescopes/petzval_lens_geometric_modulation_transfer_function`

APPLICATION LIBRARIES

1. From the **File** menu, choose **Application Libraries**.
2. In the **Application Libraries** window, select **Ray Optics Module>Lenses Cameras and Telescopes>petzval_lens** in the tree.
3. Click **Open**.

GLOBAL DEFINITIONS

To increase the accuracy of the MTF calculation we increase the number of rays to be traced.

Parameters 2: General

1. In the **Model Builder** window, under **Global Definitions** click **Parameters 2: General**.
2. In the **Settings** window for **Parameters**, locate the **Parameters** section.
3. In the table, enter the following settings:

Name	Expression	Value	Description
`N_ring`	`30`	30	`Number of hexapolar rings`

APPLICATION BUILDER

The geometric MTF can be computed using an Application Method. These may be added to an existing model using the **Application Builder**. Note that the **Application Builder** is only available in the Windows® version of the COMSOL Desktop.

In the **Home** toolbar, click **Application Builder**.

METHODS

The code for the computing and plotting the MTF can be simplified by using utility classes.

util1

1. In the **Home** toolbar, click **More Libraries** and choose **Utility Class**.
2. In the **Application Builder** window, click **util1**.
3. In the **Settings** window for **Utility Class**, type `mtfutil` in the **Name** text field.

mtfutil

1. Right-click **mtfutil** and choose **Edit**.

2 Copy the code for the MTF utilities `getReleaseList`, `createGroups`, `getSpotRadius`, and `evaluateLSFRays` and paste it into the **Utility Class** editor for `mtfutil`. This code may be found in the Appendix to this document.

3 In the **Home** toolbar, click **More Libraries** and choose **Utility Class**.

util1

1 In the **Application Builder** window, under **Libraries** click **util1**.

2 In the **Settings** window for **Utility Class**, type `plotutil` in the **Name** text field.

plotutil

1 Right-click **plotutil** and choose **Edit**.

2 Copy the code for the plot utilities `plotLSF`, `plotMTF`, `getPlotFeature`, and `updateDatasets` and paste it into the **Utility Class** editor for `plotutil` This code may be found in an appendix to this document.

GLOBAL METHOD

Next, create the Application Method `computeMTF`.

1 In the **Home** toolbar, click **New Method** and choose **Global Method**.

2 In the **Global Method** dialog box, type `computeMTF` in the **Name** text field.

3 Click **OK**.

computeMTF

1 In the **Application Builder** window, under **Methods** click **computeMTF**.

2 In the **Settings** window for **Method**, locate the **Inputs and Output** section.

3 Find the **Inputs** subsection. Click **Add**. Repeat this action to add a total of 9 **Inputs**.

4 In the table, enter the following settings:

Name	Type	Default	Description	Unit
runstudy	Boolean	true	Always run the study?	
ismono	Boolean	true	Compute monochromatic MTF?	
lambda_mono	String	550[nm]	Monochromatic wavelength	
useall	Boolean	false	Compute MTF for all enabled releases? (Default is to use the first enabled release)	
nustep_str	String	21	Number of frequency steps	

Name	Type	Default	Description	Unit
`numax_str`	String	`125`	`Maximum spatial frequency (1/mm)`	
`orient`	String	`xy`	`Focal plane orientation (xy, yz, or zx)`	
`lsfplot`	Boolean	`true`	`Create LSF plot?`	
`newplot`	Boolean	`false`	`Create new plots? (Default is to update existing plots)`	

5 Copy the code for method `computeMTF` and paste it into the **Method** editor. This code may be found in an appendix to this document.

METHODS

1 In the **Home** toolbar, click **Model Builder**.

Add `computeMTF` to the **Model Builder**.

2 Click **Method Call** and choose **computeMTF**.

GLOBAL DEFINITIONS

Compute MTF

1 In the **Model Builder** window, under **Global Definitions** click **ComputeMTF 1**.

2 In the **Settings** window for **Method Call**, type `Compute MTF` in the **Label** text field.

3 Click **Run**. First, run the method using the default settings.

RESULTS

LSF Plot

1 In the **Model Builder** window, under **Results** click **LSF Plot**.

2 In the **LSF Plot** toolbar, click **Plot**. Compare the result to Figure 2.

MTF Plot

1 In the **Model Builder** window, click **MTF Plot**.

2 In the **MTF Plot** toolbar, click **Plot**. Compare the result to Figure 5.

Next, compute the MTF with all release features. These were disabled when the first MTF calculation was made.

COMPONENT 1 (COMP1)

In the **Model Builder** window, expand the **Component 1 (comp1)** node.

GEOMETRICAL OPTICS (GOP)

In the **Model Builder** window, expand the **Component 1 (comp1)>Geometrical Optics (gop)** node.

Release from Grid 2, Release from Grid 3

1 In the **Model Builder** window, under **Component 1 (comp1)>Geometrical Optics (gop)**, Ctrl-click to select **Release from Grid 2** and **Release from Grid 3**.

2 Right-click and choose **Enable**.

GLOBAL DEFINITIONS

Compute MTF

1 In the **Model Builder** window, under **Global Definitions** click **Compute MTF**.

2 In the **Settings** window for **Method Call**, locate the **Inputs** section.

3 Select the **Compute MTF for all enabled releases? (Default is to use the first enabled release)** check box. The MTF calculation will now include all enabled releases.

4 Click **Run**.

RESULTS

LSF Plot

1 In the **Model Builder** window, under **Results** click **LSF Plot**.

2 In the **LSF Plot** toolbar, click **Plot**. Compare the result to Figure 7.

MTF Plot

1 In the **Model Builder** window, click **MTF Plot**.

2 In the **MTF Plot** toolbar, click **Plot**. Compare the result to Figure 8.

Appendix: Java Code

METHODS

computeMTF

```
/** Compute and plot the Modulation Transfer Function (MTF) of a lens.
Inputs are:
* runstudy - Always run the study?
* ismono - Compute monochromatic MTF?
* lambda_mono - Monochromatic wavelength
* useall - Compute MTF for all enabled releases? (Default is to use the first
enabled release)
```

```
* nustep_str - Number of frequency steps
* numax_str - Maximum spatial frequency (1/mm)
* orient - Focal plane orientation (xy, yz, or zx)
* lsfplot - Create LSF plot?
* newplot - Create new plots? (Default is to update existing plots)*/

long timestart = timeStamp();
message("Starting MTF calculation...");

Physics gop = getCurrentPhysics();
if (gop == null) {
 return;
}
if (findIn(gop.getType(), "GeometricalOptics") == -1) {
 error("A Geometrical Optics interface does not exist.");
}
ModelNode comp = getCurrentComponent();
GeomSequence geom = comp.geom(comp.geom().tags()[0]);
if (geom.getSDim() != 3) {
 error("The model does not have a 3D geometry.");
}
double geomscale = model.param().evaluate("1["+geom.lengthUnit()+"]/1[m]");

/** Set the focal plane orientation. This method assumes that the
"last" time step is on the focal plane and that the sagittal and
meridional directions are one of the xy, yz, zx axis pairs. */
String[] xystr = new String[2];
if (findIn(orient, "xy") == 0) {
 xystr = new String[]{"x", "y"};
}
else if (findIn(orient, "yz") == 0) {
 xystr = new String[]{"y", "z"};
}
else if (findIn(orient, "zx") == 0) {
 xystr = new String[]{"z", "x"};
}
else {
 error("The focal plane orientation must be one of ’xy’, ’yz’ or ’zx’.");
}

// Set the frequency step size and the number of LSF bins
int nu_nstep = Integer.parseInt(nustep_str);
double numax = Double.parseDouble(numax_str);
double nu_step = numax/(nu_nstep-1);
double[] nu = new double[nu_nstep];
for (int k = 0; k < nu_nstep; k++) {
 nu[k] = nu_step*k;
}
int nbin = nu_nstep; // The number of LSF bins is actually 2*nu_nstep

// Get list of all active release features
String[][] releaseList = mtfutil.getReleaseList(gop, useall);
int num = releaseList.length;

// Create node groups for model housekeeping
```

```
NodeGroupList groupList = mtfutil.createGroups(releaseList);

if (ismono) {
 String lamtype = gop.prop("WavelengthDistribution")
 .getString("WavelengthDistribution");
 if (findIn(lamtype, "Monochromatic") == -1) {
 gop.prop("WavelengthDistribution")
 .setIndex("WavelengthDistribution", "Monochromatic", 0);
 runstudy = true;
 }
 String lambda_current = gop.feature("op1").getString("lambda0");
 if (findIn(lambda_current, lambda_mono) == -1) {
 gop.feature("op1").set("lambda0", lambda_mono);
 runstudy = true;
 }
}

if (runstudy) {
 long timestart_study = timeStamp();
 model.study("std1").run();
 long timeend_study = timeStamp();
 message(" Time to run study = "+(timeend_study-timestart_study)/1e3);
}

// Get maximum spot radius (r_max, in microns) and bin size (dxy)
double[][] rmax0 = mtfutil.getSpotRadius(releaseList, num, geomscale, nbin);

// Create (or update) line spread functions (LSF)
long timestart_lsf = timeStamp();
FunctionFeatureList functionList = model.func();
int[] ii = {0, 1};
String[] cossin = {"cos", "sin"};
String[][] evalList = new String[2][num];
String[][] lsfList = new String[2][num];
String[][] lsfTag = new String[2][num];
String[][] mfunList = new String[4*num][4];
String[][] intList = new String[4*num][2];
String[][] mtfList = new String[2][num];
String[][] mtfTag = new String[2][num];
// String[][] lineTag = new String[2][num];
int m = 0;
for (int j = 0; j < num; j++) {
 for (int i : ii) {
 evalList[i][j] = "gevLSF"+xystr[i]+(j+1);
 lsfList[i][j] = "LSF"+xystr[i]+(j+1);
 mtfList[i][j] = "MTF"+xystr[i]+(j+1);
 mtfTag[i][j] = "mtf"+xystr[i]+(j+1);
 lsfTag[i][j] = "lsf"+xystr[i]+(j+1);
 String arg1 = "(2*pi*nu*"+xystr[i]+"/1e3)";
 String arg2 = "("+xystr[i]+",nu_local)";
 String rmaxstr = Double.toString(rmax0[j][0]);
 String rminmax = "-"+rmaxstr+", "+rmaxstr;
 /** Create (and populate) the cosine and sine multiplication analytic
 functions, that is, functions with the form
 LSF[x/y][k]([x/y])*[cos/sin](2*pi*nu*[x,y]/1e3).
```

```
The spatial frequency is an input to these functions, so that the
range of spatial frequencies between 0 and numax_local can be sampled
in the integration below. */
/** Also, create a list of evaluations to compute the integrals as a
function of spatial frequency, that is:
integrate(L[cos/sin][x/y][k]([x,y],nu_local), [x,y], -rmax, rmax)/...
integrate(LSF[x/y]([x/y]), [x,y], -rmax, rmax).
Set the current spatial frequency to "nu_local" (to avoid a name
conflict with built-in variables). This will be updated in a result
parameter node as we cycle through frequencies below. */
for (int k : ii) {
String Lstr = "L"+cossin[k]+xystr[i]+(j+1);
String LSFstr = "LSF"+xystr[i]+(j+1)+"("+xystr[i]+")";
// Create the analytic function features
FunctionFeature mFeature;
mfunList[m] = new String[]{Lstr, LSFstr+"*"+cossin[k]+arg1,
xystr[i]+", nu"};
if (functionList.index(mfunList[m][0]) == -1) {
mFeature = functionList.create(mfunList[m][0], "Analytic");
mFeature.label(mfunList[m][0]);
groupList.get("mtfgrp"+(j+1)).add("func", mfunList[m][0]);
}
else {
mFeature = functionList.get(mfunList[m][0]);
}
with(mFeature);
set("funcname", mfunList[m][0]);
set("expr", mfunList[m][1]);
set("args", mfunList[m][2]);
setIndex("plotargs", -rmax0[j][0], 0, 1);
setIndex("plotargs", rmax0[j][0], 0, 2);
setIndex("plotargs", 0, 1, 1);
setIndex("plotargs", numax_str, 1, 2);
endwith();
String intstr1 = "integrate("+Lstr+arg2+","+xystr[i]+","+rminmax+")";
String intstr2 = "integrate("+LSFstr+","+xystr[i]+","+rminmax+")";
intList[m] = new String[]{intstr1+"/"+intstr2, "L"+xystr[i]+(j+1)};
m++;
}
// Evaluate the LSF
int nn = (int) (rmax0[j][2]);
double[][] Lxy = mtfutil.evaluateLSFRays(evalList[i][j], releaseList[j][0],
j, xystr[i], rmax0[j][0], rmax0[j][1], nn);
// Populate the LSF interpolation table
FunctionFeature functionFeature;
if (functionList.index(lsfList[i][j]) == -1) {
functionFeature = functionList.create(lsfList[i][j], "Interpolation");
functionFeature.label(lsfList[i][j]);
groupList.get("mtfgrp"+(j+1)).add("func", lsfList[i][j]);
with(functionFeature);
set("funcname", lsfList[i][j]);
set("interp", "piecewisecubic");
set("extrap", "const");
set("defineprimfun", true);
endwith();
```

```
 }
 else {
 functionFeature = functionList.get(lsfList[i][j]);
 }
 with(functionFeature);
 set("table", new String[0][0]);
 int nk = (int) (rmax0[j][2]);
 for (int k = 0; k < nk; k++) {
 setIndex("table", Lxy[k][0], k, 0);
 setIndex("table", Lxy[k][1], k, 1);
 }
 endwith();
 if (lsfplot) { // Create or update the LSF grid datasets
 String pmin = toString(-rmax0[j][0]);
 String pmax = toString(rmax0[j][0]);
 String[] params = {pmin, pmax, xystr[i]+"_out"};
 plotutil.updateDatsets(i, j, params, lsfList, lsfTag, groupList);
 }
 }
}

if (lsfplot) { // Create or update the LSF plot group
 plotutil.plotLSF(releaseList, lsfList, lsfTag, newplot);
}

// Update the integration evaluations
NumericalFeature evalFeature0 = model.result().numerical().get("gevMTF");
with(evalFeature0);
 set("expr", new String[]{});
 set("descr", new String[]{});
 for (int i = 0; i < m; i++) {
 setIndex("expr", intList[i][0], i);
 setIndex("descr", intList[i][1], i);
 }
endwith();
double timeend_lsf = timeStamp();
message(" Time to compute LSFs = "+(timeend_lsf-timestart_lsf)/1e3);

// Update the solution to ensure that the LSF is evaluated within the current
study.
double timestart_update = timeStamp();
model.sol("sol1").updateSolution();
double timeend_update = timeStamp();
message(" Time to update solution = "+(timeend_update-timestart_update)/1e3);

// Loop through spatial frequency ("nu_local") and evaluate the MTF
long timestart_mtf = timeStamp();
double[][][] MTFxy = new double[2][num][nu_nstep];
for (int k = 0; k < nu_nstep; k++) {
 model.result().param().set("nu_local", Double.toString(nu[k]));
 double[][] Lxy_all = evalFeature0.getReal();
 for (int j = 0; j < num; j++) {
 double Lxc = Lxy_all[4*j][0];
 double Lxs = Lxy_all[4*j+1][0];
 double Lyc = Lxy_all[4*j+2][0];
```

```
 double Lys = Lxy_all[4*j+3][0];
 MTFxy[0][j][k] = Math.sqrt(Lxc*Lxc+Lxs*Lxs);
 MTFxy[1][j][k] = Math.sqrt(Lyc*Lyc+Lys*Lys);
 }
}
long timestop_mtf = timeStamp();
message(" Time to compute MTF = "+(timestop_mtf-timestart_mtf)/1e3);

// Populate interpolation tables for MTF
long timestart_mtftable = timeStamp();
for (int j = 0; j < num; j++) {
 for (int i : ii) {
 // Create MTF interpolation table
 FunctionFeature functionFeature;
 if (functionList.index(mtfList[i][j]) == -1) {
 functionFeature = functionList.create(mtfList[i][j], "Interpolation");
 functionFeature.label(mtfList[i][j]);
 groupList.get("mtfgrp").add("func", mtfList[i][j]);
 }
 else {
 functionFeature = functionList.get(mtfList[i][j]);
 }
 with(functionFeature);
 set("funcname", mtfList[i][j]);
 set("interp", "cubicspline");
 set("extrap", "const");
 set("table", new String[0][0]);
 for (int k = 0; k < nu_nstep; k++) {
 setIndex("table", nu[k], k, 0);
 setIndex("table", MTFxy[i][j][k], k, 1);
 }
 endwith();
 // Create or update the MTF grid datasets
 String pmin = "0";
 String pmax = toString(numax);
 String[] params = {pmin, pmax, "nu_out"};
 plotutil.updateDatsets(i, j, params, mtfList, mtfTag, groupList);
 }
}
long timestop_mtftable = timeStamp();
message(" Time to update MTF table = "+(timestop_mtftable-timestart_mtftable)/
1e3);

// Create or update the MTF plot group
plotutil.plotMTF(releaseList, numax, mtfList, mtfTag, newplot);

long timestop = timeStamp();
message(" Total time = "+(timestop-timestart)/1e3);
```

UTILITY CLASSES

mtfutil

```
/** Evaluate the LSF using the ray dataset. */
public static double[][] evaluateLSFRays(String evalTag, String releaseTag,
 int j, String xystr, double rmax0,
 double dxy, int nn) {
 NodeGroupList groupList = model.nodeGroup();
 NumericalFeatureList numericalList = model.result().numerical();
 NumericalFeature evalFeature;
 if (numericalList.index(evalTag) == -1) {
 evalFeature = numericalList.create(evalTag, "EvalGlobal");
 evalFeature.label(evalTag);
 groupList.get("numgrp").add("numerical", evalTag);
 }
 else {
 evalFeature = numericalList.get(evalTag);
 }
 double[] xymid = new double[nn];
 double xymin = 0;
 double xymax = 0;
 String qevalstr = "(1e6*(q"+xystr+"-gop."+releaseTag+".qave"+xystr+")";
 with(evalFeature);
 set("data", "ray1");
 set("innerinput", "last");
 set("expr", new String[]{});
 set("descr", new String[]{});
 for (int k = 0; k < nn; k++) {
 xymid[k] = -rmax0+(k*dxy);
 xymin = xymid[k]-dxy/2;
 xymax = xymid[k]+dxy/2;
 String evalstr_m = qevalstr+">"+toString(xymin)+")";
 String evalstr_p = qevalstr+"<="+toString(xymax)+")";
 String evalstr = "gop.gopop1("+evalstr_m+"&&"+evalstr_p
 +"&&gop.prf=="+(j+1)+")";
 setIndex("expr", evalstr, k);
 }
 endwith();
 double[][] Lxyall = evalFeature.getReal();
 double Lxymax = 0;
 for (int k = 0; k < nn; k++) {
 Lxymax = Math.max(Lxyall[k][0], Lxymax);
 }
 double[][] Lxy = new double[nn][2];
 for (int k = 0; k < nn; k++) {
 Lxy[k][0] = xymid[k];
 Lxy[k][1] = Lxyall[k][0]/Lxymax;
 }
 return Lxy;
}


/** Get the list of release features. */
public static String[][] getReleaseList(Physics gop, boolean useall) {
 PhysicsFeatureList gopFeature = gop.feature();
```

```
 int numAll = 0;
 String[] releaseTagAll = new String[gopFeature.size()];
 for (PhysicsFeature feature : gopFeature) {
 if (findIn(feature.getType(), "Release") >= 0 && feature.isActive()) {
 releaseTagAll[numAll] = feature.tag();
 numAll++;
 }
 }
 int num = (useall ? numAll : 1);
 String[] releaseTag = new String[num];
 String[][] releaseList = new String[num][2];
 if (useall) {
 for (int i = 0; i < num; i++) {
 releaseTag[i] = releaseTagAll[i];
 releaseList[i][0] = releaseTagAll[i];
 releaseList[i][1] = gop.feature(releaseTagAll[i]).label();
 }
 }
 else {
 releaseTag[0] = releaseTagAll[0];
 releaseList[0][0] = releaseTagAll[0];
 releaseList[0][1] = gop.feature(releaseTagAll[0]).label();
 }
 if (useall) {
 for (int i = 0; i < num; i++) {
 gop.feature(releaseTag[i]).active(true);
 }
 }
 else {
 for (int i = 0; i < numAll; i++) {
 if (releaseTagAll[i] == releaseTag[0]) {
 gop.feature(releaseTagAll[i]).active(true);
 }
 else {
 gop.feature(releaseTagAll[i]).active(false);
 }
 }
 }
 return releaseList;
}

/** Create groups for model housekeeping. */
public static NodeGroupList createGroups(String[][] releaseList) {
 int num = releaseList.length;
 NodeGroupList groupList = model.nodeGroup();
 String[] mtftableGroupTag = {"mtfgrp", "GlobalDefinitions", "", "MTF Tables"};
 String[] numGroupTag = {"numgrp", "Results", "numerical", "MTF Evaluations"};
 String[] dataGroupTag = {"datagrp", "Results", "dataset", "MTF Data"};
 String[] mtfGroupTag = new String[num];
 for (int j = 0; j < num; j++) {
 mtfGroupTag[j] = "mtfgrp"+(j+1);
 }
 if (groupList.index(mtftableGroupTag[0]) == -1) {
 NodeGroup mtftableGroup = groupList.create(mtftableGroupTag[0],
 mtftableGroupTag[1]);
```

```
 mtftableGroup.label(mtftableGroupTag[3]);
 }
 if (groupList.index(numGroupTag[0]) == -1) {
 NodeGroup numGroup = groupList.create(numGroupTag[0],
 numGroupTag[1]);
 numGroup.set("type", numGroupTag[2]);
 numGroup.label(numGroupTag[3]);
 }
 for (int j = 0; j < num; j++) {
 if (groupList.index(mtfGroupTag[j]) == -1) {
 NodeGroup mtfGroup = groupList.create(mtfGroupTag[j],
 "GlobalDefinitions");
 mtfGroup.label("MTF: "+releaseList[j][1]);
 }
 }
 if (groupList.index(dataGroupTag[0]) == -1) {
 NodeGroup dataGroup = groupList.create(dataGroupTag[0], dataGroupTag[1]);
 dataGroup.set("type", dataGroupTag[2]);
 dataGroup.label(dataGroupTag[3]);
 }
 return groupList;
}


/** Get the maximum spot radius (rmax0) accounting for the bin size (dxy). */
public static double[][] getSpotRadius(String[][] releaseList, int num,
 double geomscale, int nbin) {
 NodeGroupList groupList = model.nodeGroup();
 NumericalFeatureList numericalList = model.result().numerical();
 NumericalFeature evalFeature0;
 if (numericalList.index("gevMTF") == -1) {
 evalFeature0 = numericalList.create("gevMTF", "EvalGlobal");
 evalFeature0.label("gevMTF");
 groupList.get("numgrp").add("numerical", "gevMTF");
 }
 else {
 evalFeature0 = numericalList.get("gevMTF");
 }
 with(evalFeature0);
 set("data", "ray1");
 set("innerinput", "last");
 set("expr", new String[]{});
 set("descr", new String[]{});
 for (int j = 0; j < num; j++) {
 setIndex("expr", "gop."+releaseList[j][0]+".rmaxall", j);
 }
 endwith();
 double[][] rmaxall = evalFeature0.getReal();
 double[][] rmax0 = new double[num][3];
 double rmax0_;
 double[] rmax = new double[num];
 double[] dxy = new double[num];
 int[] nn = new int[num];
 for (int j = 0; j < num; j++) {
 rmax[j] = 1e6*rmaxall[j][0]*geomscale;
 dxy[j] = rmax[j]/nbin;
```

```
 rmax0_ = Math.ceil((rmax[j]+dxy[j])/dxy[j])*dxy[j];
 nn[j] = (int) (2*rmax0_/dxy[j]+1);
 rmax0[j][0] = rmax0_;
 rmax0[j][1] = dxy[j];
 rmax0[j][2] = nn[j];
 }
 return rmax0;
}
```

plotutil

```
/** Create or update LSF plots. */
public static void plotLSF(String[][] releaseList, String[][] lsfList,
 String[][] lsfTag, boolean newplot) {
 // Create or get the LSF plot group
 ResultFeature lsfPlot = getPlotFeature("LSF Plot", newplot);
 // Create or update line graphs
 ResultFeatureList lsfPlotList = lsfPlot.feature();
 int[] ii = {0, 1};
 String[] xystr = new String[]{"x", "y"};
 for (int j = 0; j < releaseList.length; j++) {
 for (int i : ii) {
 if (lsfPlotList.index(lsfTag[i][j]) == -1) {
 ResultFeature lsfLine = lsfPlot.create(lsfTag[i][j], "LineGraph");
 lsfLine.label(lsfList[i][j]);
 String expr_str = lsfList[i][j]+"("+xystr[i]+"_out)";
 expr_str = expr_str+"/"+lsfList[i][j]+"_prim(1e6)";
 with(lsfLine);
 set("xdata", "expr");
 set("expr", expr_str);
 set("xdataexpr", xystr[i]+"_out");
 set("xdatadescractive", true);
 set("xdatadescr", "Distance relative to centroid (um)");
 set("data", lsfTag[i][j]);
 set("legend", true);
 set("autodescr", true);
 set("autosolution", false);
 set("descractive", true);
 set("descr", releaseList[j][1]+": "+xystr[i]);
 set("smooth", "none");
 set("resolution", "norefine");
 endwith();
 }
 }
 }
}

/** Create or update MTF plots. */
public static void plotMTF(String[][] releaseList, double numax,
 String[][] mtfList, String[][] mtfTag,
 boolean newplot) {
 // Create or get the MTF plot group
 ResultFeature mtfPlot = getPlotFeature("MTF Plot", newplot);
 // Update the axes
```

```
 with(mtfPlot);
 set("axislimits", true);
 set("xmin", -0.02*numax);
 set("xmax", 1.02*numax);
 set("ymin", 0);
 set("ymax", 1.05);
 set("yminsec", 0);
 set("ymaxsec", 1.05);
 endwith();
 // Create or update line graphs
 ResultFeatureList mtfPlotList = mtfPlot.feature();
 int[] ii = {0, 1};
 String[] xystr = new String[]{"x", "y"};
 for (int j = 0; j < releaseList.length; j++) {
 for (int i : ii) {
 if (mtfPlotList.index(mtfTag[i][j]) == -1) {
 ResultFeature mtfLine = mtfPlot.create(mtfTag[i][j], "LineGraph");
 mtfLine.label(mtfList[i][j]);
 with(mtfLine);
 set("xdata", "expr");
 set("expr", mtfList[i][j]+"(nu_out) ");
 set("xdataexpr", "nu_out");
 set("xdatadescractive", true);
 set("xdatadescr", "Frequency (cycles/mm)");
 set("data", mtfTag[i][j]);
 set("legend", true);
 set("autodescr", true);
 set("autosolution", false);
 set("descractive", true);
 set("descr", releaseList[j][1]+": "+xystr[i]);
 set("smooth", "none");
 set("resolution", "norefine");
 endwith();
 }
 }
 }
}

/** Create or get the plot feature. */
public static ResultFeature getPlotFeature(String pLabel, boolean newplot) {
 ResultFeature mtfPlot;
 String pTag = "";
 String pLabel_in = pLabel;
 String[] rTag = model.result().tags();
 int nplt = 0;
 for (int i = 0; i < rTag.length; i++) {
 if (findIn(model.result(rTag[i]).label(), pLabel_in) > -1) {
 pTag = rTag[i];
 pLabel = model.result(rTag[i]).label();
 nplt++;
 }
 }
 if (nplt > 0) {
 if (newplot) {
 if (findIn(substring(pLabel, pLabel.length()-1, 1), "t") > -1) {
```

```
 pLabel = pLabel+" 1";
 }
 else {
 int np = Integer.parseInt(substring(pLabel, pLabel.length()-1, 1))+1;
 pLabel = substring(pLabel, 0, pLabel.length()-1)+np;
 }
 }
 }
 if (newplot || pTag.length() == 0) {
 pTag = model.result().uniquetag("pg");
 mtfPlot = model.result().create(pTag, "PlotGroup1D");
 mtfPlot.label(pLabel);
 with(mtfPlot);
 set("data", "none");
 set("titletype", "none");
 set("legendpos", "upperright");
 endwith();
 }
 else {
 mtfPlot = model.result().get(pTag);
 }
 return mtfPlot;
}


/** Create or update the grid datasets. */
public static void updateDatsets(int i, int j, String[] param,
 String[][] labelList, String[][] tagList,
 NodeGroupList groupList) {
 DatasetFeature dataFeature;
 if (model.result().dataset().index(tagList[i][j]) == -1) {
 dataFeature = model.result().dataset().create(tagList[i][j], "Grid1D");
 dataFeature.label(labelList[i][j]);
 groupList.get("datagrp").add("dataset", tagList[i][j]);
 }
 else {
 dataFeature = model.result().dataset().get(tagList[i][j]);
 }
 with(dataFeature);
 set("source", "function");
 set("function", labelList[i][j]);
 set("parmin1", param[0]);
 set("parmax1", param[1]);
 set("par1", param[2]);
 endwith();
}
```