Getting Started with *LensLab*

Introduction to Chapter 1

This chapter will provide you with an introduction to *LensLab*'s basic features. By learning the functions introduced in this chapter, you will have a foothold for using *LensLab*. First we learn how to define components and rays using *LensLab*'s built-in functions. Later, in Section 1.6, we use components and rays together for ray tracing. Before beginning, we must first load *LensLab* into memory.

1 Loading LensLab

Make sure that the *LensLab* package is located either in the home directory, or on a directory path recognized by *Mathematica* for packages. The *LensLab* package is named **`LensLab.m`** and located in the *LensLab* directory, and the *LensLab* package is loaded with the following expression.

```
Needs["LensLab`LensLab`"]

    LensLab version 1.5 is now loaded.
```

This loading process should only take a few seconds. In addition to being loaded as a package, the **`LensLab.m`** file is formatted as a *Mathematica* notebook. The *LensLab* source code is made accessible so that you can develop new functions of your own by studying *LensLab*'s built-in functions. This is particularly helpful when you wish to model new component ideas in *LensLab*. However, you should receive permission from Optica Software before distributing any user-created functions that are derived from the *LensLab* source code. The unauthorized distribution of *LensLab*-derived code may be a *LensLab* license agreement violation or a copyright infringement.

2 Creating Components and Rays

There are two distinct types of *Mathematica* objects defined and used with *LensLab*, namely **`Component`** and **`Ray`**. **`Component`** describes optical components and **`Ray`** describes rays. In this section we learn how to create **`Component`** and **`Ray`**.

■ Creating a Component

LensLab has many built-in functions for creating **`Component`** objects. An example of such a component function is **`PlanoConvexLens`**. First we define **`PlanoConvexLens`**.

`PlanoConvexLens[` *focallength*, *aperture*, *thickness*, *options***`]`** refers to a lens with a planar surface on one side and a convex spherical surface on the other side.

`PlanoConvexLens` outputs a **`Component`** object carrying the description for a planoconvex lens element. Next we use **`PlanoConvexLens`**.

```
PlanoConvexLens[100,50,10]
```

```
PlanoConvexLens
```

Hidden behind the "PlanoConvexLens" output returned to the screen is a large expression headed by **`Component`** that describes a planoconvex lens having a focal length of 100 millimeters, a circular aperture 50 millimeters in diameter, and a lens center thickness of 10 millimeters. Instead of showing this entire expression, *LensLab* normally hides the contents of **`Component`** by outputting back to the screen a single descriptive label of the component. You can learn about the anatomy of **`Component`** in Chapter 10.

You can use **`Component`** objects immediately after creating them for doing ray tracing and rendering, or you can assign created **`Component`** objects to variables for future work. While *focallength*, *aperture*, and *thickness* are all parameters given explicitly in the inputs of **`PlanoConvexLens`**, other implicit parameters (such as the type of refractive material) are options of **`PlanoConvexLens`**.

We use **`Options`** to see the default options of **`PlanoConvexLens`**.

```
Options[PlanoConvexLens]

 {ComponentMedium → BK7, Temperature → 20.,
  Transmittance → 100, GraphicDesign → Automatic,
  CurvatureDirection → Front, DesignWaveLength → 0.5461,
  OffAxis → {0., 0.}, SwitchDirectionOnReflection → True,
  Automatic → {SurfaceRendering → Empty,
    EdgeRendering → Mesh, CrossRendering → {{Fill, Trace}}},
  Sketch → {SurfaceRendering → Trace, EdgeRendering → Empty,
    CrossRendering → {{Fill, Trace}}}, Wire → {SurfaceRendering → Mesh,
    EdgeRendering → Mesh, CrossRendering → {{Fill, Trace}}},
  Solid → {SurfaceRendering → {{Fill, Trace}},
    EdgeRendering → Fill, CrossRendering → Empty}}
```

Notice that the first option is **`ComponentMedium -> BK7`**. This option indicates that the refractive material of the planoconvex lens is assumed to be made of BK7 glass material. See Section 6.9 for more information about the refractive index options used in *LensLab*. Other options of **`PlanoConvexLens`** are described in Sections 3.9, 3.11, 6.3, and 6.4.

■ Creating a Ray

LensLab also has many built-in functions for creating rays. Called ray sources, these functions generate various types of ray-tracing light sources. The simplest type of ray source is **`Ray[]`**. First we define **`Ray`**.

> **`Ray[`***options***`]`** contains rules that characterize a single ray of light as it propagates through component surfaces and creates a single **`Ray`** object with its starting position at the origin and directed down the positive *x* axis.

Next we use **`Ray`**.

```
Ray[]

    Ray
```

`Ray` is used by *LensLab* to initiate a single ray trace down the *x* axis in space. Like **`Component`**, *LensLab* normally hides the contents of **`Ray`** by outputting the "`Ray`" message back to the screen. After a ray trace has been carried out, there will be as many new **`Ray`** objects created as there are optical surfaces encountered by the single ray. Each **`Ray`** object contains information about a particular intersection point with an optical surface. You can learn about the anatomy of **`Ray`** objects in Chapter 9.

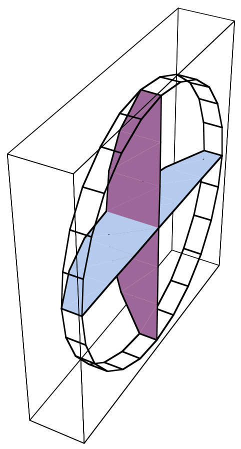
3 The DrawSystem Function

The most important function in *LensLab* is **`DrawSystem`**. **`DrawSystem`** is used for both rendering and ray tracing. We now define **`DrawSystem`**.

`DrawSystem[`*objectset*, *options***`]`** takes an object set containing a mixed list of **`Ray`**, **`Component`**, and **`OpticalSystem`** objects, propagates the rays through the components, and renders the ray-tracing result according to **`PlotType`**, **`RayChoice`**, **`ShowRange`**, and **`ColorView`** options.

Next we render the planoconvex lens with **`DrawSystem`**.

```
DrawSystem[PlanoConvexLens[100,50,10]]
```



```
{{PlanoConvexLens}, -OpticsGraphics-}
```

PlanoConvexLens is created with its first surface at the origin and its second surface positioned down the positive *x* axis. The *aperture* parameter of **PlanoConvexLens** may designate a circle, rectangle, or polygon, depending on the number and type of elements listed by it. Here we create a circular lens having a diameter of 50 millimeters by using **50** in the *aperture* parameter. You can learn more about shaping the edges of components in Section 6.8. After rendering the planoconvex lens, **DrawSystem** returns the **Component** object back to its output. You can suppress the printed text by including a semicolon at the end of the input expression.

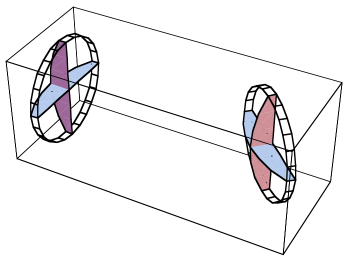
4 The Move Function

To put an optical component at a location other than $x = 0$, $y = 0$, $z = 0$, **Move** is needed. You can also use **Move** to move rays around in space. We first define **Move**.

Move[*objectset*, **{***x*, *y***}**, *rotationangle*, *options***]** is used to move the relative position and orientation of a set of components and rays within a horizontal plane.

The *rotationangle* determines the angular orientation of the object set within the horizontal plane. The option **TwistAngle ->** # can be used to specify a rotation around the axis of orientation. Other related commands used for positioning components and rays include: **Move3D**, **MoveDirected**, **MoveDirected3D**, **MoveLinear**, **MoveLinear3D**, **MoveReflected**, and **MoveReflected3D**. We next use **Move** in **DrawSystem** to draw two lenses spaced apart from each other. We place the first one at $x = 0$, $y = 0$, with no rotation, and a second one at $x = 100$, $y = 0$, with a 45-degree rotation. See Chapter 4 to learn more about the various **Move** functions.

```
DrawSystem[{
    PlanoConvexLens[100,50,10],
    Move[PlanoConvexLens[100,50,10],100,45]}];
```



This time the printed text has been surpressed with a semicolon at the end of **`DrawSystem`**.

5 The ShowSystem Function

To avoid recalculating the optical system, you can display any previously calculated system more quickly by using **`ShowSystem`**. First we define **`ShowSystem`**.

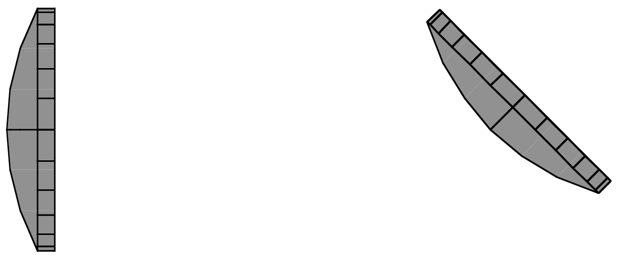
> **`ShowSystem[`***objectset*, *options***`]`** takes *objectset* made up of **`Ray`**, **`Component`**, and **`OpticalSystem`** objects and renders them according to **`PlotType`**, **`RayChoice`**, **`ShowRange`**, and **`ColorView`** options.

`ShowSystem` and **`DrawSystem`** share many of the same options. Here we define the **`PlotType`** option.

> **`PlotType`** is an option of **`DrawSystem`** and **`ShowSystem`** designating the display form of the graphics rendering.

`PlotType` takes **`TopView`**, **`FrontView`**, **`SideView`**, **`Full3D`**, **`Off`**, **`Surface`**, and **`ShadowProject`** as word values. Here we use **`PlotType->TopView`** with **`ShowSystem`**.

```
ShowSystem[%,PlotType->TopView];
```



The **`%`** symbol feeds the output from the previous result into the expression input. Here the output from **`DrawSystem`** of the last example has been used as input to **`ShowSystem`** in this example.

6 Using DrawSystem for Ray Tracing

You can do ray tracing in *LensLab* using either **DrawSystem** or **PropagateSystem**. **PropagateSystem** is called internally by **DrawSystem**. Here we define **PropagateSystem**.

PropagateSystem[*objectset*, *options***]** takes *objectset* made up of a mixed list of **Ray**, **Component**, and **OpticalSystem** objects, traces the rays through the components, and returns the ray-tracing result as an **OpticalSystem** object carrying a divided listing of **Ray** and **Component** objects.

PropagateSystem works in the following way. **PropagateSystem** takes the input *objectset* and separates the **Ray** objects from the **Component** objects, storing each **Ray** object's original position index as a parameter within the **Ray** object's shell (since the relative ordering of the **Ray** objects within the mixed **Ray**/**Component** object set determines the propagation starting position of each ray within the component list.) Each time a ray intersects a component surface, a new **Ray** object is created containing new **Ray** parameter values that are either updated from old parameters or introduced as new parameters, according to the component's ray-tracing functions. If only **Ray** objects, **Component** objects, or a single **OpticalSystem** object exists at the input, then **PropagateSystem** returns them without change. For each object headed by **OpticalSystem** present in the object set input (indicating a previous ray-trace result), **PropagateSystem** separates and propagates only the last generation of **Ray** objects present, using all **Component** objects present.

Since **PropagateSystem** performs ray tracing without rendering the results, you can use **PropagateSystem** when you are interested only in quantitative results without the pretty pictures. However, since **DrawSystem** performs both the ray tracing and rendering, we use **DrawSystem** for most of the examples in this manual.

■ Tracing a Single Ray

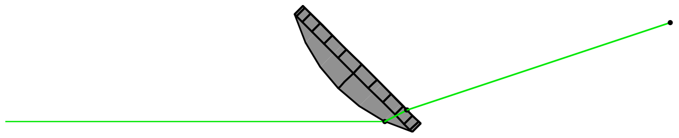
Next we trace a single ray through a lens using **Ray** and **PlanoConvexLens** with **DrawSystem**. First we need to set up a boundary to contain the rays. For this we use **Boundary**.

`Boundary[`*boundaryparameters***`]`** denotes a rectangular box that absorbs rays intercepted by its walls.

There are three methods for specifying *boundaryparameters*: **`Boundary[{`***x1*, *y1*, *z1***`}`**, **`{`***x2*, *y2*, *z2***`}]`** uses the coordinates of top and bottom opposite corners of a rectangular box, **`Boundary[`***side***`]`** assumes a cube boundary, and **`Boundary[`***aside*, *bside***`]`** assumes a three-dimensional box having a length specified by *aside*, a width specified by *bside*, and a height specified by *bside*. Optical systems propagating rays usually have at least one boundary component listed at the end. **`Boundary`** is not rendered. In this example we use **`Boundary[200]`**.

Finally we use **`DrawSystem`** for ray tracing. This time we leave off the semicolon from **`DrawSystem`**, allowing the text output to be printed.

```
DrawSystem[{
    Ray[],
    Move[PlanoConvexLens[100,50,10],{100,10},45],
    Boundary[200]},PlotType->TopView]
```



```
OpticalSystem[{Ray, Ray, Ray},
 {PlanoConvexLens, Boundary}, -OpticsGraphics-]
```

Three **`Ray`** objects are returned by **`DrawSystem`**, each created at an optical surface and holding a record about a particular surface intersection with the ray.

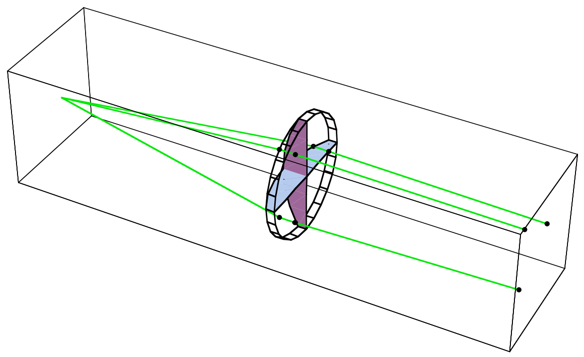
■ Tracing a Cone of Rays

To propagate several rays through a lens, you can use a ray source function. Here we define **`ConeOfRays`**. More ray source functions are discussed in Section 2.3.

`ConeOfRays[`*conicangle*, *options***`]`** creates a set of rays, starting at the origin, equally distributed on the surface of a cone placed symmetrically about the positive *x* axis.

Next we use **ConeOfRays** in **DrawSystem** for ray tracing.

```
DrawSystem[{ConeOfRays[20],
    Move[PlanoConvexLens[100,50,10],100],
    Boundary[{-100,-100,-100},{200,100,100}]}];
```

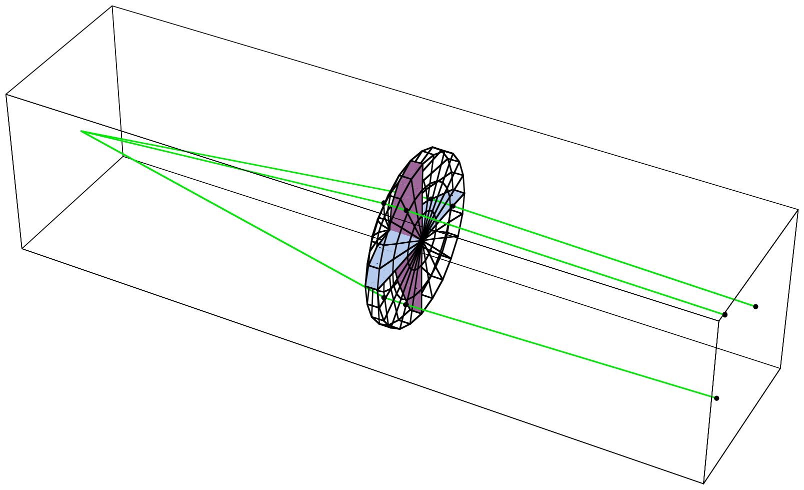


You can change the way components are rendered with **GraphicDesign**. First, we define **GraphicDesign**.

> **GraphicDesign** is an option of all rendered components designating the style of rendering.

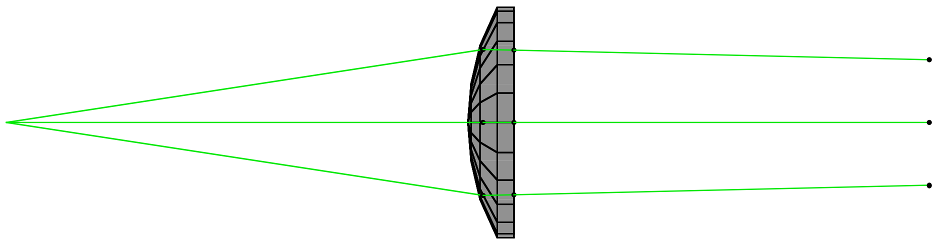
GraphicDesign can be set to **Automatic**, **Sketch**, **Wire**, **Solid**, or **Off**. Here we use **GraphicDesign->Wire** in **PlanoConvexLens**.

```
DrawSystem[{ConeOfRays[20],
    Move[PlanoConvexLens[100,50,10,GraphicDesign->Wire],
    {100,0,0}],
    Boundary[{-100,-100,-100},{200,100,100}]}];
```



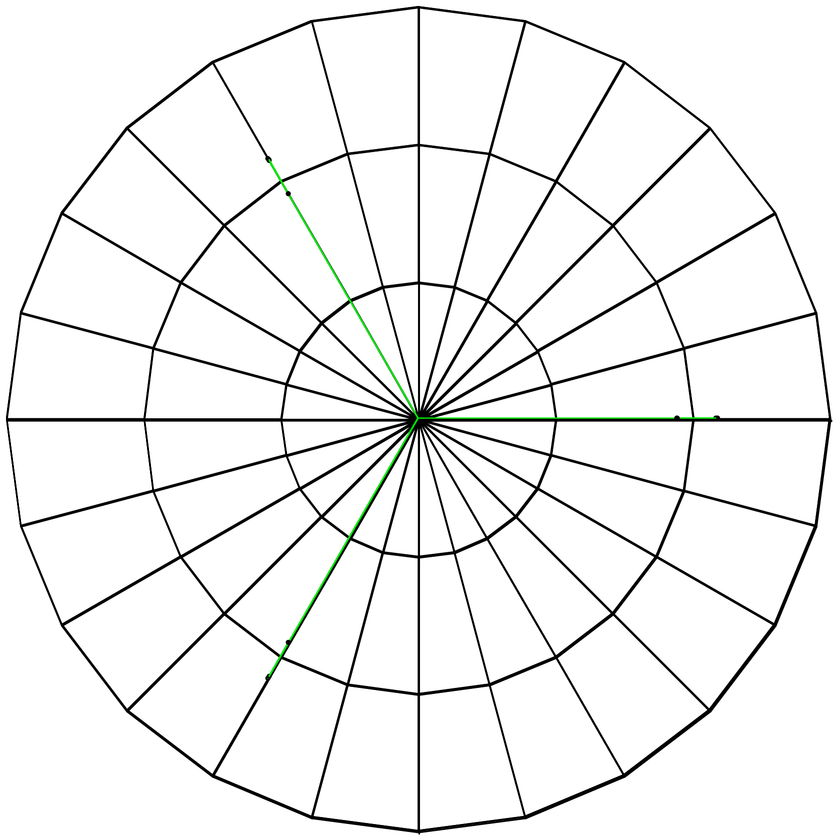
Now we use **`PlotType->SideView`** with **`ShowSystem`** to look at the rendered result from the side.

```
ShowSystem[%,PlotType->SideView];
```



Finally we view the result from the front.

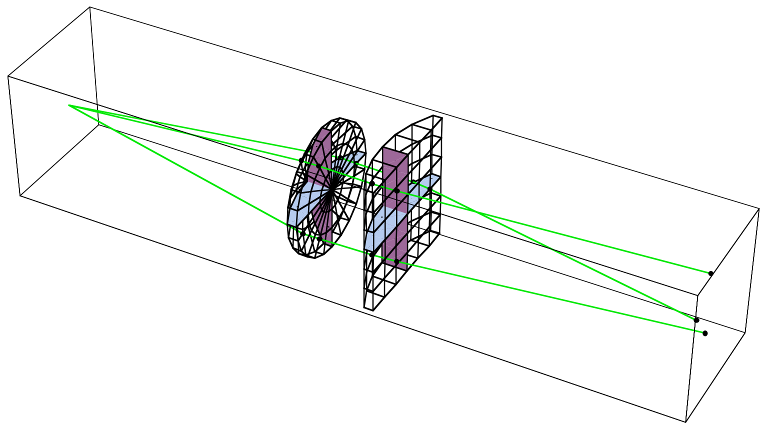
```
ShowSystem[%,PlotType->FrontView];
```



7 Adding a Cylindrical Lens to the System

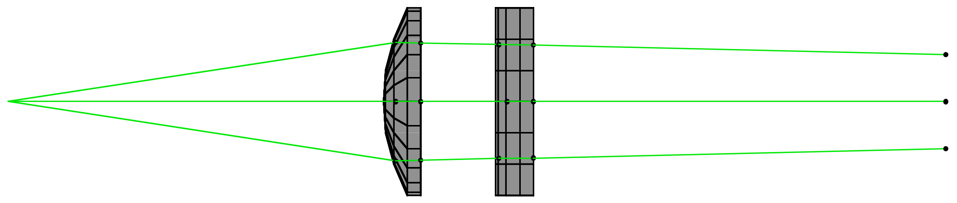
For some additional interest, we place a cylindrical lens directly behind the planoconvex lens. The cylindrical lens component is created with **`PlanoConvexCylindricalLens`**. Here we use **`{50,50}`** in the *aperture* parameter of **`PlanoConvexCylindricalLens`** to make a rectangular-edged cylindrical lens. The shapes of component apertures are discussed further in Section 6.8.

```
opticalsystem = DrawSystem[{
    ConeOfRays[20],
    Move[PlanoConvexLens[100,50,10,GraphicDesign->Wire], 100],
    Move[PlanoConvexCylindricalLens[100,{50,50},10,
GraphicDesign->Wire],130],
    Boundary[{-100,-100,-100},{250,200,200}]}];
```

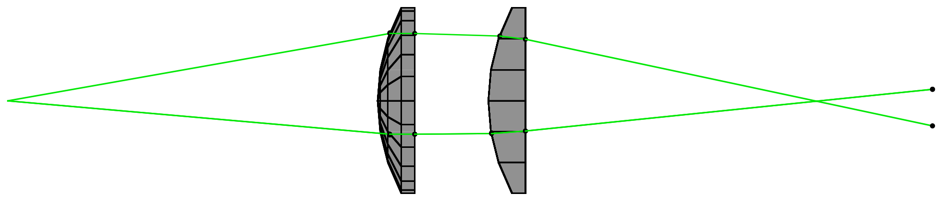


Again, we see different views of the system.

```
ShowSystem[opticalsystem,PlotType->SideView];
```



```
ShowSystem[opticalsystem,PlotType->TopView];
```



You can make exact positional measurements in the two-dimensional fields by clicking on the graphics display cell with the mouse and then holding down the Command key while moving the cursor over the image. The cursor coordinates are displayed in the bottom corner of the window. More accurate measurements can be taken by expanding the graphic display size. By examining the **TopView** image, we measure the focus position to be $x = 218$ millimeters.

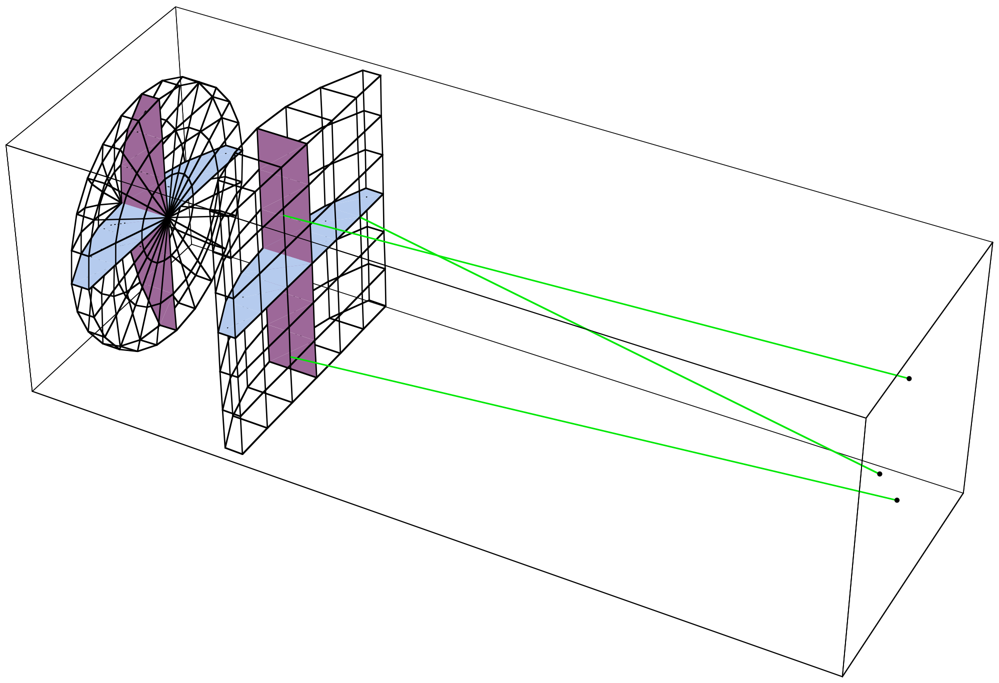
8 Using RayChoice

You can use **RayChoice** to display portions of the ray-tracing result. Here we define **RayChoice**.

RayChoice -> *selectionproperties* uses *selectionproperties* to selectively display ray segments in **DrawSystem** and **ShowSystem**.

Now we use **RayChoice** to examine the ray segments after the cylindrical lens in the system. We use **RayChoice->{ComponentNumber->3}** to view the ray segments immediately following the cylindrical lens. Here we use **ComponentNumber** to choose ray segments associated with a particular component. You can learn more about **ComponentNumber** and other ray pointers in Section 9.3.

```
ShowSystem[opticalsystem, RayChoice->{ComponentNumber->3}];
```

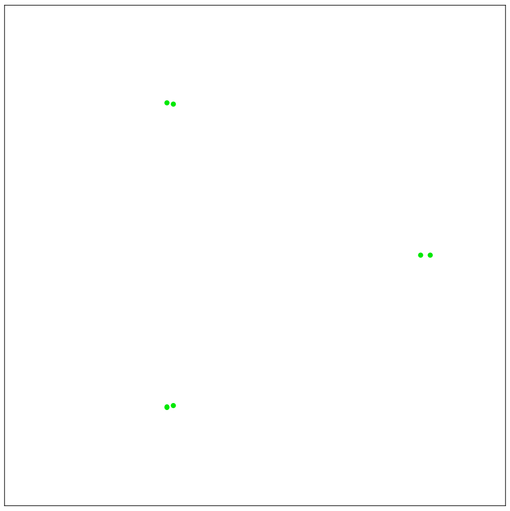


You can also use **`PlotType->Surface`** with **`RayChoice`** to look at the ray intersection points on any surface in the ray-tracing result. We now define **`Surface`**.

`Surface` is a value of **`PlotType`** giving a two-dimensional plot showing selected intersection points between rays and one or more optical surfaces.

Next we use **`RayChoice->{ComponentNumber->2}`** with **`PlotType->Surface`** to view the intersection points on the two surfaces of the cylindrical lens.

```
ShowSystem[opticalsystem, PlotType->Surface,
RayChoice->{ComponentNumber->2}];
```



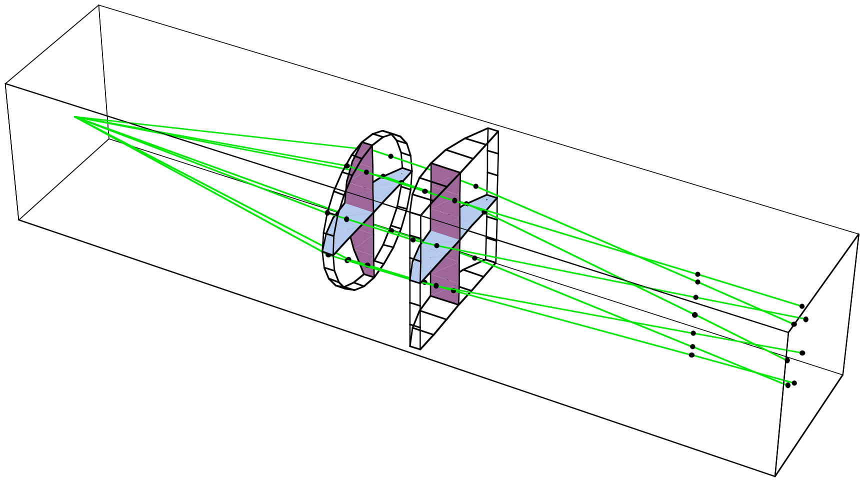
9 Using Screen to Look at the Focal Plane

`Screen` defines a component that samples rays at a given plane in space. You can use **`Screen`** to look at the focal plane of a system. More advanced screens can have curved surfaces and circular or polygonal boundaries. Here is the definition of **`Screen`**.

`Screen[`*aperture*, *options***`]`** denotes a planar component that intersects rays without disturbing them.

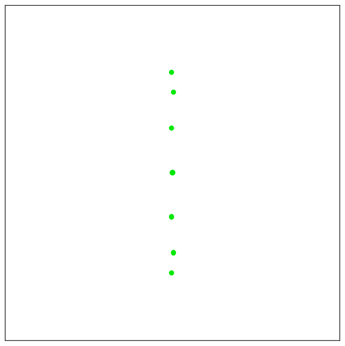
Next we use **Screen** in **DrawSystem** to look at the focal plane of the previous system. We use **Move** to place the screen at the focal position $x = 218$ as measured in Section 1.7.

```
screensystem = DrawSystem[{
    ConeOfRays[20,NumberOfRays->7],
    Move[PlanoConvexLens[100,50,10 ],100],
    Move[PlanoConvexCylindricalLens[100,{50,50},10],130],
    Move[Screen[{50,50}],218],
    Boundary[{-100,-100,-100},{250,200,200}]}];
```



Note that this graphical output shows little indication of a screen being placed in the system other than some additional colored points at the ray/screen intersection. However, by using **PlotType->Surface** with **RayChoice->{ComponentNumber->3}**, we clearly see ray intersection points at the screen surface.

```
ShowSystem[screensystem,PlotType->Surface,
RayChoice->{ComponentNumber->3}];
```



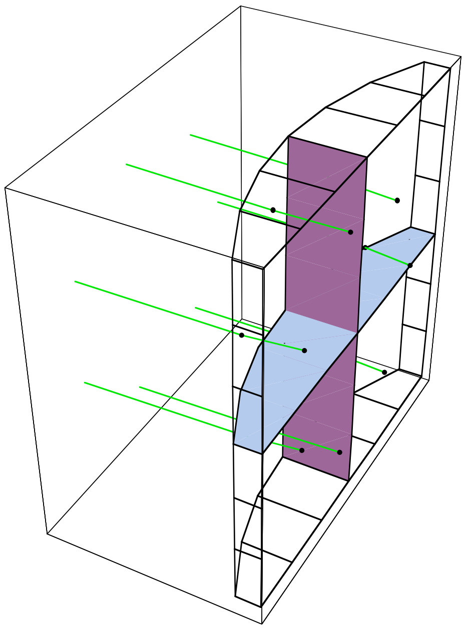
10 The ShowRange Option

Another useful option is **`ShowRange`**. You can use **`ShowRange`** to zoom in on a particular portion of the system.

> **`ShowRange ->`** *values* uses **`ComponentNumber`** values to select the components and ray segments displayed by **`DrawSystem`** and **`ShowSystem`**.

`ShowRange` can take either the value **`All`** or a list of the **`ComponentNumber`** values. Here we examine the ray segments connected with the cylindrical lens.

```
ShowSystem[screensystem,ShowRange->{2}];
```



`ShowRange` works differently from **`RayChoice`** because **`ShowRange`** displays selected components associated with the selected ray segments. In addition, **`ShowRange`** only works with **`ComponentNumber`** values whereas **`RayChoice`** works with many different selection parameters. See Section 6.3 for the other rendering options of **`DrawSystem`**.

11 The ReadRays Function

It is often desirable to get specific parameter values for a selected group of intersection points at an optical surface. This can be accomplished by using **`ReadRays`**.

> **`ReadRays[`***objectset*, *rayparameters*, *selectionproperties*, *options***`]`** is an advanced function that takes *objectset* containing **`Ray`** objects and returns a list of values for *rayparameters* given.

Here we use **`ReadRays`** to examine the optical path-length of rays at the focal plane. Using the optical path-length parameter **`OpticalLength`**, we use the **`ComponentNumber->3`** as a selection property to examine the optical path-length from the ray starting point to the surface of the screen component.

```
ReadRays[screensystem,OpticalLength,{ComponentNumber->3}]
```

```
{228.706, 228.552, 228.49, 228.652, 228.652, 228.49, 228.552}
```

To see more decimal places, we use **`InputForm`**.

```
InputForm[%]
```

```
{228.7056856186972, 228.5522736042738, 228.49049069994487,
228.65214729175403, 228.65214729175403, 228.49049069994487,
228.5522736042738}
```

12 Energy Calculations with `FindIntensity`

LensLab can help you determine the light intensity profile at specified optical surfaces and to measure the transmitted energy through the optical system. This is accomplished with the **`FindIntensity`** function.

> **`FindIntensity[`***system*, *options***`]`** calculates the intensity function for each optical surface that gets reported from the ray trace of the *system*.

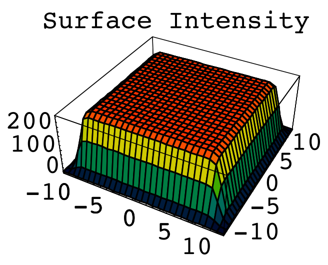
For its input, **FindIntensity** works best with either an untraced "raw" optical system or a previously calculated result by **FindIntensity**. However, when required, **FindIntensity** can also work with externally generated trace results. **FindIntensity** works equally well for surfaces that either are close to a focal plane or far from any focus. If a light sheet source is used (ie. **WedgeOfRays** or **LineOfRays**), then a one-dimensional intensity function is automatically calculated by **FindIntensity**. If a volume-filling source is used (ie. **PointOfRays** or **GridOfRays**), then the intensity calculations are automatically carried out for each reported surface in two-dimensions. Because **FindIntensity** does not calculate interference or keep track of the coherent phase information, it can only measure incoherent light flux properties.

■ 2-D Calculations with FindIntensity

As an example, we will plot the intensity present for a planar cross-section of a Gaussian beam. We will use **GridOfRays** function to generate a three-dimensional pattern of rays. **FindIntensity** uses the **Plot2D** option to specify how the intensity information is rendered. With **Plot2D -> False**, the intensity is rendered as a three-dimensional plot. (**Plot2D -> True** constructs a two-dimensional plot.) In addition, the plot is colored according to the intensity levels present.

```
intensityresult =
    FindIntensity[{
    GridOfRays[{20,20}, NumberOfRays->32],
    Move[Screen[50],50]}, Plot2D -> False]
```

```
Surface Information : {ComponentNumber → 1, SurfaceNumber → 1,
  NumberOfRays → 1024, SmoothKernelSize → 0.912396}
```



```
{ComponentNumber → 1, Energy → 102400., Full3D → True,
 IntensityFunction → CompiledFunction[-intensity data-],
 NumberOfRays → 1024, OutputGraphics → (- SurfaceGraphics -),
 RayBoundary → {{-10., 10.}, {-10., 10.}},
 SmoothKernelSize → 0.912396, SurfaceNumber → 1}
```

Note that since we specified **`NumberOfRays -> 32`** there have been, in fact, 32^2 = 2048 rays created. In general, you are free to choose the number of rays to be traced with **`FindIntensity`**. Bear in mind, however, that there needs to be a sufficient number of ray samples to make the **`FindIntensity`** result meaningful. As a rule of thumb, **`NumberOfRays -> 32`** is the minimum value required for reasonable full-surface calculations (to produce 4096 rays) and **`NumberOfRays -> 256`** is the minimum required for light-sheet ray-traces (to produce 256 rays).

You can use **`Options[FindIntensity]`** to observe its default option settings.

```
Options[FindIntensity]
```

```
{KernelScale → Relative, SmoothKernelSize → 1, SmoothKernelRange → 4,
 IntensitySetting → Automatic, Energy → Automatic, IntensityScale → 1,
 Print → True, Show → True, ReportedSurfaces → Last, GenerationLimit → 200,
 PlotRange → All, PlotDomain → Automatic, PlotPoints → 30,
 Plot2D → ContourPlot, ContourLines → False, Contours → 50,
 Full3D → Automatic, ColorFunction → (Hue[0.65 - #1 0.65, 1, #1 0.9 + 0.1] &),
 ThresholdIntensity → 0.001, CosineCompensation → True,
 SequentialRead → Automatic, InterpolatingFunction → False,
 InterpolationOrder → 1, SampleFactor → 2,
 RecenterData → False, RunningCommentary → False}
```

Here we can see that **`FindIntensity`** uses a number of options that we have not encountered before. The most significant of these are listed below.

Out[12]//TableForm=

IntensitySetting	Print
SequentialRead	ReportedSurfaces
IntensityScale	Show
KernelScale	SmoothKernelRange
Plot2D	SmoothKernelSize

Options that help characterize **`FindIntensity`**.

While some of these options will be considered further in this section, others will be left to discussion elsewhere.

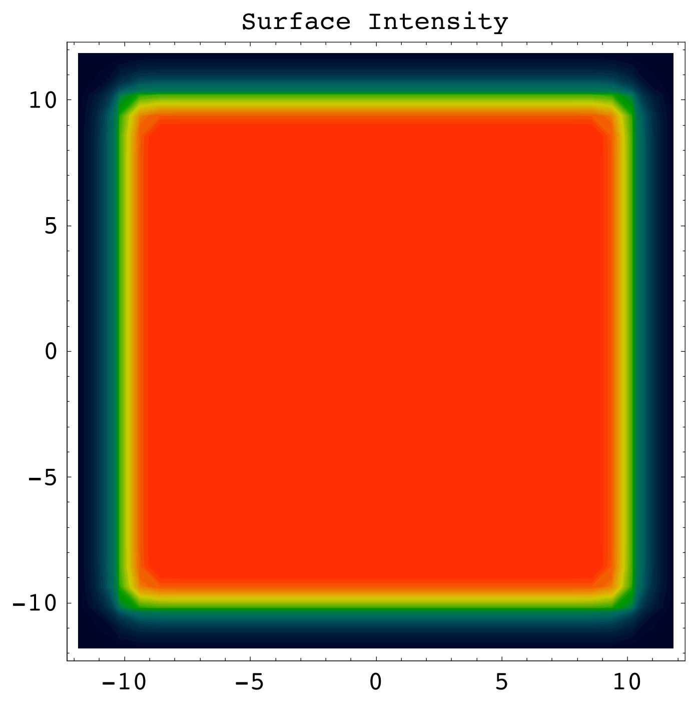
In addition to showing information as a three-dimensional plot, the **`Plot2D`** option in **`FindIntensity`** has several other settings. Its settings are shown below.

`False` or **`Full3D`** gives the full three-dimensional rendering of the intensity profile.
`True` or **`SideView`** gives a two-dimensional cross-section rendering at the x-z plane of the intensity profile.
`FrontView` gives a two-dimensional cross-section rendering at the y-z plane of the intensity profile.
`ContourPlot` gives a contour plot of the intensity profile.

Settings of **`Plot2D`**.

Next we shall plot the same **`FindIntensity`** calculation with its default plot setting (**`Plot2D -> ContourPlot`**). We will use **`Print -> False`** to switch off the printed messages. (**`Show -> False`** can be used to switch off the graphical rendering.)

```
FindIntensity[intensityresult, Print -> False]
```

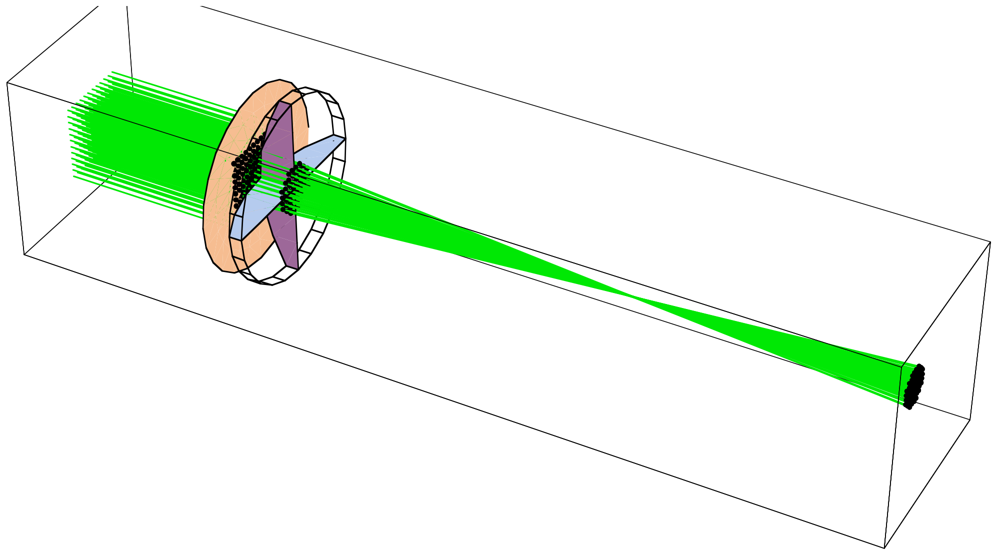


```
{ComponentNumber → 1, Energy → 102400., Full3D → True,
 IntensityFunction → CompiledFunction[-intensity data-],
 NumberOfRays → 1024, OutputGraphics → (- ContourGraphics -),
 RayBoundary → {{-10., 10.}, {-10., 10.}},
 SmoothKernelSize → 0.912396, SurfaceNumber → 1}
```

■ FindFocus

In the previous example, we examined the intensity at a single surface. However, if multiple surfaces are reported, **FindIntensity** can measure the energy losses in the system that occurs between the first reported surface and every subsequent reported surface. We will see this in the next example. Here, we use an **PinHole** as a pupil that restricts the transmitted light through a lens. In addition, we will use **FindIntensity** to calculate the point spread function of the optical system by placing one of the reported surfaces at the lens focal point. Before running **FindIntensity**, we will first use **DrawSystem** to show a ray-trace of our intended system.

```
trace = DrawSystem[{GridOfRays[{20,20}, NumberOfRays->12],
    Move[PinHole[50,15],49],
    Move[PlanoConvexLens[100,50,10],50],
    Move[Screen[50],225]}, PlotType->Full3D];
```

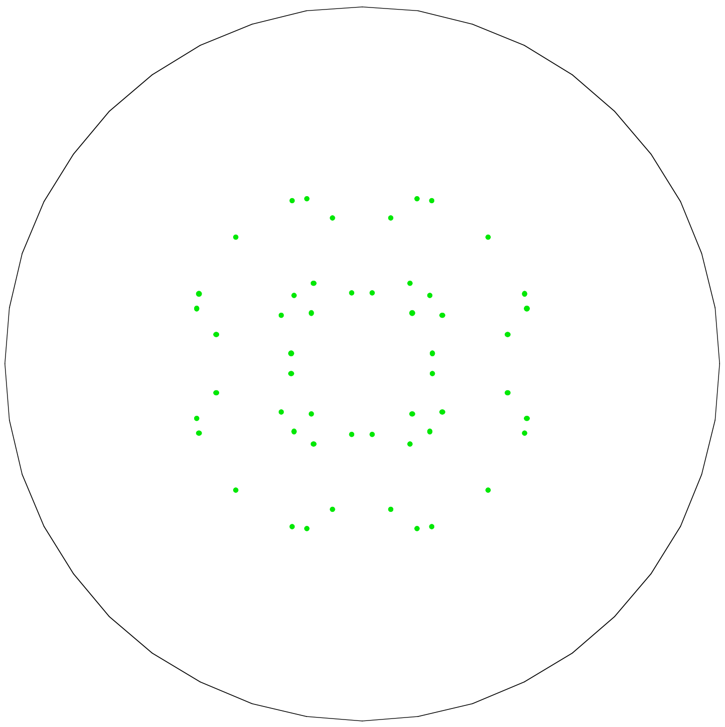


In order to measure the point spread function of the lens system, you first need to determine the focal point of the system. You can use the **FindFocus** function for this. Here is a description of **FindFocus**.

FindFocus[*objectset*, *selectionproperties*, *options***]** determines the minimum spot size for a locus of rays specified in selectionproperties and plots the results.

Next, we can use the **FindFocus** function to find focal point of our lens system. In this case, since we have already calculated a ray trace using **DrawSystem**, we can simply pass to **FindFocus** the result of our previous calculation.

```
focus = FindFocus[trace,ComponentNumber->3]
```



```
{FocalPoint → {153.135, 0., 0.}, SpotSize → 0.0177769,
 FocalLength → 93.1427, FocalPlaneTilt → {1., 0., 0.}}
```

FindFocus has automatically generated a plot of the locus of rays at the focal plane. In addition, **FindFocus** returns a series of rules that describe various aspects of its focus calculation. Of the different rules returned by **FindFocus**, we are only presently interested in the **FocalPoint** rule since it holds the three-dimensional focal-point coordinates. As such, we can assign the focal point result to a **focalpoint** variable in the following way:

```
focalpoint = FocalPoint/.focus
```

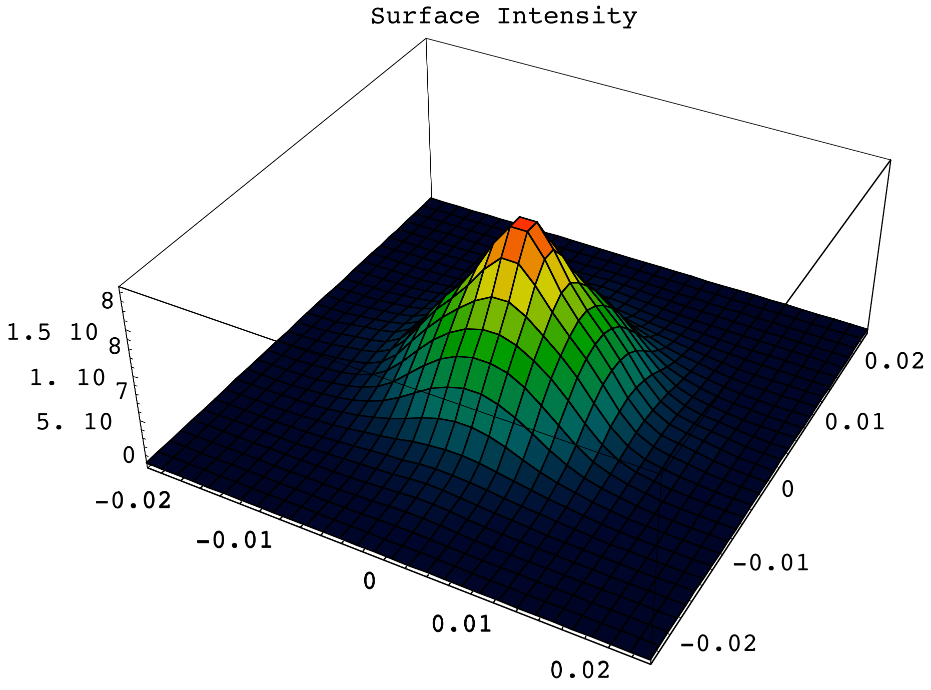
```
{153.135, 0., 0.}
```

■ Measuring the Point Spread Function

We are now ready to apply **FindIntensity** to our lens system to measure the point spread function of the system. This time, we will place a **Screen** object at the focalpoint determined by **FindFocus**.

```
focusintensity =
FindIntensity[{GridOfRays[{20,20}, NumberOfRays->32],
    Move[PinHole[50,15],49],
    Move[PlanoConvexLens[100,50,10],50],
    Move[Screen[{.6,.6}],focalpoint]}, Plot2D -> False]
```

```
Surface Information : {ComponentNumber → 3, SurfaceNumber → 1,
  NumberOfRays → 432, SmoothKernelSize → 0.00376605}
```

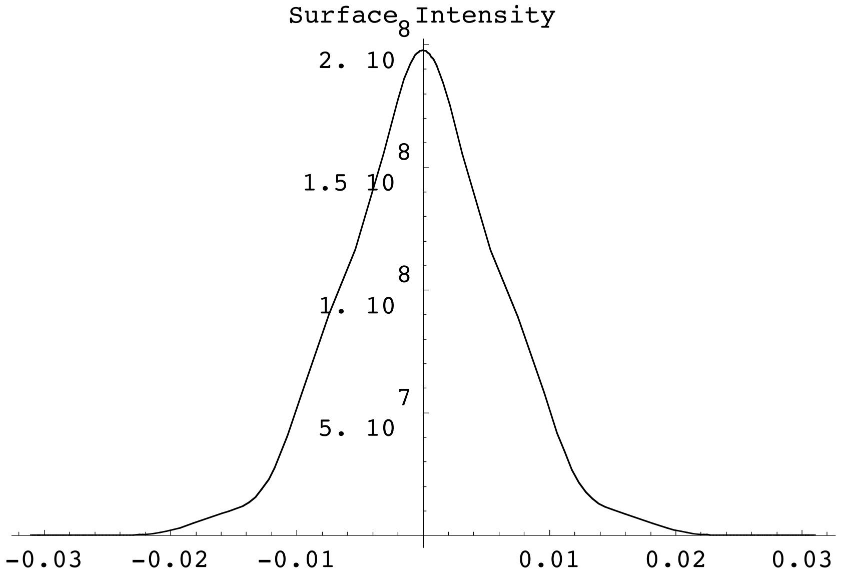


```
{ComponentNumber → 3, Energy → 43200., Full3D → True,
 IntensityFunction → CompiledFunction[-intensity data-],
 NumberOfRays → 432, OutputGraphics → (- SurfaceGraphics -),
 RayBoundary → {{-0.01599, 0.01599}, {-0.01599, 0.01599}},
 SmoothKernelSize → 0.00376605, SurfaceNumber → 1}
```

Previously, we had used **`Plot2D -> False`** to display a three-dimensional plot of the surface intensity profile. Next we will call **`FindIntensity`** for a second time with **`Plot2D -> True`** to examine the intensity function along the horizontal-axis of each surface.

```
FindIntensity[focusintensity, Plot2D -> True];
```

```
Surface Information : {ComponentNumber → 3, SurfaceNumber → 1,
  NumberOfRays → 432, SmoothKernelSize → 0.00376605}
```



The last intensity plot is taken at the focal plane of the system and shows the point spread function. **`FindIntensity`** works by convolving a Gaussian smoothing kernel with the ray-trace data. However, the result may not accurately depict the diffractive performance of the system. In particular, **`FindIntensity`** automatically adjusts the smooth kernel size to maximize the measurement resolution to the available ray-density. In this way, if the number of input rays are sufficiently great, **`FindIntensity`** models the geometric behavior of the system. However, you can approximate some effects of the diffraction performance if you have independent knowledge of the diffraction-limited spot-size for the system. In this case, you can manually set the **`SmoothKernelSize`** option to correspond with the diffraction-limited spot-size of the optical system (together with **`KernelScale -> Absolute`**). In that case, the intensity plot generated by **`FindIntensity`** can depict both the geometric and incoherent diffractive properties of a system. However, if the **`SmoothKernelSize`** is manually set, it is always necessary that make sure that there are a sufficient number of rays are present in order to meet the Nyquist sampling criterion. Otherwise, the resulting calculation will not be correct. Here is the definition of **`SmoothKernelSize`**.

`SmoothKernelSize ->` *radius* is an option that specifies the *radius* of a Gaussian smoothing kernel: **`Exp[-`***s*^2/*radius*^2**`]`**, where *s* is distance along the surface. In particular, this smoothing kernel is convolved with another function in order to low-pass filter its shape along a spatial surface.

`SmoothKernelSize` works together with the **`KernelScale`** option to smooth the calculated intensity function. Here is the definition of **`KernelScale`**.

`KernelScale -> Relative` / **`Absolute`** is used with **`FindIntensity`** to specify the form of the **`SmoothKernelSize`** option.

`KernelScale -> Relative` indicates that the specified **`SmoothKernelSize`** dimensions is a relative multiplicative factor of the minimum estimated spatial sampling dimension, as determined from Nyquist sampling criteria. **`KernelScale -> Absolute`** denotes that the specified **`SmoothKernelSize`** dimensions are given in absolute spatial dimensions.

■ Measuring the Modulation Transfer Function

Now that we have used **`FindIntensity`** to determine the point spread function of this lens system, we can also calculate the **`ModulationTransferFunction`** of this system. First we define **`ModulationTransferFunction`**.

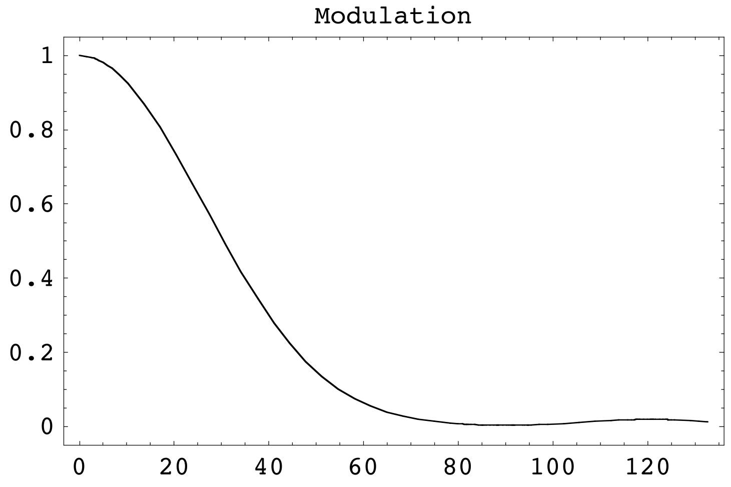
`ModulationTransferFunction[`*intensitydata*, *options***`]`** calculates the modulation and phase transfer functions of an optical system for a given object source input.

`ModulationTransferFunction` works together with **`FindIntensity`**. As input, **`ModulationTransferFunction`** takes the returned output from **`FindIntensity`**. The optical system must contain a light source followed by the imaging optics with the focal surface as its last element.

```
Options[ModulationTransferFunction]
```

```
{SpatialScale → 1, FrequencyCutoff → Automatic,
 PaddingFactor → 7, NormalizePlot → True, InterpolationOrder → 1,
 PlotPoints → 40, Plot2D → True, ContourLines → False, Contours → 50,
 ColorFunction → (Hue[0.65 - #1 0.65, 1, #1 0.9 + 0.1] &),
 RenderedParameters → {ModulationTransferFunction}}
```

```
ModulationTransferFunction[focusintensity]
```

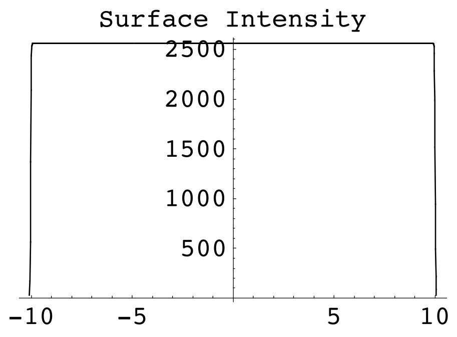


```
{ComponentNumber → 3, Energy → 43200.,
 FrequencyCutoff → {132.765, 132.765}, Full3D → True,
 ImageSampleSize → {0.00355333, 0.00355333},
 IntensityFunction → CompiledFunction[-intensity data-],
 ModulationTransferFunction →
  InterpolatingFunction[{{-140.713, 137.44}, {-140.713, 137.44}}, <>],
 NumberOfPoints → {10, 10}, NumberOfRays → 432, OutputGraphics →
  {- SurfaceGraphics -, ModulationTransferFunction → (- Graphics -)},
 PhaseTransferFunction → InterpolatingFunction[
   {{-140.713, 137.44}, {-140.713, 137.44}}, <>],
 RayBoundary → {{-0.01599, 0.01599}, {-0.01599, 0.01599}},
 SmoothKernelSize → 0.00376605, SurfaceNumber → 1}
```

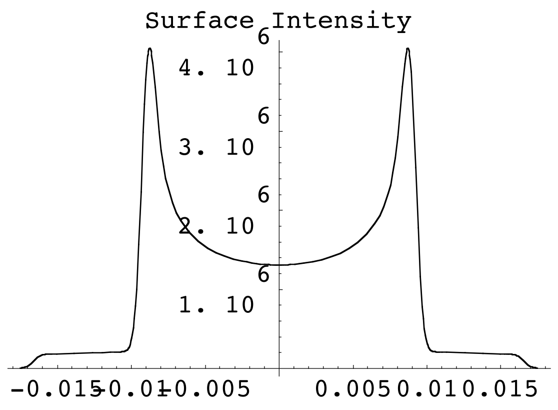
`ModulationTransferFunction` works equally well for both point sources and planar sources, as long as the described imaging system contains a focus. If a one-dimensional light source is used (ie. **`WedgeOfRays`** or **`LineOfRays`**), then a one-dimensional modulation transfer function is calculated. If a two-dimensional light source is used (ie. **`PointOfRays`** or **`GridOfRays`**), then the optical transfer function calculations are carried out in two-dimensions.

```
intensity = FindIntensity[{LineOfRays[20, NumberOfRays->512],
    Move[PinHole[50,15],49],
    Move[PlanoConvexLens[100,50,10],50],
    Move[Screen[{.6,.6}],focalpoint]}, ReportedSurfaces->{{1,1},{3,1}}]
```

```
    Surface Information : {ComponentNumber → 1, SurfaceNumber → 1,
      NumberOfRays → 512, SmoothKernelSize → 0.0391389}
```



```
    Surface Information : {ComponentNumber → 3, SurfaceNumber → 1,
      NumberOfRays → 384, SmoothKernelSize → 0.000588641}
```



```
{{ComponentNumber → 1, Energy → 51200., Full3D → False,
  IntensityFunction → CompiledFunction[-intensity data-],
  NumberOfRays → 512, OutputGraphics → (- Graphics -),
  RayBoundary → {-10., 10.}, SmoothKernelSize → 0.0391389,
  SurfaceNumber → 1}, {ComponentNumber → 3, Energy → 38400.,
  Full3D → False, IntensityFunction → CompiledFunction[-intensity data-],
  NumberOfRays → 384, OutputGraphics → (- Graphics -),
  RayBoundary → {-0.0163078, 0.0163078},
  SmoothKernelSize → 0.000588641, SurfaceNumber → 1}}
```

```
Energy/.intensity
```

```
{51200., 38400.}
```

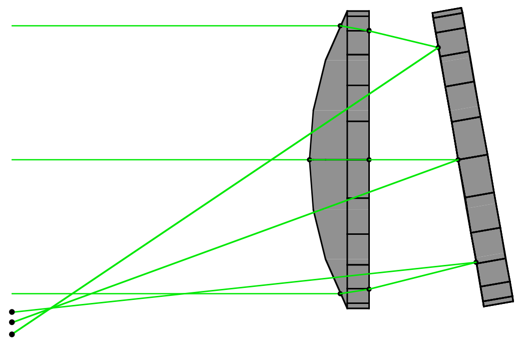
13 The `Resonate` Function

`Resonate` is used to create non-sequential behavior between formerly distinct optical elements. **`Resonate`** is routinely used by many built-in component functions of *LensLab* to describe complex optical elements, such as multi-faceted prisms. Since many optical systems do require non-sequential ray interactions between multiple component surfaces, **`Resonate`** is one of the most important functions in *LensLab*. Here we define **`Resonate`**.

`Resonate[`*listofcomponents*, *options***`]`** causes a ray to be nonsequentially traced within all of the surfaces defined by *listofcomponents*.

LensLab always traces rays between distinct optical components in a sequential fashion. This means that the ray-trace is always occurring in the same sequence as the component list order. However, anytime the ray trace occurs within the surfaces of an optical component (such as a **`Prism`**) then non-sequential ray-tracing becomes possible. In some cases, however, it becomes necessary for rays to travel non-sequentially between one or more components. For this purpose, you can use **`Resonate`**. Lets take, for an example, the case of a lens-mirror combination. Here, the rays might travel through a lens, reflect off a mirror, and then travel back through the same lens for a second pass. In its normal mode of operation, *LensLab* does not see the lens on the second pass when the components are listed separately.

```
DrawSystem[{
    LineOfRays[45],
    Move[PlanoConvexLens[100,50,10],50],
    Move[Mirror[50,5],75,10],
    Boundary[100]},PlotType->TopView];
```



Copyright Statement