Getting Started With Optica

As will be seen, *Optica* is an authoring tool based on the *Mathematica* backbone. This aspect of *Optica* allows systems to be easily designed by engineers and visually demonstrated to non-technical persons to faciliate concept communication.

The guide is separated into two parts. The first section begins with an introduction to very basic *Mathematica* concepts, such as variables, lists, and function formatting. These concepts are then combined to create a *Mathematica* function. The second section focus is on *Optica* functions and optical system design tools.

Mathematica Basics

Cells

A cell in *Mathematica* is visually depicted by a brace located on the right side of a notebook. Cells contain evaluation and data for various functions.

Evaluation

An expression is evaluated after either the user invokes the key sequence **`'Shift + Enter'`** or depresses the **`'Enter'`** key on the numerical keypad.

Variables

Variable names not defined are highlighted in blue and are lowercase. They do not have to be declared as a specific type. As an example for programmers with a background in C/C++, you would not have to declare a variable as type 'Int' or 'Char'.

By default, variables are global. The global aspect means that when values are assigned to variables, that value can be recalled from any evaluatable cells in a notebook. A global variable also represents that a change in a variable value will wipeout any previous value that was stored in the variable.

When a variable is undefined, its evaluation output is its name. So, if we were to evaluate an undefined variable called 'variable1', its output would be 'variable1'.

```
variable1
```

```
variable1
```

Below, we have 'variable1' which is assigned the value 42.

```
variable1 = 42
```

```
42
```

Here, 'variable1' is recalled. The value 42 is displayed, as was assigned in a previous cell.

```
variable1
```

```
42
```

Variables may also be cleared if need be. This is done using the 'Clear' function. Detailed information on Functions and their usage is described later in this guide. For now, simply take note that 'Clear' clears out a variable, and its implementation is below.

```
Clear[variable1]
```

```
variable1
```

```
variable1
```

Variables can be made local in a few ways. One method of temporary assignment is to use 'Rule' represented by the '->' symbol.

The followng expression is of the form '3 + variable2'. If we would like to substitute the number 5 for **`'variable2'`** in this expression, we follow the form 'expr /. variableToBeReplaced -> value'. After evaluation, our expected result is the number 8.

```
3 + variable2 /. variable2 → 5
```

```
8
```

Also, we can store a symbolic expression as a *Mathematica* variable. This higher level expression can be called globally, and have it's internals substituted locally.

```
newExpression = 3 + variable2
```

```
3 + variable2
```

```
newExpression /. variable2 → 5
```

```
8
```

Interesting to note is that expressions within expressions can also be assigned to higher level expressions...

```
newNewExpression = newExpression + newExpression
```

```
6 + 2 variable2
```

...and have variable substitution performed on them in the same manner. If we replace **`'variable2'`** with the number 5, as **`'newNewExpression'`** is equal to **`'6 + 2 * variable2'`** we should receive a result of 16.

```
newNewExpression /. variable2 → 5
```

```
16
```

The fact that variables as part of expressions can be assigned to other variables and expressions relates to the next topic of discussion - lists.

Lists

Another major *Mathematica* concept to master is that items can be grouped into lists and have mathematical operations performed on them. In fact, lists can be grouped into lists, mathematically manipulated, which can also be assigned to yet another list.

```
a = {1, 2, 3}
```

```
{1, 2, 3}
```

```
a = a + 1
```

```
{2, 3, 4}
```

```
b = {a, a}
```

```
{{2, 3, 4}, {2, 3, 4}}
```

```
c = {a, b + 5}
```

```
{{2, 3, 4}, {{7, 8, 9}, {7, 8, 9}}}
```

```
d = {a,  b,  c}
```

```
{{2, 3, 4}, {{2, 3, 4}, {2, 3, 4}}, {{2, 3, 4}, {{7, 8, 9}, {7, 8, 9}}}}
```

Elements in lists can be selected with the '[[ ]]' formatter. Below, the first element in list 'a' is selected for output.

```
a[[2]]
```

```
3
```

```
d[[1]]
```

```
{2, 3, 4}
```

The semicolon ';' suppresses output.

```
a = {1, 2, 3};
```

There are also various *Mathematica* operations that allow for a greater list manipulation control.

The following operation returns the third element in the first sublist.

```
d[[1, 3]]
```

```
4
```

'All' is a built-in *Mathematica* command that can be used to output every element in a given parameter. For example, if we desire the second element of every sublist as output, the formating is a follows: 'listName[[All, 2]]'.

```
exampleList1 = {{1, 2, 3}, {1, 2, 3}, {1, 2, 3}, {1, 2, 3}}
```

```
{{1, 2, 3}, {1, 2, 3}, {1, 2, 3}, {1, 2, 3}}
```

```
exampleList1[[All, 2]]
```

```
{2, 2, 2, 2}
```

```
d[[All]] + d[[All]]
```

```
{{4, 6, 8}, {{4, 6, 8}, {4, 6, 8}}, {{4, 6, 8}, {{14, 16, 18}, {14, 16, 18}}}}
```

You should be able to apply this fundamental concept to basic functions in *Mathematica*.

Functions

The first key concept to grasp is that the first letter of each word in all *Mathematica* and *Optica* functions are Capitalized. Also, each function has square brackets **'[ ]'** following its name. Take the *Optica* function **'FindIntensity'** for example. It follows the format : **'FindIntensity[*paramter1, parameter2*]'**. Note that the words composing the function, **'Find'** and **'Intensity'** are Capitalized. Keep in mind that there are no abbreviated function names in *Mathematica* and *Optica*. There is no such thing as a *Mathematica* or *Optica* built function in the style of **'fnctnname[ ]'**.

Function Declaration

```
ExampleFunction[x_, y_] := x^3 + y^2
```

Here, we will use our newly created function. The numbers '2' and '3' are input. The number '2' will be cubed and added to the square of '3'. Our result should be 17.

```
ExampleFunction[2, 3]
```

```
17
```

Function Usage

If one has a question about a Function in *Optica* or *Mathematica*, they may receive help by simply entering the command following the form **'?Function'**.

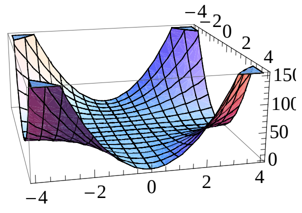
```
? Plot3D
```

Plot3D[f, {x, x_{min}, x_{max}}, {y, y_{min}, y_{max}}] generates a three-dimensional plot of f as a function of x and y.

Plot3D[{f_1, f_2, …}, {x, x_{min}, x_{max}}, {y, y_{min}, y_{max}}] plots several functions. ≫

Inferring from the **`'Plot3D'`** usage message, we can graph the equation **'x ^{2} × y ^{2}'**, with x bounded from -4 to 4, and y bounded from -4 to 4 in the following manner :

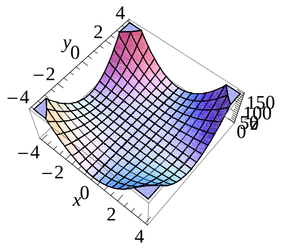
```
Plot3D[x^2 y^2, {x, −4, 4}, {y, −4, 4}]
```



Function Options

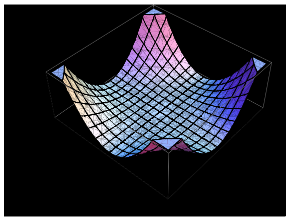
Functions may also have an option. In the example below, the x, y, and z axes will be labeled their respective titles.

```
Plot3D[x^2 y^2, {x, −4, 4}, {y, −4, 4}, AxesLabel → {x, y, z}]
```



Multiple options may be used in a single function.

```
Plot3D[x^2 y^2, {x, −4, 4}, {y, −4, 4}, AxesLabel → {x, y, z}, Background → Black]
```



Options of a specific function may be found by invoking the **`'Options[Function]'`** command.

Functions and Lists

To fully understand the versatility of *Mathematica*, we must see that lists can be passed as parameters to functions, returned from functions, and functions themselves can be stored into lists. Results from these lists and the lists themselves can be assigned and substituted to both global and local variables.

We redeclare our **'ExampleFunction'** for completeness.

```
ExampleFunction[x_, y_] := x^3 + y^2
```

Let us pass parameter **'x'** the numerical values 2 and 3 in list format, and the **'y'** parameter the number 3.

```
ExampleFunction[{2, 3}, 3]
```

```
{17, 36}
```

Our result is a list containing the values 17 and 36. Why was the result presented in this manner? The answer to this question is that the equation **'x^3 + y^2'** present in **'ExampleFunction'** was first evaluated with 'x' having the value '2', and 'y' as '3'. This evaluation yields '17' as our result. Next, *Mathematica* evaluates **'x^3 + y^2'**, substituting **'3'** for **'x'** and **'3'** for **'y'**. Our result is then **'36'**. Due to the internals and theory of operation of *Mathematica*, the single object returned was a list.

We can make a real world analogy to this process. The function call is analogous to having an order sent off to a factory. In our order, we request multiple items to be processed, some of which must undergo the same processing procedure. The request to process multiple items in our order request in the same fashion is comparable to passing multiple items in a list to the same parameter. When our order is complete, we receive a single object, one shipping crate. Although the crate is a single object, we have our multiple objects inside of it. This crate is similar to how *Mathematica* treats our resulting list - as an object containing multiple items.

Our previous result can also be stored inside of a variable and recalled.

```
exampleList2 = ExampleFunction[{2, 3}, 3]
```

```
ExampleFunction[{2, 3}, 3]
```

```
exampleList2
```

```
ExampleFunction[{2, 3}, 3]
```

To demonstrate list versatility, we will have a variable that is assigned functions formatted in a list, whose parameters are passed in list form.

```
sampleList = {ExampleFunction[{2, 3, 4}, {1, 2, 3}],
  ExampleFunction[{1, 2, 3}, {3, 5, 6}]}
```

```
{{9, 31, 73}, {10, 33, 63}}
```

To expand upon list versatility, we will have a variable that is assigned functions formatted in a list, whose parameters are passed in list form by selecting lists that are a subset of a larger list.

```
newSampleList = {ExampleFunction[sampleList[[1]], sampleList[[2]]],
  ExampleFunction[sampleList[[2]], sampleList[[1]]]}
```

```
{{829, 30 880, 392 986}, {1081, 36 898, 255 376}}
```

Optica Overview

Optica functions follow the same basic *Mathematica* format. Each optical element in *Optica* is a *Mathematica* function. Also worthy of mention, paramters aside from angles in *Optica* are unitless, that is the user specifies units used. Note that angles are in degrees.

First, load optica with the **'Needs'** command.

```
Needs["Optica`Optica`"]
```

```
++++++++++++++++++++++++++++++++
```

```
Optica 3.0 was loaded in 19 s and needs 15128 kilobytes of memory on top of 15599 kilobytes already used
```

Component Functions

Optical elements in *Optica* are considered components, and are implemented as functions. The 'ComponentFunctions' command lists all functions used to create optical elements.

ComponentFunctions

ABCDOptic	CustomDiffuserMirror	JonesMatrixOptic	RodLens
AnamorphicPrisms	CustomFiber	LensDoublet	RodMirror
ApertureStop	CustomFiberMirror	LensSurface	RoofPrism
AsphericLens	CustomGrating	LensTriplet	SchmidtLens
AsphericLensSurface	CustomGratingMirror	LinearPolarizer	SchmidtLensSurface
AsphericMirror	CustomLens	Mirror	Screen
Baffle	CustomLensSurface	MirrorSpan	SlabPrism
BaffleSpan	CustomMirror	ParabolicDiffuser	SnowConeLens
BaffleWithHole	CustomPrism	ParabolicDiffuserMirror	SolidCornerCube
BallBaffle	CustomScreen	ParabolicLensSurface	Solitaire
BallLens	CylinderGraphic	ParabolicMirror	SphereGraphic
BallMirror	CylindricalBaffle	PechanPrism	SphericalBaffle
BeamSplitter	CylindricalLens	PellinBrocaPrism	SphericalBeamSplitter
BeamSplitterCube	CylindricalLensSurface	PentaPrism	SphericalDiffuser
BiConcaveCylindricalLens	CylindricalMirror	PinHole	SphericalDiffuserMirror
BiConcaveLens	CylindricalScreen	Pipe	SphericalGratingMirror
BiConvexCylindricalLens	Diffuser	PlanoConcaveCylindricalLens	SphericalLens
BiConvexLens	DiffuserMirror	PlanoConcaveLens	SphericalLensSurface
BirefringentLensSurface	DirectVisionPrism	PlanoConvexCylindricalLens	SphericalMirror
Boundary	DovePrism	PlanoConvexLens	SphericalScreen
Box	DTIRCLens	PolarizingBeamSplitterCube	SquareConeMirror
BoxGraphic	DTIRCLensSurface	PolarizingPrism	SurfaceFacet
CircleGraphic	Fiber	PolygonalMirror	ThickLens
ClearBoundary	FresnelRhomb	PolygonGraphic	ThinLens
CompoundLens	FunnelLens	PorroPrism	ToroidalLens
ConjugateMirror	FunnelLensSurface	Prism	ToroidalLensSurface
CustomBaffle	Grating	RectangleGraphic	ToroidalMirror
CustomBeamSplitter	GratingMirror	RetardationPlate	WedgePrism
CustomBirefringentLensSurface	GRINLens	ReversionPrism	Window
CustomConjugateMirror	HalfBallLens	RhomboidPrism	WinstonConeMirror
CustomDiffuser	HollowCornerCube	RodBaffle	

If we desire to create a **`'BiConvexLens'`**, we can do the folowing.

```
? BiConvexLens
```

```
BiConvexLens[focallength, aperture, thickness, label, options] designates a lens with two equally convex spherical surfaces.

BiConvexLens is created with its first surface centered about the origin and its second surface positioned down the
    positive x axis. The aperture parameter may denote a circle, rectangle, or polygon depending on the number
    and type of elements listed by it. The user−named label parameter is optional and can be omitted. When it
    is present, its text content is used to identify the object in both the rendered graphics and the output cell
    expression. When it is omitted, Optica uses the default setting of the Labels option with the rendered graphics.

Note that the specified focallength is always determined for a particular setting of the
    DesignWaveLength and ComponentMedium options. By default, Optica uses DesignWaveLength−>0.5461
    microns and ComponentMedium−>BK7. Care should be taken to insure that these default
    settings are compatible with the intended experimental design or unintended results can occur.
```

After viewing the usage message, let us pass numerical values ot create the lens object.

```
BiConvexLens[20, 10, 5]
```

```
BiConvexLens[20, 10, 5]
```

System Visualization

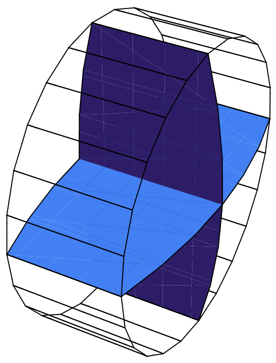
A BiConvexLens was created with focallength 20 units, aperture 10 units, and thickness 5 units. In order to display this element, we will nest the **`'BiConvexLens'`** in the **`'TurboPlot'`** function. The default view is three-dimensional. NOTE: We may use the '**`OpticaFunctions'`** command to list *Optica* functions not pertaining to optical elements and ray-sources.

```
? TurboPlot
```

```
TurboPlot[system, options] works with TurboTrace to perform
    accelerated ray−tracing and rendering of a system of light sources and optical elements.

TurboPlot[source, tracedsystem, options] can be used to more quickly trace a new light source through a
    previously traced optical system (by TurboTrace or TurboPlot). When the default setting of PlotType −>
    Surface with ReportedSurfaces −> Automatic is used, TurboPlot displays a spot diagram of the locus of
    ray intersections on the last optical surface in the traced results. However, when other PlotType settings
    are specified, all of the ray−surfaces are displayed in the same fashion as ShowSystem. TurboPlot has the
    same relationship with TurboTrace that DrawSystem does with PropagateSystem. See also TurboTrace.
```

Displaying a nested lens function.

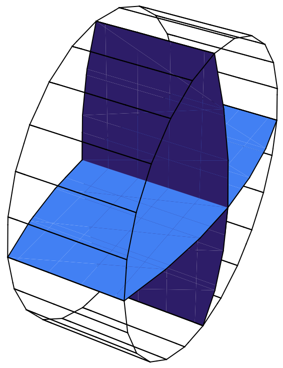
```
TurboPlot[BiConvexLens[20, 10, 5]]
```



–prepared system–

A variable can be assigned a lens and is displayed in the same manner as a nested lens function.

```
exampleLens = BiConvexLens[20, 10, 5];
TurboPlot[exampleLens]
```



–prepared system–

Move

This method of system display works for multiple elements in an optical system as well. Visualization is acheived by grouping multiple elements inside of a list. To properly position elements, we will utilize the **`'Move'`** command.

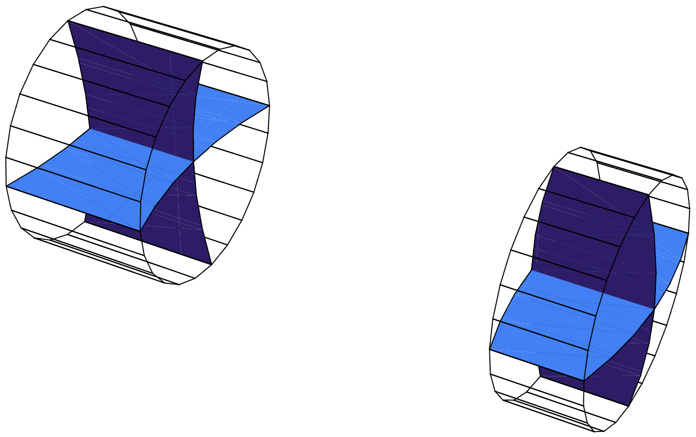
```
? Move
```

```
Move[objectset, movement, options] is used to move the relative position and orientation of a set of components and rays.

Move accepts a number of patterns to specify a movement in one, two, or three−dimensional space. In
    particular, the movement can be specified as six different sequences of parameters: x; {x, y}; {x, y, z};
    {x, y}, rotate; {x, y}, rotate, twist; {x, y, z}, rotationmatrix; {x, y, z}, tilt; and {x, y, z}, tilt, twist. Here, the
    rotation angle determines the angular orientation of objectset within the horizontal plane. To get a three
    dimensional movement, a three−dimensional tilt vector or rotation matrix is specified in addition to the
    three components of displacement, {x, y, z}. The optional twist angle, which may be given as an option
    TwistAngle −> ♯, can be used to specify a rotation around the axis of orientation (typically the optical axis).

A component or source may be moved additionally symbolically by specifying a list of a nonnumerical
    quantity and a number for each coordinate. Missing symbols are filled with the corresponding
    numerical specification, for example {{x, 10}, 5} to move symbolically {x, 5} and numerically {10, 5}.

Other related commands used for positioning components and rays include:
    Rotate, Translate, MoveSurface, MoveAligned, MoveDirected, MoveLinear, and MoveReflected.

Move −> {coordinatepoints} is an option of AnalyzeSystem/ShowSystem based on
    mouse−generated coordinatepoints that moves components and rays in the optical system.

Move uses a list of coordinatepoints taken with the mouse from the rendered graphics to specify a movement for the
    selected objects. The first coordinate point non pointing on an object indicates the point where to move objects.
    The next two points can specify an additional rotation. For one more point the objects are oriented parallel to
    the line defined by these two points. Two more points define a relative rotation, given by the lines from the
    first to the second and from the first to the third point. Object−sequence positions may also be used to specify
    objects instead of coordinatepoints. Move uses the two−dimensional coordinate system given in PlotType.

See also: DrawSystem, PlotType, ToleranceSelection, ToleranceDoubleClick, SnapGrid, SnapGridOffset,
    MoveReflected, Arrange, Rotate, Copy, Constrain, Snap, Align, Transform, Add, Swap, Remove, Identify.
```

In these optical systems, the **`'BiConvexLens'`** is moved along the x-axis in the positive x-direction by 20 units. The first example demonstrates assigning an optical system to a variable. The second demonstrates nesting element functions within a system function.

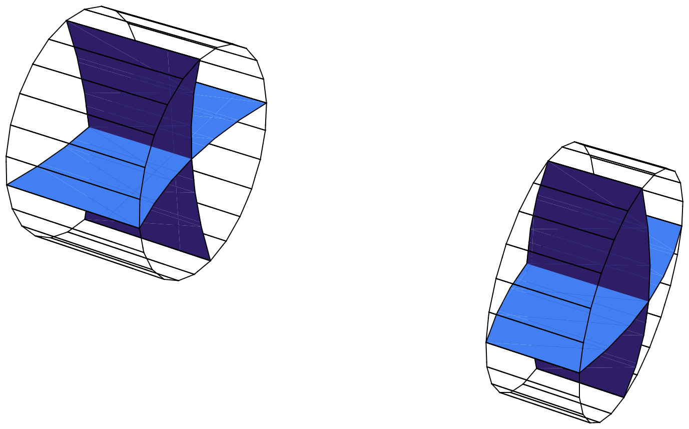
```
exampleOpticalSystem = {BiConcaveLens[-20, 10, 5], Move[BiConvexLens[20, 10, 5], 20]}
TurboPlot[exampleOpticalSystem]
```

```
{BiConcaveLens[-20, 10, 5], Move[BiConvexLens[20, 10, 5], 20.]}
```



```
-prepared system-
```

```
TurboPlot[{BiConcaveLens[-20, 10, 5], Move[BiConvexLens[20, 10, 5], 20]}]
```



```
-prepared system-
```

Source Functions

Currently, there are no rays in the optical system. They may be input using an *Optica* **`SourceFunction`**.

```
? SourceFunctions
```

SourceFunctions generates a hyperlinked listing of the common light source models available in Optica.

```
SourceFunctions
```

CircleOfRays	LineOfRays
ConeOfRays	PointOfRays
CustomRays	RainbowOfRays
GaussianBeam	SingleRay
GridOfRays	WedgeOfRays

We will use the **`'ConeOfRays'`** function as our ray source.

```
? ConeOfRays
```

ConeOfRays[seed, spread, options] initializes a set of rays distributed along a funnel–shaped surface that is oriented down the positive x–axis for the default BirthPoint setting.

The spread parameter is taken as the full cone angle, in degrees units, for the default BirthPoint setting of Automatic. However, for other BirthPoint settings, the spread parameter specifies the intercept spatial dimensions, at the y–z plane, of the ray set. If spread is a list of two positive numbers {width, height}, then the cone has rectangular shaped cross–section. If spread is a single number, then the cone has a circular shaped cross–section. If spread is a list of two negative numbers {–width, –height}, then the cone has elliptical shape given by {width, height}.

The seed parameter is optional and is used to generate more elaborate ray patterns. When it is present, it specifies additional light sources to be nested with the basic ConeOfRays result. When PropagateSystem is used for the ray trace, ConeOfRays produces a set of Ray objects. However, with TurboTrace calculations, a TurboRays object is generated from ConeOfRays instead. The distribution of rays is controlled by the GridSpacing option and the starting origin is defined by the BirthPoint option.

See also: NumberOfRays, MonteCarlo, GridSpacing, SourceFraction, SourceOffset, BirthPoint, SingleRay, CircleOfRays, WedgeOfRays, LineOfRays, GridOfRays, PointOfRays, CustomRays, FieldOfRays, GaussianBeam, and RainbowOfRays.

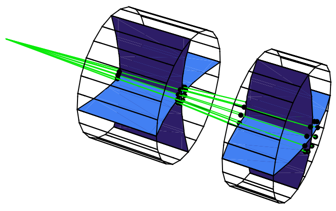
Source Function Options

To specify an arbitrary number of rays, we use the `'NumberOfRays'` option.

```
?NumberOfRays
```

NumberOfRays is an option of ray sources that designates the number of rays to be initially created by a ray source function. See also: MonteCarlo.

```
TurboPlot[{ConeOfRays[5, NumberOfRays → 10],
    Move[BiConcaveLens[-20, 10, 5], 10],
    Move[BiConvexLens[20, 10, 5], 20]}]
```



-traced system-

Note that the raytrace stops at the second surface of the `'BiConvexLens'`.

Key System Elements

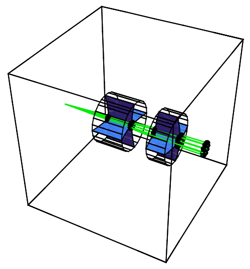
In *Optica*, we need to specify a boundary or a screen as our ray terminus. This way we can prevent premature ray termination.

```
?Boundary
```

Boundary[boundaryparameters, label, options] designates a rectangular box that absorbs rays intercepted by its walls.

Here, the user-named label parameter is optional and can be omitted. When it is present, its text content is used to identify the object in both the rendered graphics and the output cell expression. There are four 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, Boundary[xside, yside, zside] assumes a three-dimensional box that has a length specified by xside, a width specified by yside, and a height specified by zside, and Boundary[aside, bside] assumes a three-dimensional box that has a length specified by aside, a width specified by bside, and a height specified by bside. Optical systems usually have at least one boundary component listed at the end. By default, Boundary is not rendered. See also: ClearBoundary.

```
TurboPlot[{ConeOfRays[5, NumberOfRays → 10],
    Move[BiConcaveLens[−20, 10, 5], 10],
    Move[BiConvexLens[20, 10, 5], 20],
    Boundary[30]}]
```



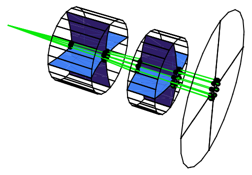
−traced system−

```
? Screen
```

Screen[aperture, label, options] refers to a planar component that intersects rays without disturbing them.

Although Screen is usually not rendered, the ray/screen intersections can be rendered by using the DrawSystem option, PlotType−>Surface. Screen is created with its surface centered about the origin. The aperture parameter may indicate a circle, rectangle, or polygon depending on the number and type of elements listed by it. The user−named label parameter is optional and can be omitted. When it is present, its text content is used to identify the object in both the rendered graphics and the output cell expression. When it is omitted, Optica uses the default setting of the Labels option with the rendered graphics.

```
TurboPlot[{ConeOfRays[5, NumberOfRays → 10],
    Move[BiConcaveLens[−20, 10, 5], 10],
    Move[BiConvexLens[20, 10, 5], 20],
    Move[Screen[20], 30]}]
```

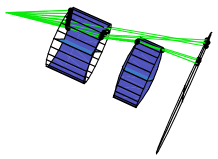


−traced system−

It is worth noting that the **'Boundary'** function will work for any extraneous rays that will NOT intersect a screen.

Here the rays are rotated 20 degrees towards the positive y - axis.
In this model, premature ray-termination is at the first surface of the 'BiConcaveLens'.

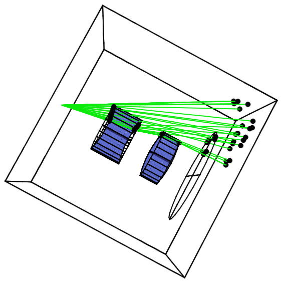
```
TurboPlot[{
   Move[ConeOfRays[20, NumberOfRays → 20], 0, 20],
    Move[BiConcaveLens[−20, 10, 5], 10],
   Move[BiConvexLens[20, 10, 5], 20],
   Move[Screen[20], 30]}]
```



−traced system−

Below, the rays are fully traced when a **'Boundary'** is in place.

```
TurboPlot[{
   Move[ConeOfRays[20, NumberOfRays → 20], 0, 20],
   Move[BiConcaveLens[−20, 10, 5], 10],
   Move[BiConvexLens[20, 10, 5], 20],
   Move[Screen[20], 30], Boundary[35]}]
```



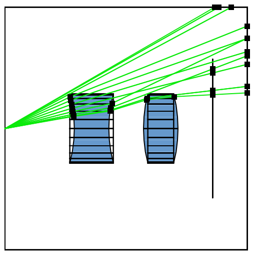
−traced system−

Basic System Options

PlotType

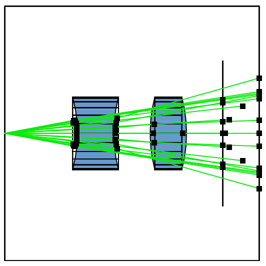
Different views of optical systems may be chosen. The default view for **`'TurboTrace'`** is **`'Full3D'`**. If we desire the view to be from above, or from the side, we can either specify the option **`'PlotType -> TopView'`** or 'PlotType -> SideView' respectively. These options are input after the end of the optical system list and before the end of the visualization function.

```
TurboPlot[{Move[ConeOfRays[20, NumberOfRays → 20], 0, 20],
    Move[BiConcaveLens[-20, 10, 5], 10],
    Move[BiConvexLens[20, 10, 5], 20],
    Move[Screen[20], 30],
    Boundary[35]}, PlotType → TopView]
```



−traced system−

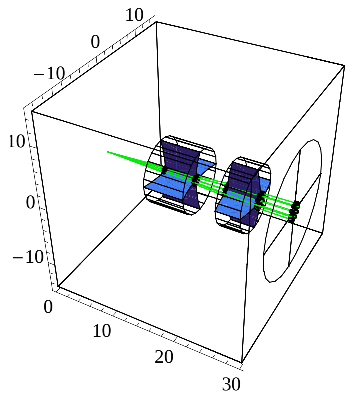
```
TurboPlot[{Move[ConeOfRays[20, NumberOfRays → 20], 0, 20],
    Move[BiConcaveLens[-20, 10, 5], 10],
    Move[BiConvexLens[20, 10, 5], 20],
    Move[Screen[20], 30],
    Boundary[35]}, PlotType → SideView]
```



−traced system−

If we wish the view system coordinates, we use the **'Axes -> True'** option.

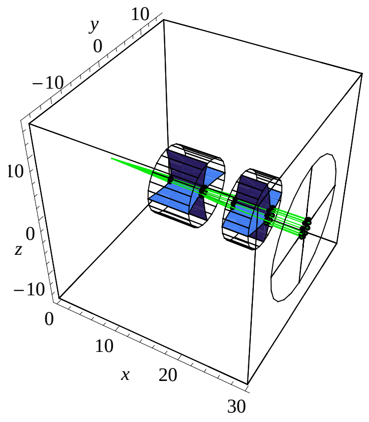
```
TurboPlot[{ConeOfRays[5, NumberOfRays → 10],
  Move[BiConcaveLens[-20, 10, 5], 10],
  Move[BiConvexLens[20, 10, 5], 20],
  Move[Screen[20], 30],
  Boundary[30]}, Axes → True]
```



```
-traced system-
```

To view the model coordinate system, use the option **'AxesLabel -> {x-coordinateLabel, y-coordinateLabel, z-coordinateLabel}'**.

```
TurboPlot[{ConeOfRays[5, NumberOfRays → 10],
  Move[BiConcaveLens[-20, 10, 5], 10],
  Move[BiConvexLens[20, 10, 5], 20],
  Move[Screen[20], 30],
  Boundary[30]}, Axes → True, AxesLabel → {x, y, z}]
```



```
-traced system-
```

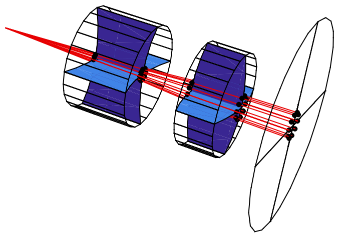
WaveLength

Up to this point, the default wavelength as set by the source has been 532 nm. Suppose that we desire a different wavelength of light to propagate through our optical system, we can use the `'WaveLength'` option. Let us use the wavelength of a ruby laser at 694.3 nm, assuming a non-Gaussian beam.

```
? WaveLength
```

WaveLength is a rule of Ray that describes the ray's optical wavelength (in microns).

```
TurboPlot[{ConeOfRays[5, NumberOfRays → 10, WaveLength → .6943],
  Move[BiConcaveLens[-20, 10, 5], 10],
  Move[BiConvexLens[20, 10, 5], 20],
  Move[Screen[20], 30]}]
```



```
-traced system-
```

Sequential vs. Non - Sequential Ray - Tracing

Sequential Ray - Tracing

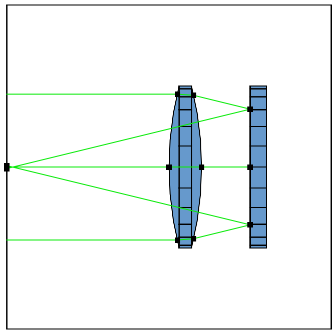
In order to further exploit *Optica*' s functionality, we must understand how *Optica* performs ray-tracing. Each ray is composed of a database that defines its properties. Some properties defined in the database are wavelength, intensity and position. Each time a ray intersects an optical component surface, a new ray is created with a new set of properties updated by the component surface.

Through this process, the intersected component surface is marked as read when a ray first hits it.

Optica can interperet the marking in two different ways by means of sequential and non-sequential ray-tracing. Through sequential ray-tracing, a surface affects rays only once.

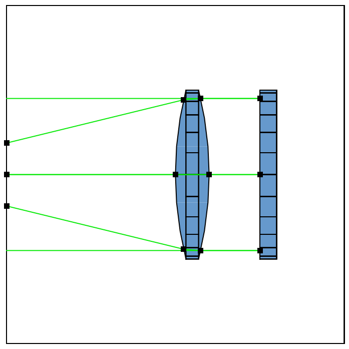
Seen here is an example of sequential ray-tracing. Note that rays are not refracted their second time through the **'BiConvexLens'**.

```
TurboPlot[{LineOfRays[45],
      Move[BiConvexLens[100, 50, 10], 50],
      Move[Mirror[50, 5], 75],
      Boundary[100]}, PlotType → TopView];
```



Optica performs ray-trace calculations based on the order elements are listed in an optical system. That is, the first object in a list will be first to traced regardless of its physical position in an optical system. In the example below, the 'Mirror' is the first surface affected by the trace. Next, the 'BiConvexLens' refracts the reflected rays. As is evident, the trace is perfomed out of our desired order.

```
TurboPlot[{LineOfRays[45],
   Move[Mirror[50, 5], 75],
   Move[BiConvexLens[100, 50, 10], 50],
   Boundary[100]}, PlotType → TopView];
```



Non-Sequential Ray-Tracing

As seen before, a sequential ray-trace can be problematic. In order to bypass this issue, non-sequential ray-tracing must be invoked. The non-sequential ray-trace can be invoked by simply nesting all system elements that need to be non-sequentially traced inside the **`'Resonate'`** function.

```
? Resonate
```

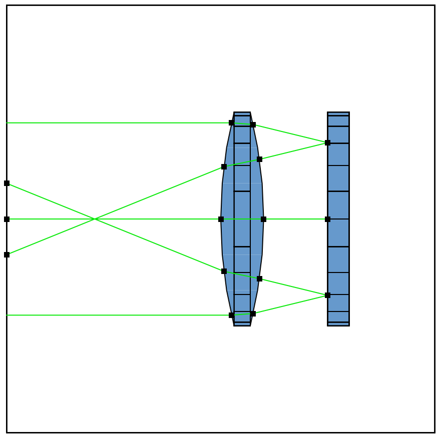
Resonate[listofcomponents, objectname, options] is a generic building block, labeled by an optional objectname, that causes a ray to be nonsequentially traced within all of the surfaces defined by listofcomponents.

Resonate takes a list of Component objects and creates a single Component object from the list. Once the all of the surfaces are no longer in its trajectory, the ray is allowed to continue propagating to other components. This effect enables repeated nonsequential ray interactions with a number of surfaces to occur.

In PropagateSystem, Resonate is also used as an unauthorized option of Ray. A Ray object carrying Resonate -> True will loop through surface targets within the resonant component until no surface intersection can be found within the component. See also: Confine and Unconfine.

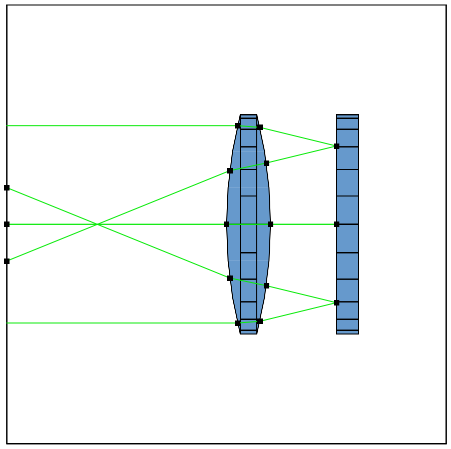
In this example, the **`'Mirror'`** and **`'BiConvexLens'`** are nested within the **`'Resonate'`** function, as these components require Non-Sequentual ray-trace invocation.

```
TurboPlot[{LineOfRays[45],
    Resonate[{Move[BiConvexLens[100, 50, 10], 50],
        Move[Mirror[50, 5], 75]}],
    Boundary[100]}, PlotType → TopView];
```



'Resonate' also fixes ray-traces that would be considered out of order.

```
TurboPlot[{LineOfRays[45],
   Resonate[{
     Move[Mirror[50, 5], 75],
     Move[BiConvexLens[100, 50, 10], 50]}],
   Boundary[100]}, PlotType → TopView];
```



Data Handling and System Calculations with *Optica*

Data Handling

Suppose we want to create an optical system that contains a **'LensDoublet'**. We can manually specify our parameters using the **'LensDoublet'** function.

```
? LensDoublet
```

LensDoublet[r1, r2, r3, aperture, t1, t2, index1, index2, label, options] designates a lens having three spherical surfaces.

The radii of curvatures are specified by r1, r2, and r3, where Infinity is used to indicate planar surfaces. In addition, LensDoublet is specified by two thicknesses: t1, t2. The two refractive indexes must also be specified either as a function of wavelength or using one of the built–in optical material labels. LensDoublet is created with its first surface centered about the origin and the remaining surfaces positioned down the x axis. The aperture parameter may designate a circle, rectangle, or polygon depending on the number and type of elements listed by it. The user–named label parameter is optional and can be omitted. When it is present, its text content is used to identify the object in both the rendered graphics and the output cell expression. When it is omitted, Optica uses the default setting of the Labels option with the rendered graphics. See also: LensTriplet and CompoundLens.

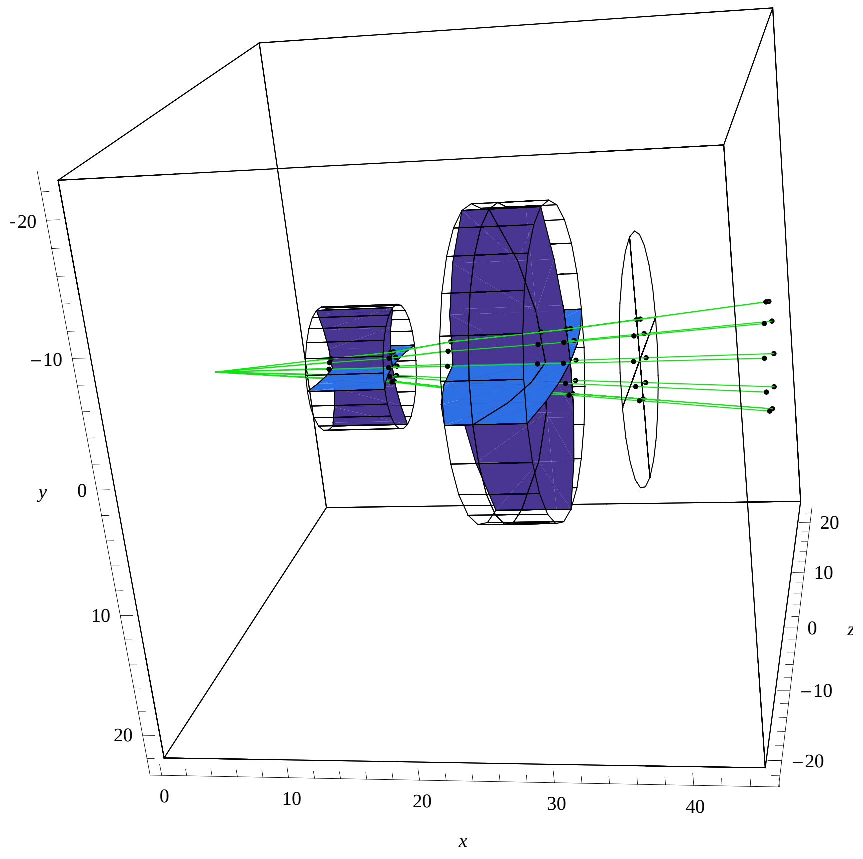
However, *Optica* contains an optical element database. With this understanding,we can import data from *Optica's* database, assign the data to a variable, then use the variable as an optical element. This act is executed following the form: '**componentVariable = DataToOptica[*desiredComponent*]**'.

```
lensDoubletVariable = DataToOptica[LensDoublet]
```

```
LensDoublet[38.35, -26.81, -71.89, 25.4, 7.5, 2., BK7, SF5]
```

We can display this element in an optical system.

```
TurboPlot[{ConeOfRays[10, NumberOfRays → 10],
  Move[BiConcaveLens[-20, 10, 5], 10],
  Move[lensDoubletVariable, 20],
  Move[Screen[20], 35],
  Boundary[45]}, Axes → True, AxesLabel → {x, y, z}]
```

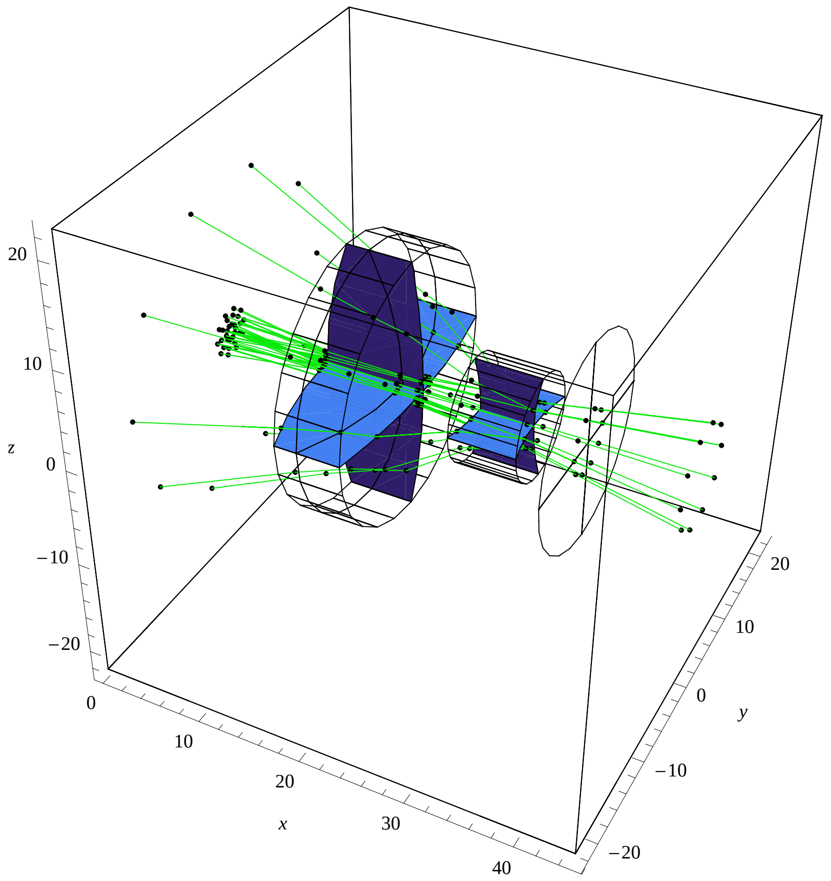


```
-traced system-
```

Optica also has tools to provide for real world optical emulation. One such tool is the **'FresnelReflections'** option. When this option is set to **'True'** *Optica* takes into account Fresnel reflections resulting from an uncoated lens surface. The option may only be used when creating a lens. Input follows the form **'DeclaredLens[*parameters*, FresnelReflections -> True]'**.

Displayed next is an optical system containing a manually input **'LensDoublet'** and **'BiConcave'** lens. **'Resonate'** is used in case any reflected stray rays require retracing.

```
TurboPlot[{ConeOfRays[10, NumberOfRays → 10],
  Resonate[{Move[BiConcaveLens[-20, 10, 5, FresnelReflections → True], 25],
    Move[LensDoublet[38.34999999999995, -26.81, -71.88999999999987,
      25.4, 7.5, 2, BK7, SF5, FresnelReflections → True], 10]}],
  Move[Screen[20], 35],
  Boundary[45]}, Axes → True, AxesLabel → {x, y, z}]
```



-traced system-

Energy is now lost from the source function to the final surface due to the fact that **'FresnelReflections'** are taken into account. If we would like to view loss in system energy, we may use the **'FindIntensity'** function.

```
?FindIntensity
```

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

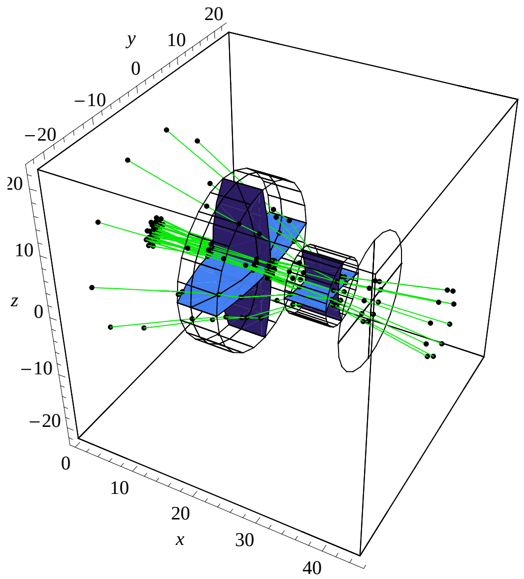
For its input, FindIntensity works best with either an untraced, raw, optical system or a previously calculated result of FindIntensity. However, when necessary, FindIntensity can also work with externally generated trace results. FindIntensity works equally well for surfaces that either are close to a focal plane or far from any focus. If a light sheet light source is used (ie. WedgeOfRays or LineOfRays), then a one−dimensional intensity function is automatically calculated. If a volume−filling light source is used (ie. PointOfRays, GaussianBeam, or GridOfRays), then the intensity calculations are automatically carried out for each reported surface in two−dimensions.

FindIntensity uses the KernelScale and SmoothKernelSize options to specify a Gaussian smoothing kernel that gets convolved with the intensity data. In effect, this determines the maximum resolution of the FindIntensity calculation. As a second argument to FindIntensity, the absolutekernelsize parameter can be directly specified in FindIntensity in order manually set the SmoothKernelSize to an absolute value. For example, absolutekernelsize can be manually set by the user to a diffraction−limited spot−size of the systen in order to incorporate diffraction behavior together with the geometric ray−trace data. Otherwise, FindItensity automatically determines a SmoothKernelSize based on the surface area and number of available sample points. In that case, the kernel size has no relation to the diffraction performance of the system.

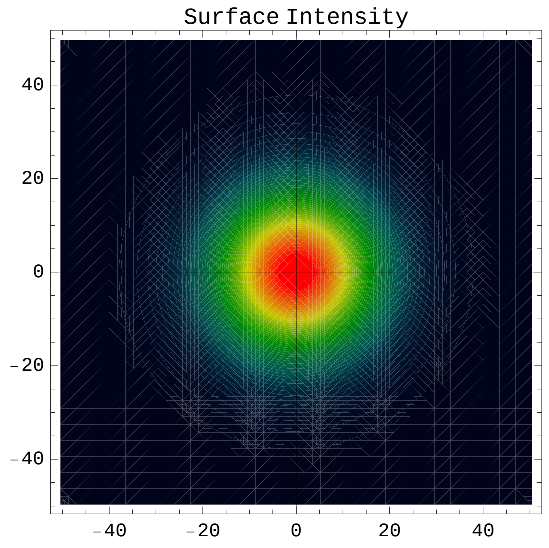
FindIntensity uses the IntensityScale option to specify spatial scaling of the system.

FindIntensity uses the Energy option to manually specify the total integrated energy at the surface. With the default Energy−> Automatic setting, the integrated energy is automatically determined from the ray−trace information. However, Energy −> number is used to manually specify a constant value for the total integrated energy. See also: IntensitySetting, KernelScale, SmoothKernelSize, and SmoothKernelRange.

```
FindIntensity[TurboPlot[{ConeOfRays[10, NumberOfRays → 10],
   Resonate[{Move[BiConcaveLens[-20, 10, 5, FresnelReflections → True], 25],
     Move[LensDoublet[38.34999999999995, -26.81, -71.88999999999987,
       25.4, 7.5, 2, BK7, SF5, FresnelReflections → True], 10]}],
   Move[Screen[20], 35],
   Boundary[45]}, Axes → True, AxesLabel → {x, y, z}]]
```

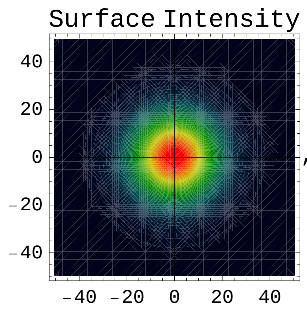


```
Surface Information :
 {ComponentNumber → 3., SurfaceNumber → 3., NumberOfRays → 40, SmoothKernelSize → 17.5197}
```



```
{IntensityFunction → CompiledFunction[-intensity data-],
```

```
OutputGraphics →
```



```
, ComponentNumber → 3.,
SurfaceNumber → 3., WaveFrontID → 1., SmoothKernelSize → 17.5197,
RayBoundary → {{-15.3976, 15.3976}, {-14.6387, 14.6387}},
NumberOfRays → 40, Full3D → True, Energy → 16.5299}
```

This output for **'FindIntensity'** contains a key pieces of information. The first detail is that **'FindIntensity'** is viewing raying intensity on the third component's third surface due to the **'ComponentNumber'** and **'SurfaceNumber'** indicators. Next, we see that the total **'Energy'** at that position is 16.5299 units. Keep in mind that the source function outputs power at 100 units by default. Based on this data, approximately 16.5% of the initial energy is transmitted through the system. Viewing the **'Surface Intensity'** diagram, we can see that rays from the **'ConeOfRays'** function is depicted here.

Suppose that viewing the energy at another component and surface is desired. In order to perform the calculation, we must use the **'ReadRays'** option nested inside the **'FindIntensity'** function.

```
? ReportedSurfaces
```

ReportedSurfaces–>settings is an option of TurboTrace and PropagateSystem
that indicates which surface results get included in the final ray–tracing report.

Special values are All and Last (or {–1}), which specify all surfaces and the last
respectively. Both TurboTrace and PropagateSystem accept a component number or a list of
component numbers to be reported. A sublist of componentnumber and surfacenumbers inside
the component may also be used. These numbers relate to the sequence of the system definition.

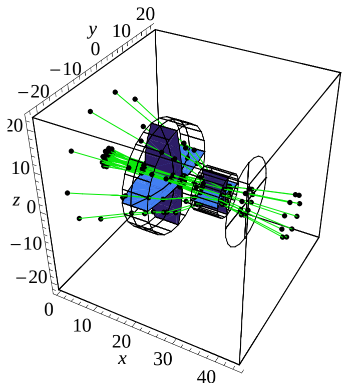
In TurboTrace, the behavior of the ReportedSurfaces –> Last setting is dependent on the SequentialTrace option. For
SequentialTrace –> True, only rays striking the last surface present in the trace sequence gets reported.
However, for SequentialTrace –> False, all ray intercepts from the last Component's surfaces are reported.

ReportedSurfaces also accepts references to ComponentNumber, SurfaceNumber, and SurfaceID. An example of valid
settings is: ReportedSurfaces–>{{–1,5,{ComponentNumber–>3,SurfaceNumber–>2}, {3,3}, {SurfaceID–>24}}.

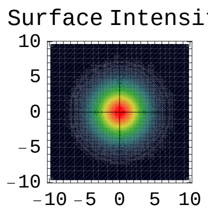
See also: ReportedFunction, OutputType, RayTraceFunction, SurpressMissedRays, and ReportedParameters.

Let's see how much energy is present at the first surface of the **'LensDoublet'**.

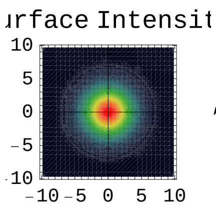
```
FindIntensity[TurboPlot[{ConeOfRays[10, NumberOfRays → 10],
   Resonate[{Move[BiConcaveLens[-20, 10, 5, FresnelReflections → True], 25],
     Move[LensDoublet[38.34999999999995, -26.81, -71.88999999999987,
       25.4, 7.5, 2, BK7, SF5, FresnelReflections → True], 10]}],
   Move[Screen[20], 35],
   Boundary[45]}, Axes → True, AxesLabel → {x, y, z}], ReportedSurfaces → {{1, 1}}]
```



```
Surface Information :
 {ComponentNumber → 1., SurfaceNumber → 1., NumberOfRays → 20, SmoothKernelSize → 3.03635}
```



```
{IntensityFunction → CompiledFunction[-intensity data-],

 OutputGraphics →
```

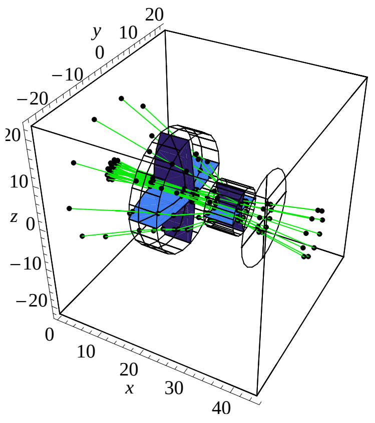


```
 , ComponentNumber → 1., SurfaceNumber → 1., WaveFrontID → 1.,

 SmoothKernelSize → 3.03635, RayBoundary → {{-3.79353, 3.79353}, {-3.60654, 3.60654}},
 NumberOfRays → 20, Full3D → True, Energy → 93.0193}
```

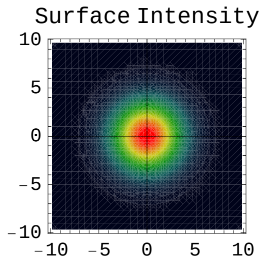
In the previous example, we see that the energy transmsitted through the first surface of the **'LensDoublet'** is approximately 93 energy units.

Utilizing our knowledge of lists, we can specify the intensity at multiple components and surfaces of an element to be calculated at once. In other words, the intensity at all optical system surfaces can be calculated with one command.

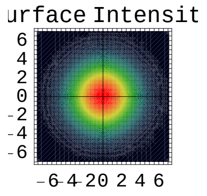
```
FindIntensity[TurboPlot[{ConeOfRays[10, NumberOfRays → 10],
   Resonate[{Move[BiConcaveLens[-20, 10, 5, FresnelReflections → True], 25],
     Move[LensDoublet[38.34999999999995, -26.81, -71.88999999999987,
       25.4, 7.5, 2, BK7, SF5, FresnelReflections → True], 10]}],
   Move[Screen[20], 35],
   Boundary[45]}, Axes → True, AxesLabel → {x, y, z}], ReportedSurfaces → {All}]
```



```
Surface Information :
 {ComponentNumber → 1., SurfaceNumber → 1., NumberOfRays → 20, SmoothKernelSize → 3.03635}
```

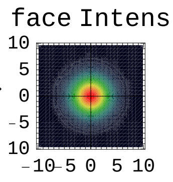


```
Surface Information :
 {ComponentNumber → 1., SurfaceNumber → 2., NumberOfRays → 10, SmoothKernelSize → 2.49542}
```

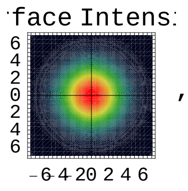


```
Surface Information :
 {ComponentNumber → 1., SurfaceNumber → 3., NumberOfRays → 40, SmoothKernelSize → 11.732}
```

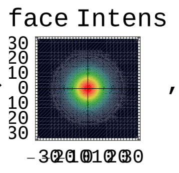
```
(*There is more of this type of output, however it has been surpressed in
 order to save page space.  Feel free to reevaluate the equation if you wish.

{{IntensityFunction → "CompiledFunction[-intensity data-]",

  OutputGraphics →
```



```
  , ComponentNumber → 1.`, SurfaceNumber → 1.`,

  WaveFrontID → 1.`, SmoothKernelSize → 3.0363489632837446`, RayBoundary →
   {{-3.793525530940937`, 3.793525530940937`}, {-3.606539375522785`, 3.6065393755225177`}},
  NumberOfRays → 20, Full3D → True, Energy → 93.01934056815956`},
 {IntensityFunction → "CompiledFunction[-intensity data-]",

  OutputGraphics →
```



```
  , ComponentNumber → 1.`, SurfaceNumber → 2.`,

  WaveFrontID → 1.`, SmoothKernelSize → 2.495424261919429`,
  RayBoundary → {{-2.1243756474607176`, 2.1243756474607176`},
    {-2.0196633338772236`, 2.0196633338771375`}}, NumberOfRays → 10, Full3D → True,
  Energy → 85.58133022874699`}, {IntensityFunction → "CompiledFunction[-intensity data-]",

  OutputGraphics →
```



```
  , ComponentNumber → 1.`, SurfaceNumber → 3.`,

  WaveFrontID → 1.`, SmoothKernelSize → 11.732022748417007`, RayBoundary →
   {{-10.314043825063209`, 10.314043825063209`}, {-9.805655681650938`, 9.805655681650022`}},
  NumberOfRays → 40, Full3D → True, Energy → 112.82857719197095`}
```

A More Abstact Approach

Symbolic Parameters

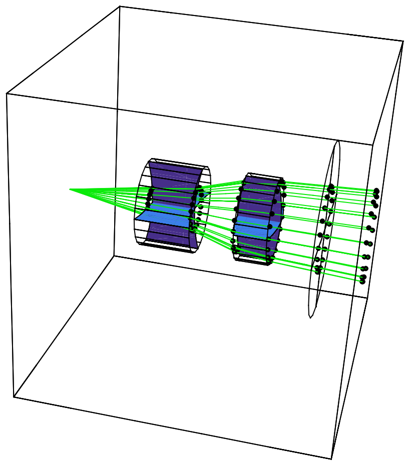
Mathematica supports symbolic representations of variables.   To demonstrate this concept, we pass the undefined variables 'g' and 'h' into respective **'ExampleFunction'** parameters. Since **'ExampleFunction'**
is defined by **'ExampleFunction[*x_*, *y_*] := x^3 + y^2'**, we expect the result **' *g*^{3} + *h*^{2}'**.

```
ExampleFunction[g, h]
```

$g^3 + h^2$

Since, *Optica* is based on the *Mathematica* backbone, *Optica* has *Mathematica's* native support for symbolic representations of variables. In order to pass a symbolic variable to an *Optica* function, the variable must be paired with a numerical value specified as a paramter formatted in a list. For example, if we want to symbolically pass the variable 'f' to the focallength paramter of the **'BiConcaveLens'** function, the proper formatting is as follows :
'BiConcaveLens[{f, -20}, 10, 5]'. A system that includes this principle is shown next.

```
TurboPlot[{
    Move[ConeOfRays[20, NumberOfRays → 20], 0, 0],
    Move[BiConcaveLens[{f, −20}, 10, 5], 10],
     Move[BiConvexLens[20, 10, 5], 20],
    Move[Screen[20], 30], Boundary[35]}]
```

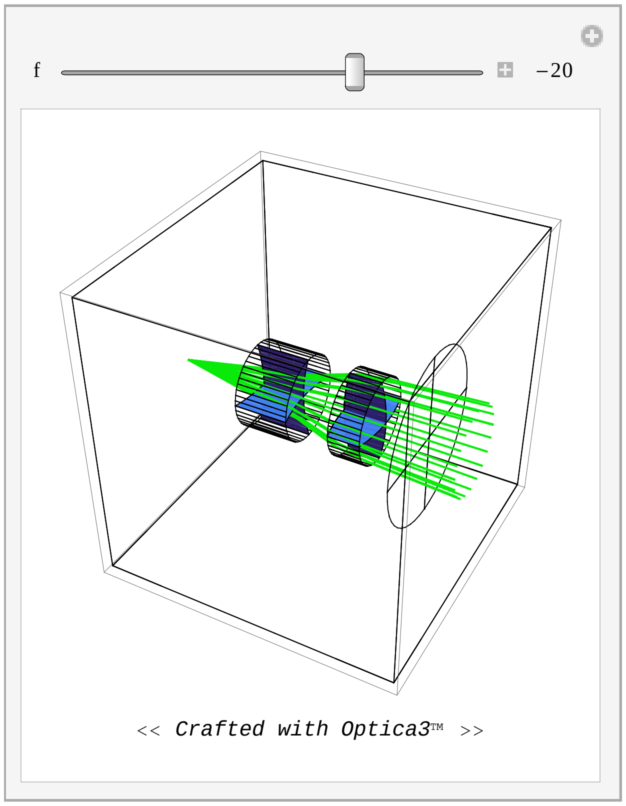


−traced system−

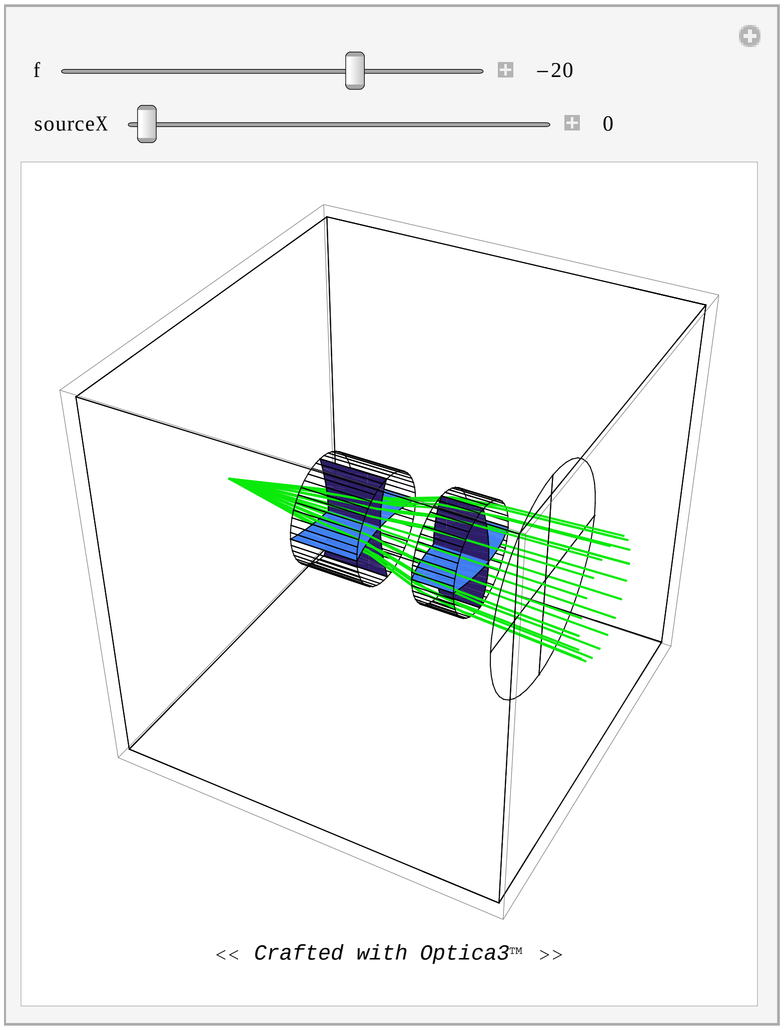
Dynamic Updating with Manipulate System

A variable parameter in **`'TurboPlot'`** cannot be manipulated in realtime. *Optica* does support dynamic variable updating during a realtime raytrace with the **`'ManipulateSystem'`** function. **`'ManipulateSystem'`** invokes the *Mathematica 6* function **`'Manipulate'`**. If we take the lens system from the 'TurboPlot' example, and utilize the **`'ManipulateSystem'`** function instead, we will see that the **`'BiConcaveLens'`** changes shape with respect to the dynamic update of 'f'.

```
ManipulateSystem[
 {Move[ConeOfRays[20, NumberOfRays → 20], 0, 0], Move[BiConcaveLens[{f, −20}, 10, 5], 10],
   Move[BiConvexLens[20, 10, 5], 20],
   Move[Screen[20], 30], Boundary[35]},
 {{f, −20}, −40, −12}, ImageSize → 550, PlotType → Full3D]
```



```
ManipulateSystem[{Move[ConeOfRays[20, NumberOfRays → 20], {sourceX, 0}, {sourceAngle, 0}],
    Move[BiConcaveLens[{f, −20}, 10, 5], {biConcaveLensX, 10}],
    Move[BiConvexLens[20, 10, 5], 20],
    Move[Screen[20], 30], Boundary[35]}, {{f, −20}, −40, −12},
  {{sourceX, 0}, 0, 40}, ImageSize → 550, PlotType → Full3D]
```



```
f        −20
sourceX  0
```

More information regarding **`'ManipulateSystem'`** can be found in the 'ManipulateSystem Reference Guide'.   That guide should enable users to create more dynamic parameters.