Raypier Documentation

Release 0.1

Bryan Cole

Mar 14, 2021

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION TO RAYPIER

Raypier is a non-sequential optical ray-tracing program. It is intended as a design tools for modelling optical systems (cameras, imaging systems, telescopes etc.).

The main features of ray-trace are:

- Non-sequential tracing (no need to specify the order of optical components)
- Physical optics propagation with "Gausslet" tracing and beam decomposition
- Nice visualisation of the traced result
- Live update to the traced result as the user adjusts the model
- Reasonable performance (tracing algorithms runs at C-speed using Cython)
- STEP export of models for integration with CAD design (using PythonOCC)
- Saving / Loading models in YAML format.
- Trace rays with full polarisation and phase information
- Define Zernike Polynomial sequences and apply them as distortions to surfaces
- Dielectric Materials with simple-coating supported, including dispersion
- A basic library of materials (from RefractiveIndex.info)
- Various analysis algorithms including E-field evaluation by sum-of-Gaussian-Modes, and dispersion calculations for ultra-fast optics applications.

At present, the primary means of using raypier is to create your model in the form of a python script. However, it is possible to launch an empty model and then add in components from the GUI / Menu.

A minimal empty model looks like:

```
from raypier.api import RayTraceModel
model = RayTraceModel()
model.configure_traits()
```

This opens a GUI window from which you can add model objects using the Insert menu.

CHAPTER

TWO

BUILD AND INSTALLATION

Installing the dependencies is probably the biggest hurdle to using Raypier. The conda package/environment manager is far and away the easiest means of getting all the requirements installed.

Once you have the requirements, building raypier requires a compiler. The usual process of:

```
python setup.py build
sudo python setup.py install
```

should work [no sudo if you're on Windows]. I've never tried building on a Mac.

If I could figure out how the heck conda-forge worked, I'd probably use it.

CHAPTER

THREE

THE COMPONENTS OF A RAYPIER MODEL

The RayTraceModel object is a container for the following components:

- **Optics** - these represent your optical elements like lenses, mirrors polarisers etc.
- **Sources** - these generate the input rays for the model. The sources also hold the traced rays output of the tracing operation
- **Probes** - These are objects which select or sample the tracing operation result. Probes have a 3D position and orientation.
- **Results** - Results represent calculated quantities to be evaluated after each trace. Results do not have a 3D position.
- **Constraints** - Constraints are auxillary objects used to co-ordinate the parameters of a model for more convenient manipulation.

While all of the above objects are optional, you probably want at least one source object in your model (otherwise, the result will be rather uninteresting).

CHAPTER

FOUR

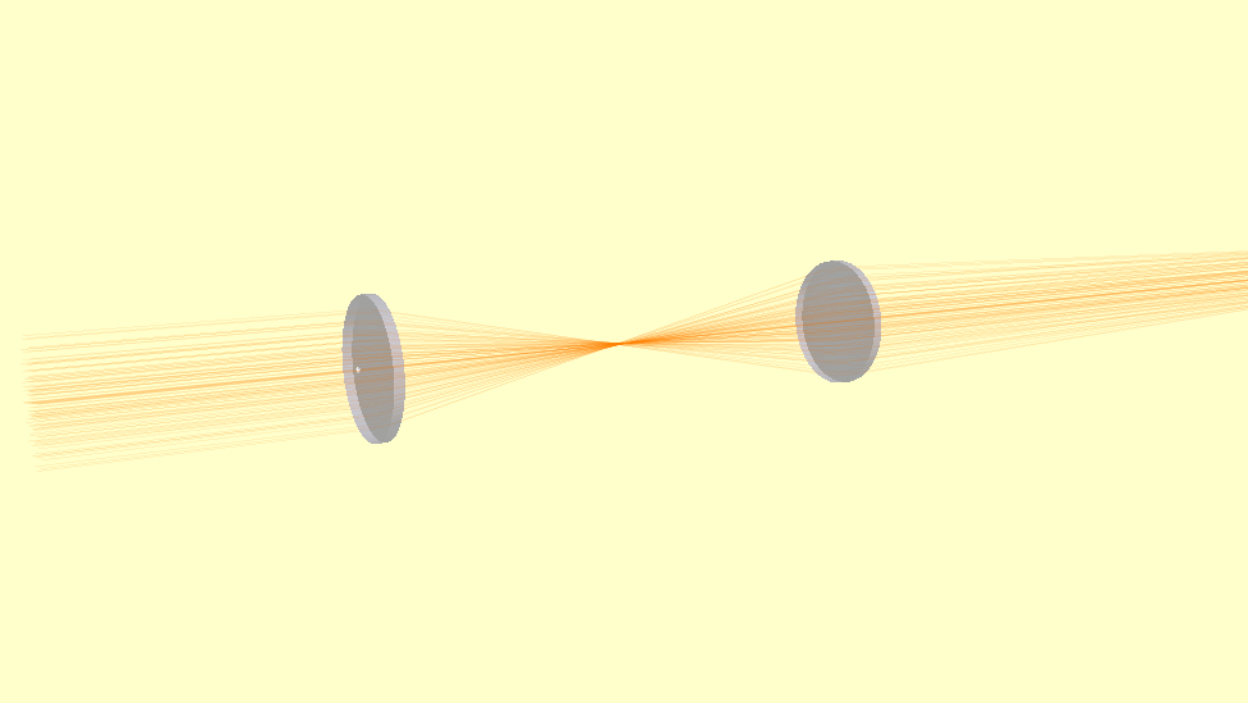
BASIC USAGE

I recommend writing models as a script, then calling `RayTraceModel.configure_traits()` on the model to view the model in the GUI.

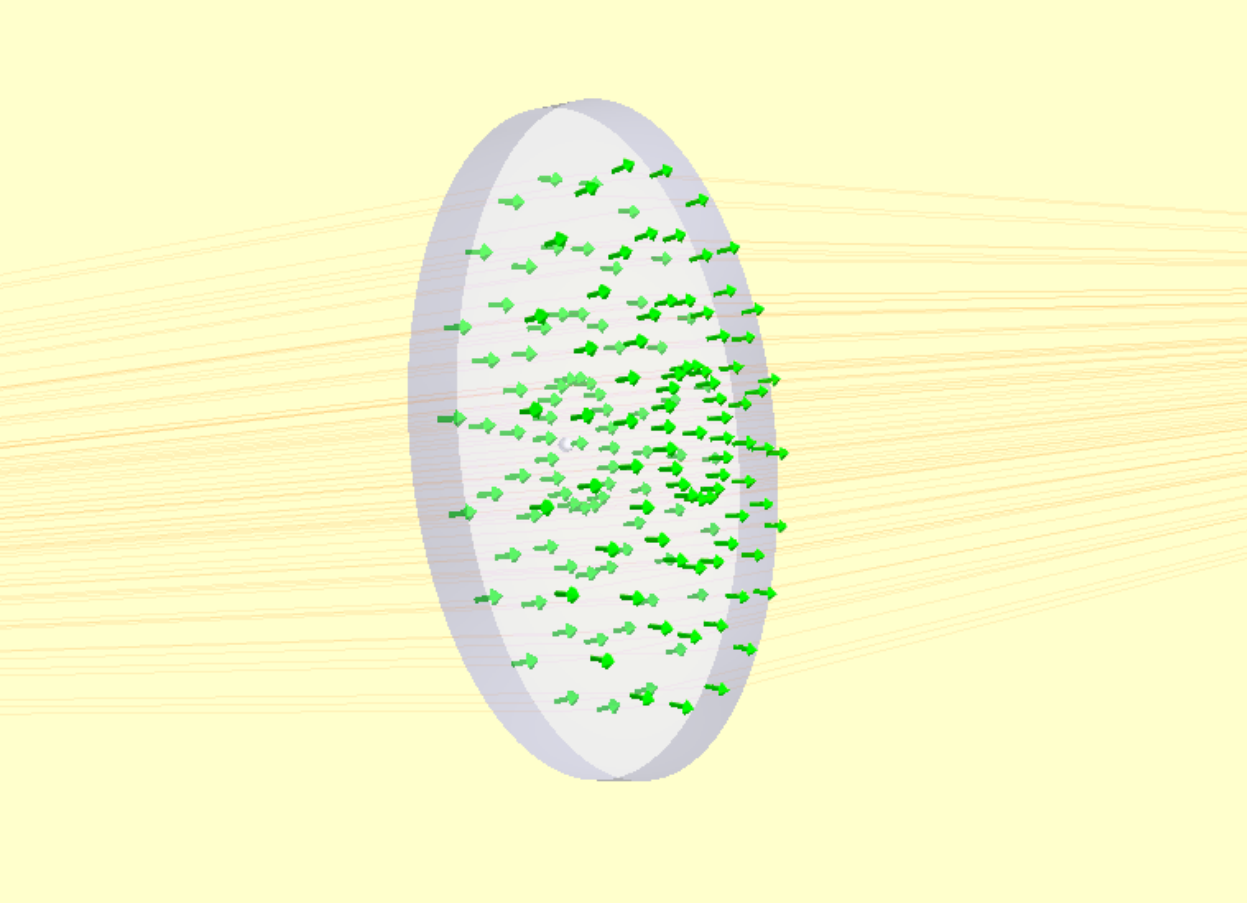
The basic method of model construction is to create instances of all the optical components you need, create one or more source-objects, and whatever probes, result or constraint objects, then give them all to an instance of RayTraceModel. For example:

```
from raypier.api import RayTraceModel, GeneralLens, ParallelRaySource, SphericalFace,␣
↪CircleShape, OpticalMaterial

### Build a couple of lenses ###
shape = CircleShape(radius=12.5)
f1 = SphericalFace(curvature=-50.0, z_height=0.0)
f2 = SphericalFace(curvature=50.0, z_height=5.0)
m = OpticalMaterial(glass_name="N-BK7")
lens1 = GeneralLens(centre=(0,0,0),
                    direction=(0,0,1),
                    shape=shape,
                    surfaces=[f1,f2],
                    materials=[m])
lens2 = GeneralLens(centre=(0,0,100.0),
                    direction=(0,0,1),
                    shape=shape,
                    surfaces=[f1,f2],
                    materials=[m])

### Add a source ###
src = ParallelRaySource(origin=(0,0,-50.0),
                        direction=(0,0,1),
                        rings=5,
                        show_normals=False,
                        display="wires",
                        opacity=0.1)

model = RayTraceModel(optics=[lens1,lens2],
                        sources=[src])

###Now open the GUI###
model.configure_traits()
```

Here's our model:



If we set *show_normals=True* on the source object, the rendering show the norma-vectors at each surface intersection. This is a useful sanity check to be sure your model is behaving physically.



Retracing of the model occurs whenever any parameter changes. You can explicitly force a re-trace using the `RayTraceModel.trace_all()` method. I.e.:

```
model.trace_all()
```

You can access the results of the trace as the `RayCollection.traced_rays` list on the source object. E.g.:

```
for rays in src.traced_rays:
    one_ray = rays[0]
    print(one_ray.origin, one_ray.accumulated_path)
```

Sometimes, having the model re-trace on every change isn't what you want (for example, if you're doing a batch calculation, or running an optimisation). You can block re-tracing using the `hold_off()` context manager. I.e.:

```
with model.hold_off():
    lens1.shape.radius=10.0
    src.origin=(0,0,-100.0)
```

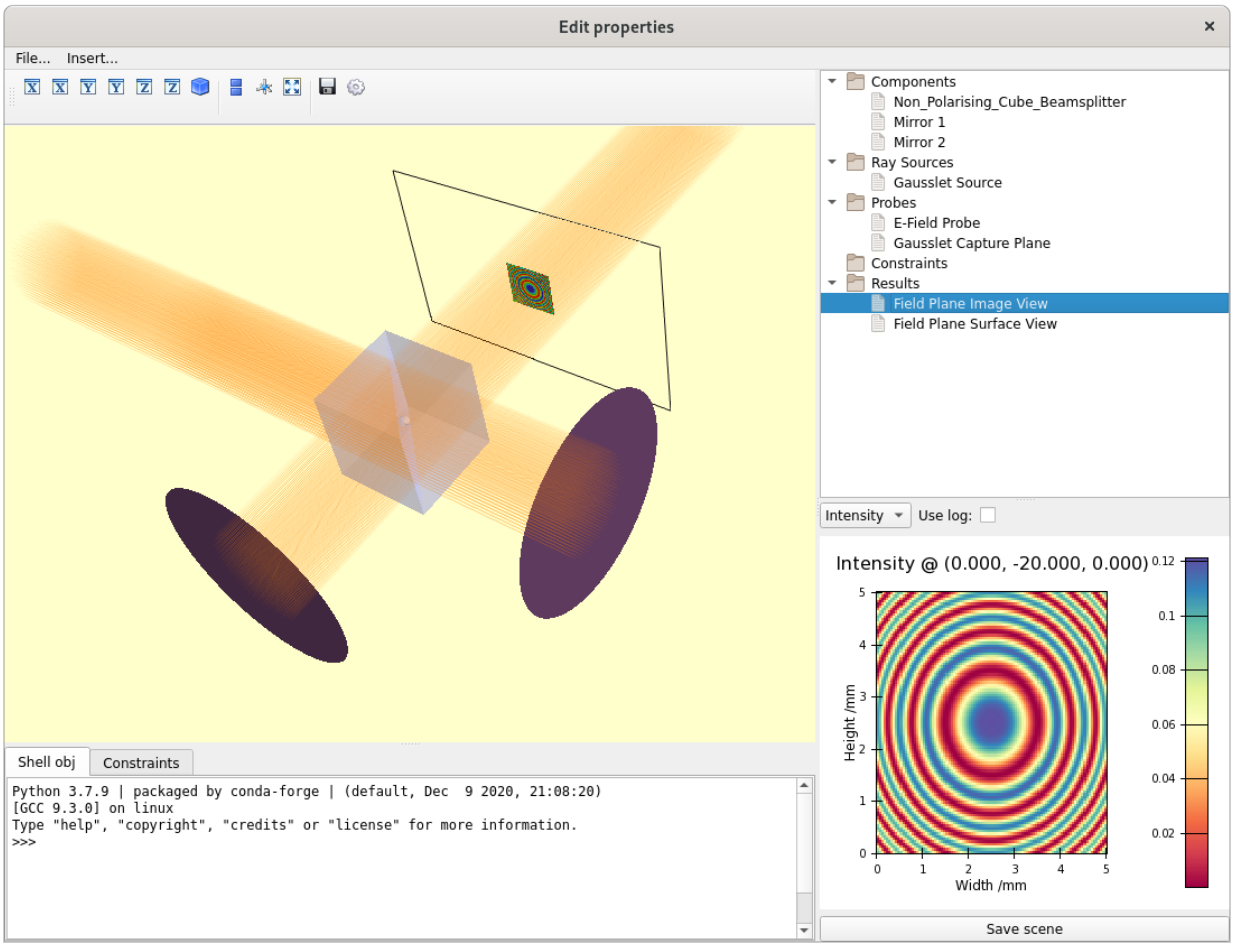
The model should re-trace automatically on exiting the context-manager.

CHAPTER

FIVE

EXPLORING THE GUI

Raypier is based on the Traits/TraitsUI framework. The Traits library provides a notification framework and improved type-declaration. TraitsUI is a GUI framework that extends Traits to make developing custom GUIs fast and easy. The UI for *any* traited object can be invoked with a call to my_object.configure_traits(). Typically, I recommend you define your base optical model in a script where an instance of RayTraceModel is created, then launch ui GUI with a call to the model's configure_traits() method. Lets see this in action. Running the michelson_interferometer_example.py script, you should see the following window

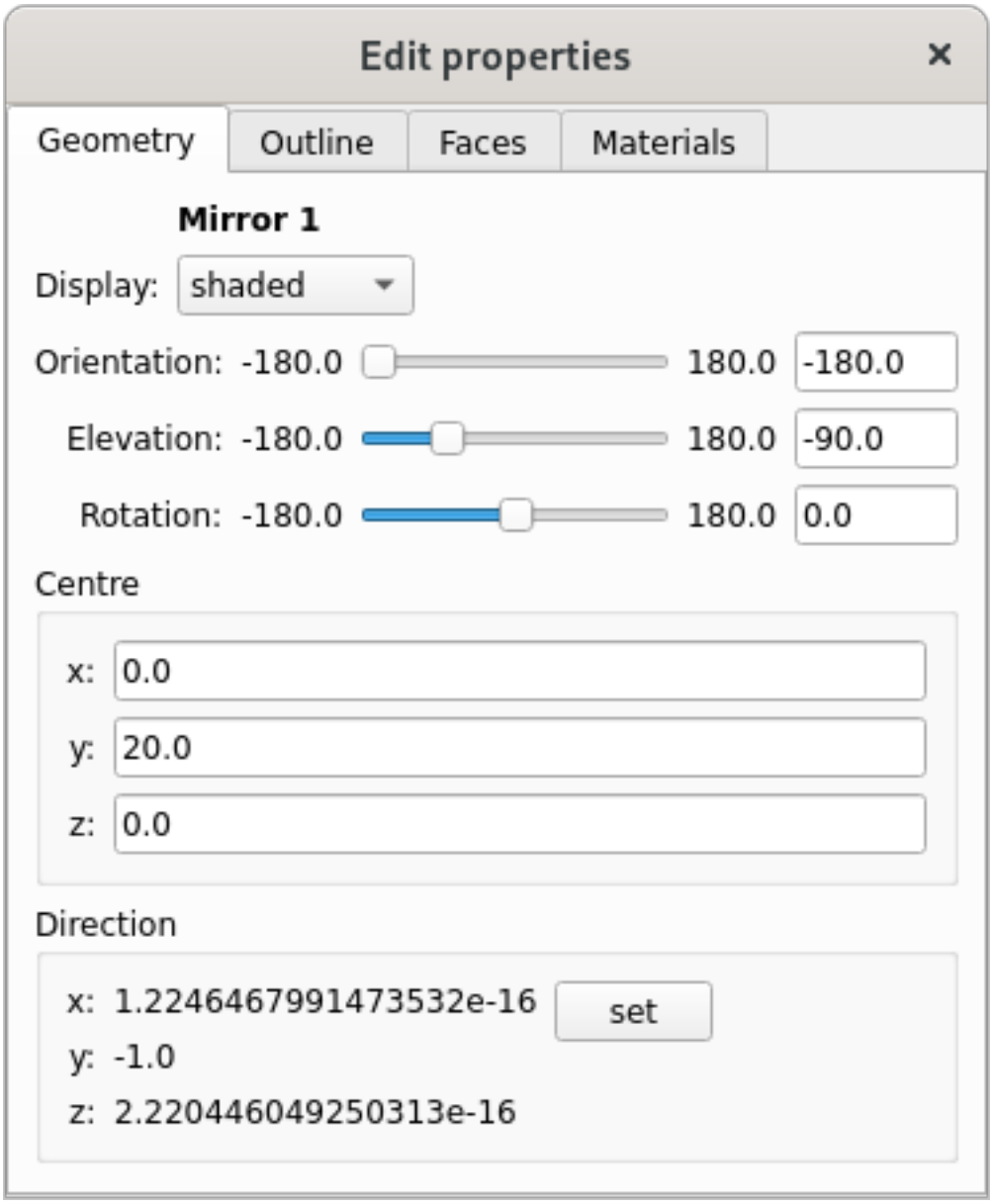


The main window is divided into 4 main regions. At the upper left, we have the main 3D model view. <left-click>-drag to rotate the view, <middle-click>-drag to pan and <right-click>-drag to zoom. You can also zoom using the mouse-wheel.

The upper right panel shows the model tree, with sub-groups for each of the model components types. Selecting any one model component object displays the properties-view for that object in the lower-right panel.

Double-clicking any object in the tree-view will open another properties-view window. This is useful for when you want to have the properties of multiple objects open at once (N.B. this doesn't yet work for the *image view* and *surface view* objects. I plan to lift this restriction later on).

For example, here's the properties view for the *Mirror 1* object:



All property windows edit their objects "live". The model will re-calculate itself automatically whenever anything changes, so you can explore how the ray-tracing results change as you move/adjust objects.

The bottom-left region contains two tabs: a python console and the Constraints view. There is no constraints in the example shown.

Within scope of the python terminal window, the model object is bound to the name *self*. While the rest of the model components can be accessed from this reference, another way to access objects is to simply drag and drop items from the tree-view into the terminal window. These will be added to the local namespace of the terminal using either the

name of the object (as given in the tree), or, if the name is not a valid python name (e.g. it contains spaces) it will simply be called "dragged".

At the top of the main window there is a menu-bar with two menus. The *File...* menu lets you save/load models in YAML format. However, I strongly advise against using this as a means of persisting models. The internals of Raypier are under active development and I don't give any guarantees that the YAML format is stable. I prefer to create my models as python scripts. This way, if the API changes, one can update the code to restore the model function. Whereas, if a class-name or API changes, a YAML file will simply refuse to load.

Model components can be added from the "Insert..." menu. The delete or rename components, use the context-sensitive menu in the Tree View (<right-click> on a tree item).

CHAPTER

SIX

JUPYTER INTEGRATION

Sometimes the live GUI isn't what you want. Sometimes you want to manipulate or analyse a model in a jupyter notebooks environment. If you want to display the model view in a notebook, you simply call the *ipython_view()* method of the model object to show the view. The arguments *width* and *height* specify the size of the displayed image. The optional argument *view* is a dictionary describing the view camera parameters.

class raypier.tracer.**RayTraceModel**

raypier.tracer.RayTraceModel.**ipython_view**(*width: int*, *height: int*, *view={}*) → view dict

Renders the 3D view of the model as a bitmap to display in a jupyter notebook. The image is displayed along with a set of ipywidget buttons to allow the user to rotate, pan and zoom the view.

The method returns a *dict* containing the view pan/zoom/rotation state. This dict can then be passed to subsequent calls to *ipython_view()* as the *view* keyword argument, to create another view with similar state.

An example of jupyter notebook integration can be found here: Jupyter Notebook Example The original .ipynb document is included in the examples folder.

Note, the view-control buttons are not present in the html-output of the link above. You need to run the notebook in a jupyter notebook server to get the widgets.

CHAPTER

SEVEN

RAYCOLLECTION AND RAY OBJECTS

The most important object in Raypier is the RayCollection. This is (as you might guess from the name) a 1D array of Ray objects.

The Ray object (*raypier.core.ctracer.Ray*) represents a single ray, and wraps an underlying C-structure. The single Ray object exists as a convenience for python scripting. The actual ray-tracing operation operates only RayCollection objects.

`Ray` objects have the following attributes:

- origin - A 3-vector (tuple) of floats giving the start-point for the ray
- direction - A 3-vector giving the direction of the ray. This is always normalised to unit length
- E_vector - A 3-vector defining the polarisation-axes. This vector is unit-length and orthogonal to the *direction* vector
- normal - A 3-vector giving the unit-length surface normal of the face from which the ray originated. May be undefined for rays which have not originated from a face intersection
- refractive_index - a complex value giving the refractive index of the material through which the ray has propagated
- E1_amp - the complex electric field amplitude for polarisations parallel to the E_vector axis
- E2_amp - the complex electric field amplitude for the polarisation orthogonal to the E_vector axis
- length - the geometric length of the ray from it's start-point to its termination at an intersecting face (or may be set to the *max length* of the ray, it no intersection has occured
- phase - An additional phase-factor that may be introduced by face interactions. Currently, only used to hold the "grating phase" arising from diffraction-grating surfaces.
- accumulated_path - the total *optical* path length accumulated from the parent ray plus (parent length * real part of refractive index)
- wavelength_idx - a index into the wavelength list (held by both the RayCollection and/or source object)
- parent_idx - the index of the parent-ray in the parent RayCollection object
- end_face_idx - the index into the Global Face List of the face at which the ray terminates (i.e. intersects). The Global Face List can be accessed on the *all_faces* atrtibure of the `RayTraceModel` object.
- ray_type_idx - a bitfield indicating is the ray is a reflected or transmitted ray. Other ray-types may be defined in future

Rays have some additional read-only properties defined:

- power - the sum of squares of the E1_amp and E2_amp components
- amplitude - the square-root of the power

- jones_vector - Returns a 2-tuple (alpha, beta) representing the Jones Vector describing the polarisation state of the ray. See _https://en.wikipedia.org/wiki/Jones_calculus#Jones_vector
- E_left - Returns the complex electric field amplitude for the left circular polarisation state
- E_right - Returns the complex electric field amplitude for the right circular polarisation state
- ellipticity - Returns the ratio of the power in the right-hand polarisation state to the left-hand polarisation state I.e. A value of +1 indicates 100% right-circular polarisation, 0 indicates linear polarisation, -1 indicates 100% left polarisation.
- major_minor_axes - Returns a 2-tuple of unit-length vectors describing the major and minor polarisation axes

RayCollection objects have substantially the same attributes/properties as the Ray object, except that each property returns a numpy array containing the values for all rays in the collection.

RayCollection objects are iterable (yielding single Rays) and subscriptable.

7.1 Creating RayCollections

If you need to create a RayCollection with a large number of rays (say, if you are writing your own Source class), the easiest method is to create a numpy array with the equivalent numpy dtype:

```
>>> from raypier.api import ray_dtype, RayCollection
>>> print(ray_dtype)
[('origin', '<f8', (3,)),
 ('direction', '<f8', (3,)),
 ('normal', '<f8', (3,)),
 ('E_vector', '<f8', (3,)),
 ('refractive_index', '<c16'),
 ('E1_amp', '<c16'),
 ('E2_amp', '<c16'),
 ('length', '<f8'),
 ('phase', '<f8'),
 ('accumulated_path', '<f8'),
 ('wavelength_idx', '<u4'),
 ('parent_idx', '<u4'),
 ('end_face_idx', '<u4'),
 ('ray_type_id', '<u4')]
```

Once you've created a numpy array with this dtype, you populate its fields as required. You can then create a RayCollection instance from this array using :py:meth:`RayCollection.from_array' classmethod. E.g.:

```
>>> import numpy
>>> my_rays = numpy.zeros(500, ray_dtype)
>>> my_rays['direction'] = numpy.array([[0,1,1]],'d')
<... assign other members as necessary ...>
>>> rc = RayCollection.from_array(my_rays)
```

Note, data is always copied to and from RayCollections. The reason why we don't use memory views is that RayCollections have a dynamic size and can grow in size by re-allocation of their memory. Numpy array, by contrast are static in size.

Likewise, we can convert a RayCollection to a numpy array using its `RayCollection.copy_as_array()` method.:

```
>>> arr = rc.copy_as_array()
```

CHAPTER

EIGHT

RAY SOURCES

Ray sources generate the input rays for the model. The Ray-source objects also hold the results of the trace, in the `traced_rays` attribute, as a list of `RayCollection` objects. Each item in this list represents one "generation" of rays.

Ray source classes are subclasses of *raypier.sources.BaseRaySource*. Besides generating the `input_rays` (a `RayCollection` instance) attribute as the input to the model, and holding the trace results in the `traced_rays` member, the source objects also control the visualisation of the rays. The following visualisation-related attributes are available.

class raypier.sources.**BaseRaySource**

display: enum("pipes", "wires", "none")
Indicates how the rays should be drawn.

opacity: float
Sets the opacity for the rays where 0.0 is fully transparent and 1.0 is fully opaque.

wavelength_list
An numpy.ndarray listing all the wavelengths used generated by this ray source.

max_ray_len: float
Sets the maximum length of the rays.

scale_factor: float
Adjusts the scaling for the rays and related glyphs such as normals-arrows.

show_start: bool
If True, a sphere glyph is displayed at the source origin.

show_normals: bool
If True, the normal-vectors at each ray intersection will be drawn.

class raypier.sources.**SingleRaySource**(*raypier.sources.BaseRaySource*)

8.1 Ray Field Sources

A further sub-catagory of ray-sources are Ray Field Sources, being subclasses of `RayFieldSource`. In addition to generating the input_rays, Ray Field Source objects also generate the `neighbours` attribute. This is a (N,6) shaped array of ints which gives the index of each rays immediate neighbour.

The *neighbours* data for each RayCollection object is derived from it's parent RayCollection. They are evaluated lazily so if neighbours are not required they don't incure any penatly. Having neighbours present means that a RayCollection object can be summed as a set of Gaussian modes to generate the E-field.

Ray Field Sources were a precursor to the Gausslets implementation and are now somewhat redundant. If you want to include diffraction-effects, your starting point should be a Gausslet Source (see below). Ray Field Sources do not include the effects of diffraction in the propagation of their parent rays. Ray Fields can thus give you the point-spread function. One benefit of Ray Fields over Gausslet Sources would be that performance: without the need to trace parabasal rays, we expect ray-tracing RayFields to be faster than the full Gausslet implementation.

CHAPTER

NINE

THE GENERAL OPTIC FRAMEWORK

The General Optic framework provides the most flexible means of defining optical components. The GeneralLens class is an optic which is composed of:

- A shape (from `raypier.shapes`)
- A list of surfaces (from `raypier.faces`)
- A list of materials (from `raypier.materials`)

These are high-level objects which aim to streamline the process of configuring a custom optical element. The general optic framework is appropriate for optical elements which fit the "planar" model of a outline defined in the XY-plane with a surface sag defined in z as a function of (x,y). Many optics are not a good fit for this description (e.g. prisms) and hence other sub-modules provide these components.

9.1 Shapes

Shapes define the 2D outline of the optic in its local XY-plane (remember that every component has its own local coordinate system. The optic can have any position and orientation in the global frame of reference).

There are four shape primitives, at the time of writing:

- `RectangleShape`
- `CircleShape`
- `PolygonShape`
- `HexagonShape`

There is one more I have yet to implement, being EllipseShape.

Shapes support boolean operation so that they can be combined into more complex shapes. For example, to make a rectangular lens with a hole in it. You simply XOR a RectangleShape and a CircleShape, as follows:

```
from raypier.api import CircleShape, RectangleShape

my_shape = RectangleShape(width=30,height=25) ^ CircleShape(radius=3.0)
```

Likewise, shapes support AND and OR, giving you the intersection and the union of the two shapes respectively. Internally, NOT is also available but annoyingly, the VTK boolean operations for implicit functions don't seem to offer a way to invert them, so I couldn't get the "not" visualisation to work.

9.2 Surfaces

The surfaces represent the geometry of the faces of the GeneralLens. While a simple singlet lens will have two surfaces, a doublet will have 3. In fact, any number of surfaces can be added.

I wanted to call the *surfaces* list of the GeneralLens "faces" but faces is already over-used as an attribute name.

There are 7 core face types supported by the General Optic framework:

- Planar
- Spherical
- Cylinderical
- Conic (a conic surface of revolution)
- Aspheric
- Axicon
- Distortion

These all have a "mirror" boolean trait. If this is true, the face is considered 100% reflective. Otherwise, the reflection and transmission characteristics will be derived from the dielectric material coefficients on either side of the surface.

The DistortionFace object is a special type of Face that wraps another "base face" with some sort of geometric distortion function (for example, a Zernike polynomial function). The range of available distortion functions can be found in the `raypier.distortions` module.

9.3 Materials

The materials list gives the optical properties of the dielectric on either side of each surface. For `n` surfaces, we need `n-1` materials.

Materials define their dispersion functions either as constant values, given as refractive index and absorption-coefficient, or as functions obtained from a database of optical properties (taken from _https://refractiveindex.info/). In the later case, you can specify the dielectric material by name e.g. "N-BK7".

Put all this together, here's in example:

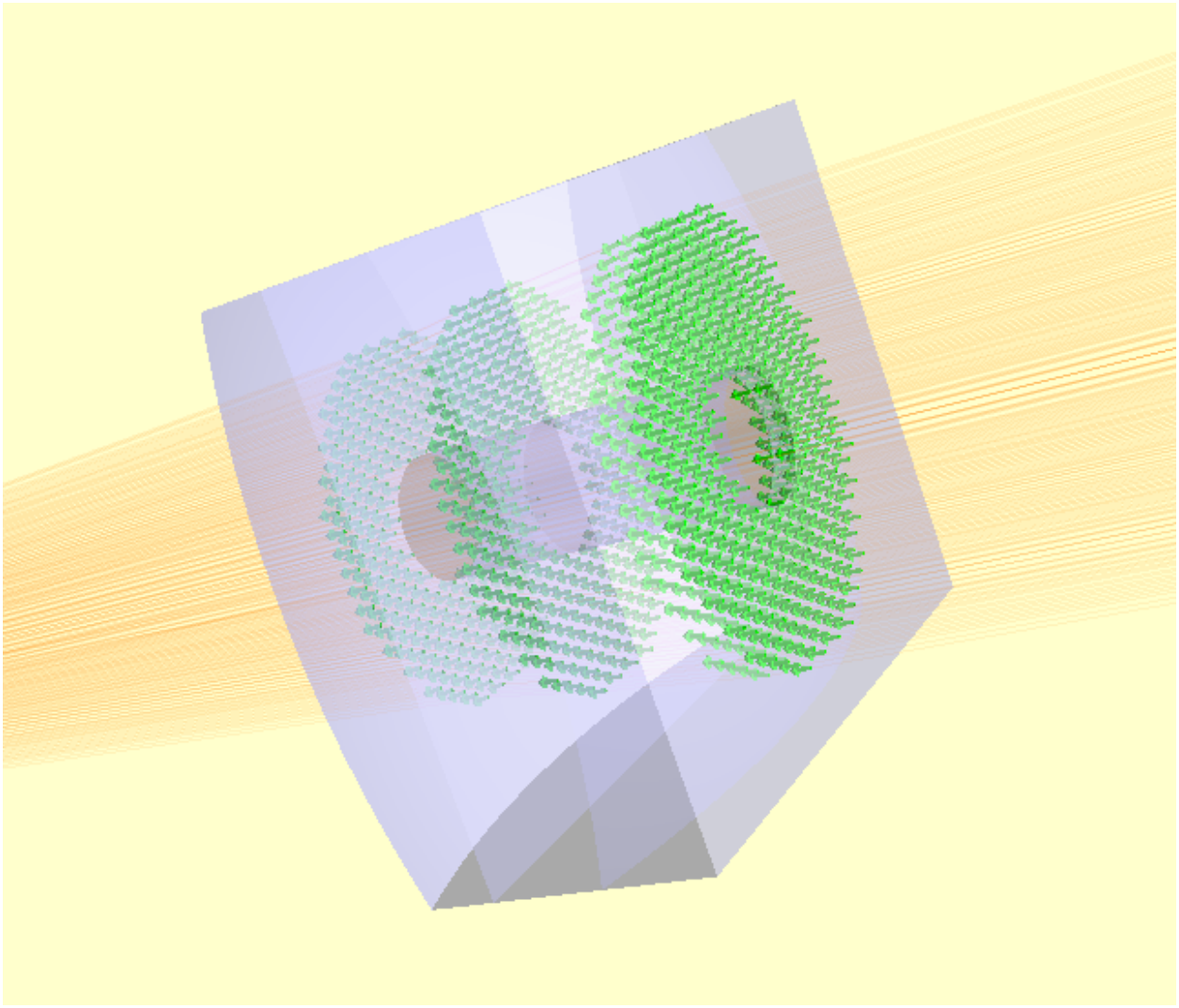
```
s1 = RectangleShape(width=30,height=25) ^ CircleShape(radius=3.0)

f1 = SphericalFace(z_height=8.0, curvature=50.0)
m1 = OpticalMaterial(from_database=False, refractive_index=1.5)
f2 = PlanarFace(z_height=1.0)
m2 = OpticalMaterial(from_database=False, refractive_index=1.1)
f3 = SphericalFace(z_height=-8.0, curvature=-50.0)
m3 = OpticalMaterial(from_database=False, refractive_index=1.6)
f4 = PlanarFace(z_height=-9.0, invert=False)

faces = [f1,f2,f3,f4]
mats = [m1, m2, m3]

lens = GeneralLens(centre=(0,0,50),
                   shape=s1,
                   surfaces=faces,
                   materials=mats)
```

(continues on next page)

(continued from previous page)

```
src = HexagonalRayFieldSource(gauss_width=5.0,
                              display="wires",
                              opacity=0.1,
                              show_normals=True)


model = RayTraceModel(optics=[lens], sources=[src])

model.configure_traits()
```

Gives us the following result:



CHAPTER

TEN

OTHER OPTIC TYPES

For optic-types which don't fit the General Optics framework, we have support for each type in their own submodules. There are also a few submodules for optics which predate the General Optics framework.

10.1 Apertures

10.2 Prisms

10.3 Waveplates

10.4 Beamsplitters

10.5 Diffraction Gratings

I intend to migrate the DiffractionGratingFace to the GeneralOptic framework.

10.6 Corner-cube retro-reflectors

10.7 Off-axis ellisoids

CHAPTER

ELEVEN

GAUSSIAN BEAMLET PROPAGATION

Gaussian beams are a solution to the paraxial wave equation. It turns out that the propagation of such modes can be represented by a chief ray and a number of skew marginal rays which define the 1/e^2 edge (and divergence) of the beam. It has been shown that such the propagation of such modes obeys the rules of geometric ray tracing (i.e. Snell's law).

An arbirary wavefront can be decomposed into a set of paraxial Gaussian beamlets (called "Gausslets" henceforth) and propagated through an optical system using raytracing. This method provides a convenient route to extend a geometric ray-tracing algorithm to obtain a physical-optics modelling solution. The method has been successfully employed in a number of commercial optical modelling packages, such as CODE-V, FRED and ASAP. Given a set of Gausslets, the vector (E) field can be evaluated at any position by summing the fields from each Gausslet.

The method is accurate provided some restrictions are observed:

- The Gausslets must remain paraxial (i.e. more-or-less collimated).
- Individual Gausslets should interact with refracting or reflecting surfaces over a sufficiently small area such that the surface may be considered locally parabolic.

Typically, if either of these restrictions are violated, the solution is to interrupt the ray-tracing operation to evaluate the field, then decompose the field into a new set of Gausslets. In cases where the incident Gausslet is sampling too large an area of the surface for it to be considered parabolic, the field is decomposed into a grid of smaller Gausslets. At the other extreme, Gausslets incident on a small aperture may be decomposed into a set of spatially overlapped Gausslets with range of directions. Such an angle decomposition is sometimes known as a "Gabor Decomposition".

11.1 Raypier Gausslet Implementation

Raypier support Gausslet tracing through a GaussletCollection data-structure. Gausslets are an extension of the Ray data structure. In fact, the dtype for a Gausslet object looks like:

```
para_dtype = [('origin', '<f8', (3,)), ('direction', '<f8', (3,)), ('normal', '<f8',␣
↪(3,)), ('length', '<f8')]

gausslet_dtype = [('base_ray', ray_dtype), ('para_rays', para_dtype, (6,))]
```

I.e. we define a dtype for the parabasal rays which contains only the geometric information (origin, direction, length etc.) and omits any E-field polarisation or amplitude information. The gausslet is then composed of one *base_ray* (with regular ray_dtype) and 6 parabasal rays (with para_dtype).

Gausslets have their own set of predefined source-objects, found in `raypier.gausslet_sources`.

11.2 Evaluating the E-field

Algorithms for evaluation of E-fields are provided in *raypier.core.fields*

The nice thing about Gausslet ray-tracing is that you can evaluate the E-field at any point in your model. For script-based analysis, you can give any GaussletCollection object (obtained from a source-object after a tracing operation) to the *raypier.core.fields.eval_Efield_from_gausslets()* function.

11.3 Beam Decomposition

When a Beam-decomposition object intercepts a ray during the tracing operation, instead of immediately generating child rays as most other *Traceable* objects do, the decomposition objects simply store the intercepted ray. At the completion of tracing of the current ray generation (i.e. GaussletCollection), any decomposition-objects which have received one or more rays then perform their decomposition-algorithm to generate a new set of rays to be added to the other rays created in the last generation. The new rays created by the decomposition process will, in general, not join originate from the end-point of the input-rays.

High level *Optics* objects for beam decomposition are provided here.

class raypier.gausslets.**PositionDecompositionPlane**(*BaseDecompositionPlane*)
Defines a plane at which position-decomposition will be beformed.

radius
Sets the radius used for capturing incoming rays. Rays outside of this will "miss"

curvature
An approximate radius-of-curvature to the beam focus. This is used to improve the phase-unwrapping of the wavefront. The default is zero, which means a plane-wave is assumed. Negative values imply a focus behind the decomposition plane (i.e. on the opposite side to the plane direction vector).

resolution
Sets the resampling density of the decomposition, in terms of the number of new rays per *radius* extent.

blending
Sets the blending values for the new rays. The new rays will have Gaussian 1/e**2 intensity widths equal to *spacing*/*blending*, where the *spacing* value is *radius*/*resolution*.

class raypier.gausslets.**AngleDecomposition**(*BaseDecompositionPlane*)
Defines a plane at which Gabor (angle)-decomposition is to be performed.

sample_spacing
Sets the sample-spacing at the decomposition plane, in microns.

width
A value in the range 1->512 to set the number of samples along the width of the sample-plane.

height
A value in the range 1->512 to set the number of samples along the height of the sample-plane.

mask
A 2d array with shape matching the (width, height) and dtype numpy.float64 . The array values should be in the range 0.0 -> 1.0. This will be used to mask the input E-field.

max_angle
Limits the angular divergence of the outgoing rays.

The low-level beam-decomposition algorithms are found in this module. Two types of decomposition are available: position-decomposition and angle-decomposition. Use the former when the Gausslets are found to be too wide at a particular surface in the optical path to re-sample the beam onto a set of more compact Gausslets. The later is used to

simulate the effect of apertures much smaller than the Gausslet widths, such that each Gausslet can be treated like a plane-wave and the field-distribution found using a 2d Fourier transform.

CHAPTER

TWELVE

DISTORTIONS

The `raypier.distortions` module contains objects representing distortions of a given face. The Distortion objects are intended to be used with the `raypier.faces.DistortionFace` class (part of the General Optic framework). Fundamentally, any 2D function can be implemented as a Distortion. At present, on a single type is implemented. I intend to implement a general Zernike polynomial distortion class. Other distortion functions are easy to add.

An example of their usage:

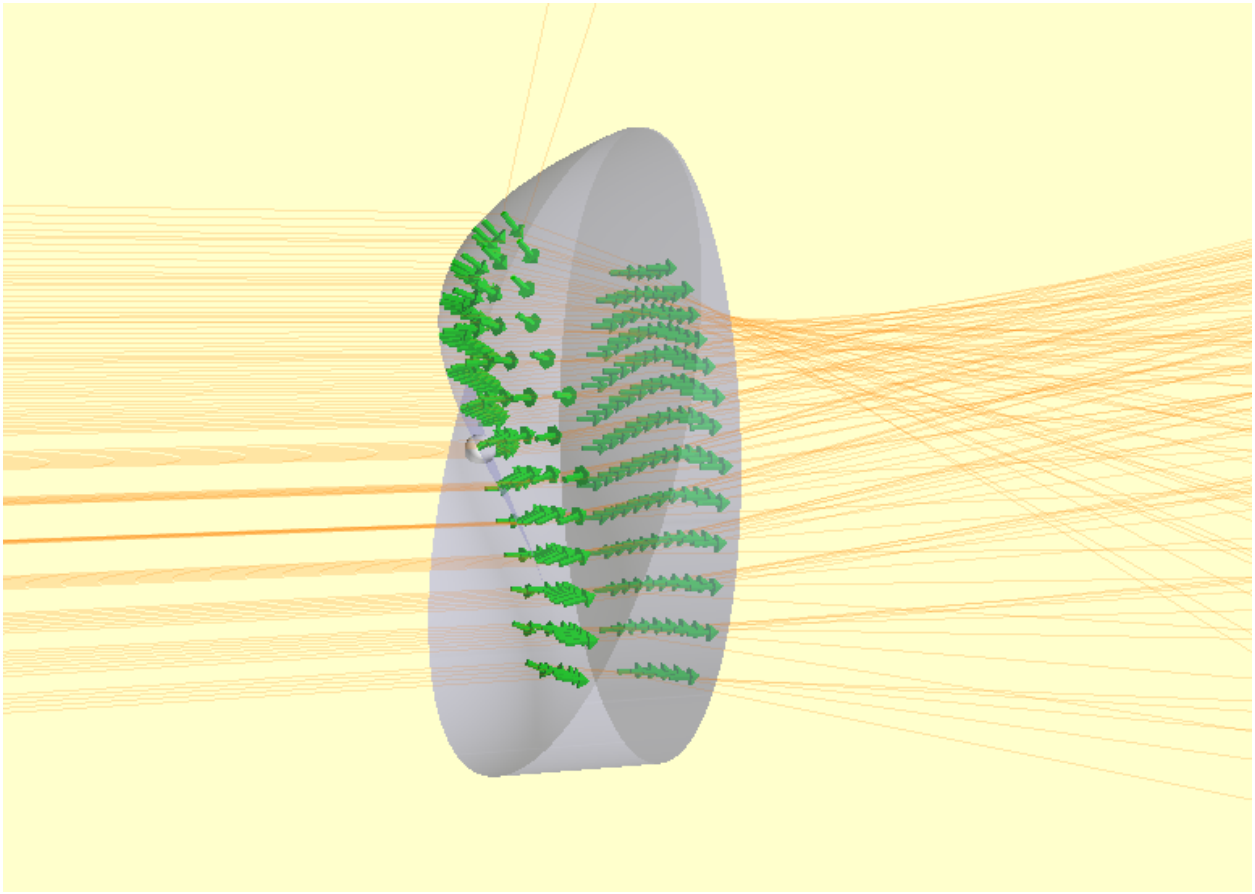
```
from raypier.tracer import RayTraceModel
from raypier.shapes import CircleShape
from raypier.faces import DistortionFace, PlanarFace, SphericalFace
from raypier.general_optic import GeneralLens
from raypier.materials import OpticalMaterial
from raypier.distortions import SimpleTestZernikeJ7, NullDistortion
from raypier.gausslet_sources import CollimatedGaussletSource
from raypier.fields import EFieldPlane
from raypier.probes import GaussletCapturePlane
from raypier.intensity_image import IntensityImageView
from raypier.intensity_surface import IntensitySurface

shape = CircleShape(radius=10.0)
f1 = SphericalFace(z_height=0.0, curvature=-25.0)
f2 = PlanarFace(z_height=5.0)

dist = SimpleTestZernikeJ7(unit_radius=10.0, amplitude=0.01)
#dist = NullDistortion()
f1 = DistortionFace(base_face=f1, distortion=dist)

mat = OpticalMaterial(glass_name="N-BK7")
lens = GeneralLens(shape=shape, surfaces=[f1,f2], materials=[mat])

src = CollimatedGaussletSource(radius=8.0, resolution=6,
                               origin=(0,0,-15), direction=(0,0,1),
                               display="wires", opacity=0.2, show_normals=True)
src.max_ray_len=50.0


cap = GaussletCapturePlane(centre = (0,0,50),
                           direction= (0,0,1),
                           width=20,
                           height=20)
```

(continues on next page)

(continued from previous page)

```
field = EFieldPlane(detector=cap,
                    align_detector=True,
                    size=100,
                    width=1,
                    height=1)

img = IntensityImageView(field_probe=field)
surf = IntensitySurface(field_probe=field)


model = RayTraceModel(optics=[lens], sources=[src], probes=[field,cap],
                      results=[img,surf])
model.configure_traits()
```

This example shows a high-amplitude distortion, for illustrative purposes.



During the ray-tracing operation, the intersections with distorted faces are found using an iterative algorithm similar to Newton- Ralphson. Typically, the intersection is found with 2 to 3 calls to the intercept-method of the underlying face. Distortions are expected to be small deviations from the underlying face (maybe no more than a few wavelengths at most). If you make the amplitude of the distortion large, the under of iterations to converge will increase and the ray-tracing hit take a performance hit. For very large distortions, the intercept my fail altogether.

One could, in principle, wrap multiple DistortionFaces over other DistortionFaces. However, I would expect the performance penalty to be quite severe. In this case, A better plan would be to implement a specialised DistortionList object which can sum the distortion-values from a list of input Distortions. On my todo list . . .

In python scripting, one can simply evaluate any Distortion object given some x- and y-coordinates as numpy arrays. This is useful for testing. For example:

```
from raypier.distortions import SimpleTestZernikeJ7
import numpy

dist = SimpleTestZernikeJ7(unit_radius=10.0, amplitude=0,1)

x=y=nmupy.linspace(-10,10,500)
X,Y = numpy.meshgrid(x,y)

Z = dist.z_offset(X.ravel(),Y.ravel())
Z.shape = X.shape #restore the 2D shape of the Z array
```

Distortions have an additional method, `Distortion.z_offset_and_gradient()`. This returns a array of shape (N,3) where the input X and Y arrays have length N. The first two columns of this array contain the gradient of the distortion, dZ/dX and dZ/dY respectively. The third column simply contains Z. Returning both Z and it's gradient turns out to be useful at the C-level during tracing. I.e.:

```
grad = dist.z_offset_and_gradient(X.ravel(), Y.ravel()).reshape(X.shape[0], X.
↪shape[1],3)
dZdX = grad[...,0]
dZdY = grad[...,1]
Z = grad[...,2]
```

12.1 Zernike Polymonial Distortions

More general distortions can be applied using the `raypier.distortions.ZernikeSeries` class.

As previously, instances of this object are passed to a `raypier.faces.DistortionFace` , along with the base-surface to which the distortion is to be applied.

An example of the this class in action can be seen here:

```


from raypier.tracer import RayTraceModel
from raypier.shapes import CircleShape
from raypier.faces import DistortionFace, PlanarFace, SphericalFace
from raypier.general_optic import GeneralLens
from raypier.materials import OpticalMaterial
from raypier.distortions import SimpleTestZernikeJ7, NullDistortion, ZernikeSeries
from raypier.gausslet_sources import CollimatedGaussletSource
from raypier.fields import EFieldPlane
from raypier.probes import GaussletCapturePlane
from raypier.intensity_image import IntensityImageView
from raypier.intensity_surface import IntensitySurface
from raypier.api import Constraint

from traits.api import Range, observe
from traitsui.api import View, Item, VGroup


shape = CircleShape(radius=10.0)

#f1 = SphericalFace(z_height=0.0, curvature=-25.0)
```

(continues on next page)

(continued from previous page)

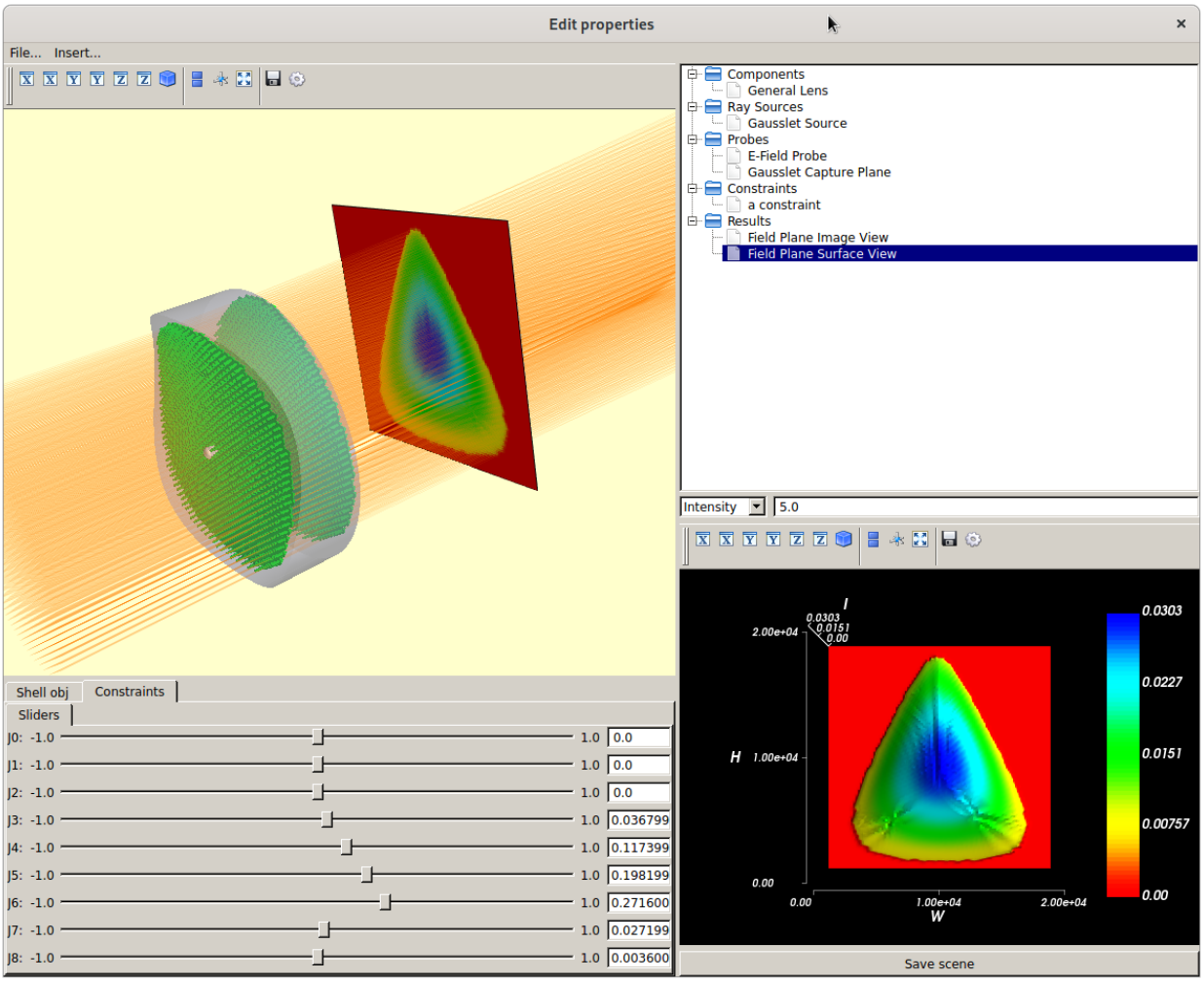
```
f1 = PlanarFace(z_height=0.0)
f2 = PlanarFace(z_height=5.0)

dist = ZernikeSeries(unit_radius=10.0, coefficients=[(i,0.0) for i in range(12)])
f1 = DistortionFace(base_face=f1, distortion=dist)


class Sliders(Constraint):
    """Make a Constrain object just to give us a more convenient UI for adjusting␣
↪Zernike coefficients.
    """
    J0 = Range(-1.0,1.0,0.0)
    J1 = Range(-1.0,1.0,0.0)
    J2 = Range(-1.0,1.0,0.0)
    J3 = Range(-1.0,1.0,0.0)
    J4 = Range(-1.0,1.0,0.0)
    J5 = Range(-1.0,1.0,0.0)
    J6 = Range(-1.0,1.0,0.0)
    J7 = Range(-1.0,1.0,0.0)
    J8 = Range(-1.0,1.0,0.0)

    traits_view = View(VGroup(
            Item("J0", style="custom"),
            Item("J1", style="custom"),
            Item("J2", style="custom"),
            Item("J3", style="custom"),
            Item("J4", style="custom"),
            Item("J5", style="custom"),
            Item("J6", style="custom"),
            Item("J7", style="custom"),
            Item("J8", style="custom"),
        ),
        resizable=True)

    def _anytrait_changed(self):
        dist.coefficients = list(enumerate([self.J0, self.J1, self.J2, self.J3, self.
↪J4,
                                            self.J5, self.J6, self.J7, self.J8]))


mat = OpticalMaterial(glass_name="N-BK7")
lens = GeneralLens(shape=shape, surfaces=[f1,f2], materials=[mat])

src = CollimatedGaussletSource(radius=9.0, resolution=20,
                               origin=(0,0,-15), direction=(0,0,1),
                               display="wires", opacity=0.02,
                               wavelength=1.0,
                               beam_waist=10.0,
                               show_normals=True)
src.max_ray_len=50.0


cap = GaussletCapturePlane(centre = (0,0,50),
                           direction= (0,0,1),
                           width=20,
                           height=20)
```

(continues on next page)

(continued from previous page)

```
field = EFieldPlane(centre=(0,0,30),
                    detector=cap,
                    align_detector=True,
                    size=100,
                    width=20,
                    height=20)

img = IntensityImageView(field_probe=field)
surf = IntensitySurface(field_probe=field)


model = RayTraceModel(optics=[lens], sources=[src], probes=[field,cap],
                      results=[img,surf], constraints=[Sliders()])
model.configure_traits()
```

Here's what the model looks like in the UI.



This example also demonstrates the use of a Constraints object to provide some UI controls for easier adjustment of the relevant model parameters.

CHAPTER

THIRTEEN

CYTHON BRANCH: HOW TO ADD NEW OPTICS

When working with the cython raypier branch, be sure you have the latest Cython version installed (as of writing, v0.12.1 was the latest).

13.1 Creating a new Face

Subclasses of raypier.core.ctracer.Face perform the mechanics of the ray-tracing operation. Most existing subclasses are defined in cfaces.pyx. To define a new face object, two "C" methods need to be defined:

cdef double intersect_c(vector_t p1, vector_t p2)

Computes the intersection of a ray with the surface, where p1 and p2 are two points defining the start and end points of an incoming ray, in the local coordinate system. The method returns a single floating-point value representing the distance along the ray where an intersection occurs. If no intersection is found, a value <= 0 may be returned (I tend to return zero, if no intersection occurs).

cdef vector_t compute_normal_c(vector_t p)

Computes the outward normal vector for the surface at the given point, p. These are also defined w.r.t. the local coordinate system.

Note, these two methods are 'cdef-methods' and hence not callable directly from python. Be sure to type all the variables you use, to be sure of good performance. Face objects also have python-callable methods compute_normal() and intersect(). These call the cdef methods internally. Don't bother trying to overload these from python as it won't work (and even if it did, performance would be horrible).

The vector_t structure is used to represent 3D points (i.e. it's a 3d vector), and has .x and .y and .z members. Outside the comfort of python/numpy, vector algebra is significantly less pleasant. The ctracer module defines a number of inline functions to (partially) simplify working with vector_t's. For vector arithmatic, you can use:

- addvv_(vector_t a, vector_t b) #add two vectors
- addvs_(vector_t a, double b) #adds a vector and a scalar
- multvv_(vector_t a, vector_t b) #multiplies two vectors
- multvs_(vector_t a, double b) #multiplies a vector and a scalar

  etc.

Similar function are defined for subtraction and element-wise division. The 'vv' function denote operations on two vectors, 'vs' denotes operations on a vector and a scalar. The trailing underscore is a convention I'm adopting to indicate that the method is a 'C'-method (i.e. not python callable).

Some of the other utility functions are:

cdef inline vector_t set_v(vector_t v, object O)

Takes a vector v and an 3-sequence O. Copies the values from the sequence into the vector and returns it (why does it need the v argument, though?).

cdef inline double sep_(vector_t p1, vector_t p2)

Computed the linear separation between two points

cdef inline double dotprod_(vector_t a, vector_t b)

Calculates the dot-product of the given vectors

cdef inline vector_t cross_(vector_t a, vector_t b)

Calculates the vector-product of it's arguments

cdef inline vector_t norm_(vector_t a)

Computes the normalised vector from its input (i.e. scales it's input to unit magnitude).

cdef inline double mag_(vector_t a)

Calculates the magnitude of it's input

cdef inline double mag_sq_(vector_t a)

Calculates the square of the magnitude of it's input

cdef inline vector_t invert_(vector_t v)

Inverts it's input

Most of the cdef functions and cdef-methods have companion def functions/methods which are callable from python and wrap the associated cdef operation. These are mostly used in testing. The python-callable versions incure the normal python interpreter overhead, and hence are not directly useful in the fast C-level tracing process.

The intersect_c and compute_normal_c cdef methods are all you need to make a new face. Obviously, you can add additional methods / attributes to implement the functionality required.

You can optionally give your custom Face class a params attribute (define it as a class-attribute) which should contain a list of attribute names which should be synchronised from the face owner to the face when a tracing operation is initiated.

13.2 Creating a new Traceable

Traceables (i.e. subclasses of raypier.bases.Traceable) are the basic unit of an optical model. Most of the functionality in the Traceable subclasses is to implement their VTK visual representation. The ray-tracing operation is handled by Face objects. Traceables own an instance of a ctracer.FaceList which turn has a list of Face objects (I choose FaceList *has a* list of faces, rather than FaceList *is a* list of faces, as it was simpler to implement). The FaceList contains the coordinate transform which maps betwee global coords and the local coords of the Traceable. Thus, all Faces belonging to a Traceable share a common transform.

To create a new Traceable, you subclass Traceable or some other more suitable subclass (transmitting optical components can derive from raypier.bases.Optic, which provides a complex refractive index). You should define a new _faces_default method which creates the FaceList for that object and populates it with the Faces appropriate to the object. Simple synchronisation between the Traceable and the Faces can be handled using the params Face class attribute described above. In most cases, more sophisticated synchronisation is required and can be handled using trait-notifications for all traits on which the Faces depends.

The physics of ray-scattering (i.e. the generation of new rays at the point of intersection) is handled by ctracer.InterfaceMaterial objects. InterfaceMaterial is an abstract base class. There are two concrete subclasses defined in the ctracer module: PECMaterial and DielectricMaterial. The former represents a perfect metal reflector. The later is a normal dielectric surface. Typically, an Optic (or Traceable subclass) will have an InterfaceMaterial trait. This

will be passed to it's faces in the _faces_default method (so all faces share the same InterfaceMaterial). However, this is not a requirement: faces can have independent materials, or share them.

13.3 Custom Interface Materials

InterfaceMaterial subclasses provide a cdef method:

```
cdef eval_child_ray_c(self, ray_t *old_ray,
                          unsigned int ray_idx,
                          vector_t point, vector_t normal,
                          RayCollection new_rays)
```

This is called for each ray intersection to create a new ray. The arguments are as follows:

> old_Ray - a pointer to the incoming ray_t structure ray_idx - the index of the incoming ray in it's RayCollection array point - the position, in global coords, of the intersection normal - the normal vector of the surface, at the point of intersection new_rays - the target RayCollection for new rays

This method should call new_rays.add_new_ray() to create as many new rays as necessary. Thus, multiple ray generation can occur at an intersection (as might be found for a diffracting interface material).

13.4 Cython Tips and Tricks

If you find performance is less than you expected, try running "cython -a yourfile.pyx" (replace yourfile.pyx with whatever .pyx file you're analysing, cfaces.pyx maybe). This produces a html-version of your file, with highlighting to show where the python API is being invoced. The less yellow the better (and red-highlights indicate real performance bottlenecks). This is a *very* cool feature of Cython.

Avoid cpdefs (i.e. methods with automatically created python wrappers), as extra overhead is incured to check for python overloading.

Creating and destroying python objects is expensive (it invokes the garbage collector / changes ref-counts etc.). However, read-only access to items in lists is fast.

Surprisingly, I can find no speed benefit in passing parameters by reference, compared to passing by values (for fixed-size types, at least).

CHAPTER

FOURTEEN

EXAMPLES

14.1 Simulating a Bessel Beam

A Pseudo-bessel beam can be created using an Axicon. Where the rays from opposing sides of the axicon interact, they create a region of high intensity along the axis of the axicon. The interested properties of this beam is that it does not diffract. Modelling this with Raypier is straightforward. We use a CollimatedGaussianBeamSource to create the input wavefront.

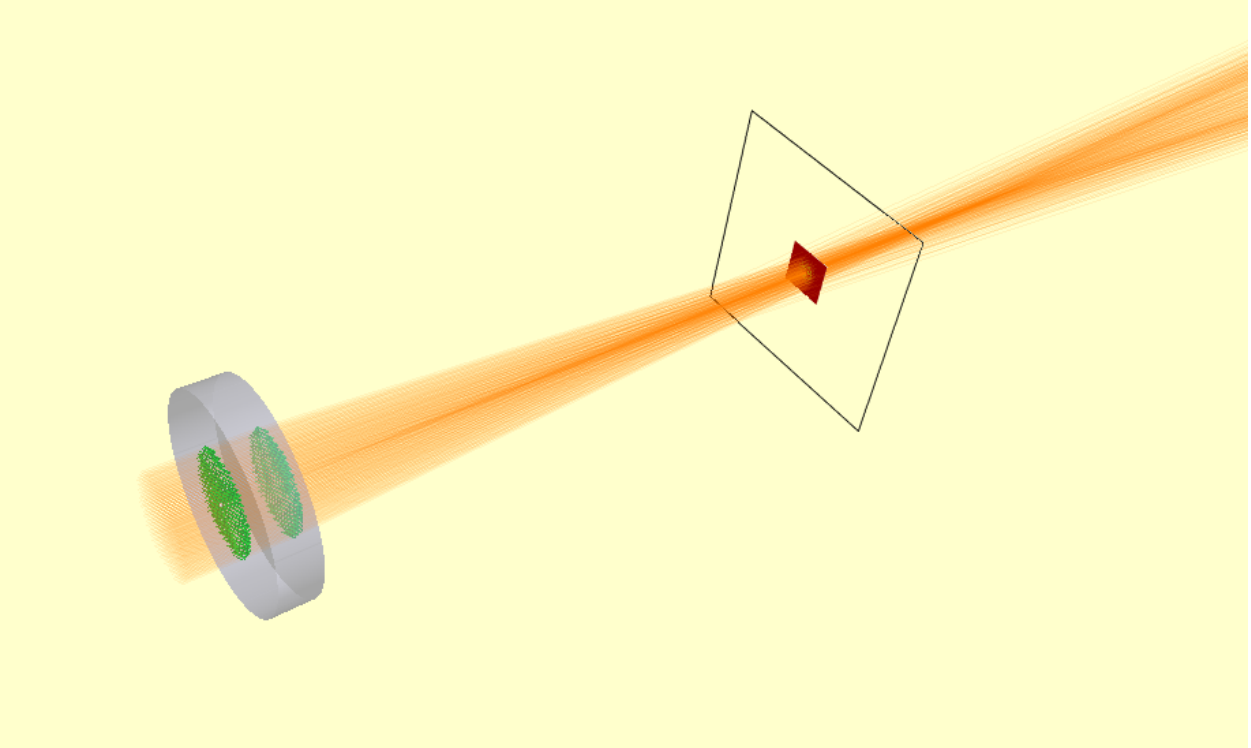
```
from raypier.api import GeneralLens, AxiconFace, PlanarFace, OpticalMaterial,␣
↪CircleShape,\
            RayTraceModel, CollimatedGaussletSource, EFieldPlane,␣
↪GaussletCapturePlane, IntensitySurface

from raypier.intensity_image import IntensityImageView


shape = CircleShape(radius=2.0)

face1 = PlanarFace(z_height=0.0)
face2 = AxiconFace(z_height=1.0, gradient=0.1)

mat = OpticalMaterial(glass_name="N-BK7")

axicon = GeneralLens(name = "My Axicon",
                     centre = (0,0,0),
                     direction=(0,0,1),
                     shape=shape,
                     surfaces=[face1,
                               face2],
                     materials=[mat])

src = CollimatedGaussletSource(origin=(0.001,0,-5.0),
                               direction=(0,0,1),
                               wavelength=0.5,
                               radius=1.0,
                               beam_waist=10.0,
                               resolution=10,
                               max_ray_len=50.0,
                               display='wires',
                               opacity=0.2
                               )
```

(continues on next page)

(continued from previous page)

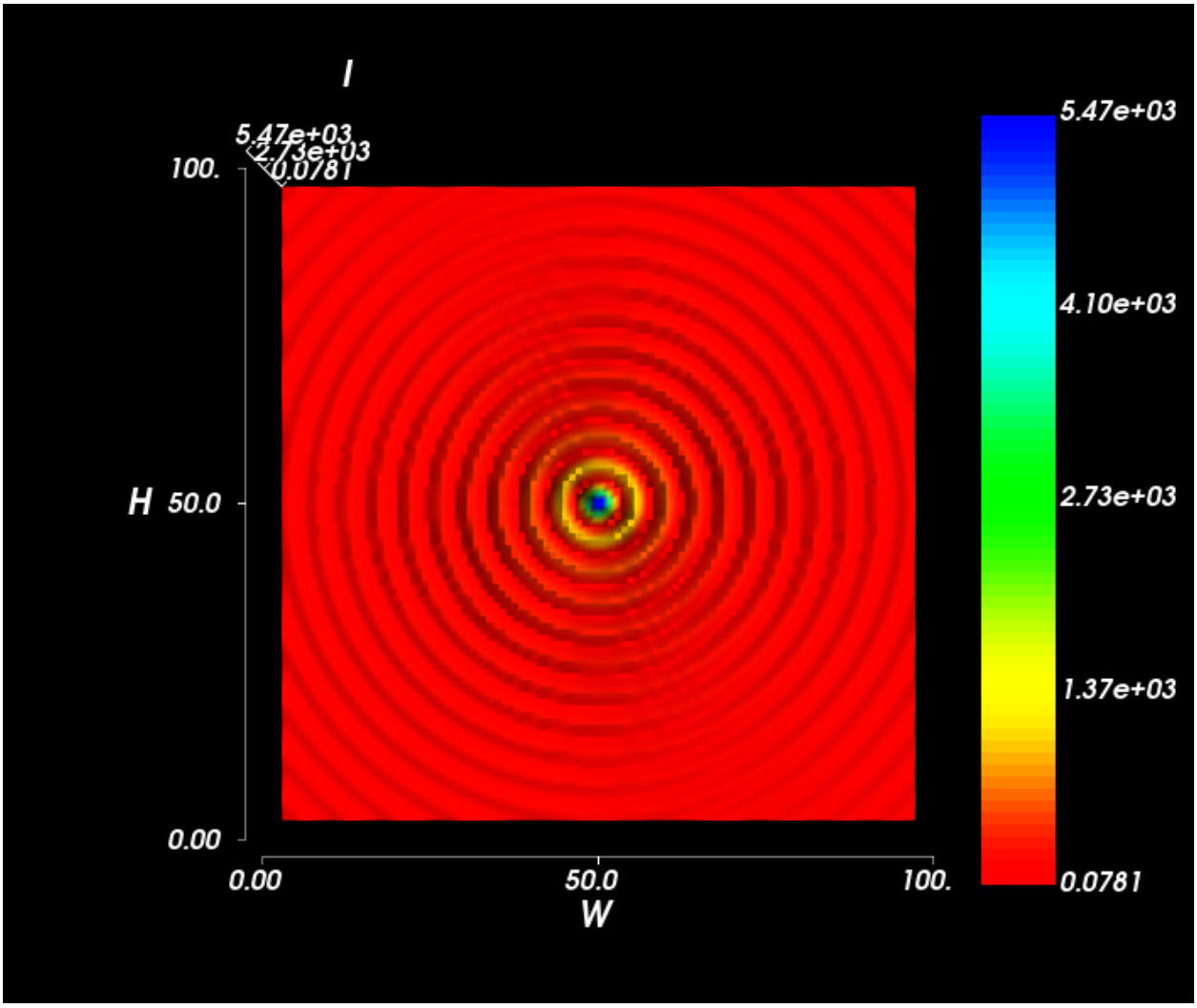
```
###Add some sensors
capture = GaussletCapturePlane(centre=(0,0,13),
                               direction=(0,0,1),
                               width=5.0,
                               height=5.0)

field = EFieldPlane(centre=(0,0,13),
                    direction=(0,0,1),
                    detector=capture,
                    align_detector=True,
                    size=100,
                    width=2,
                    height=2)

image = IntensityImageView(field_probe=field)
surf = IntensitySurface(field_probe=field)


model = RayTraceModel(optics=[axicon],
                      sources=[src],
                      probes=[capture,field],
                      results=[image,surf])
model.configure_traits()
```

Here's the model view.

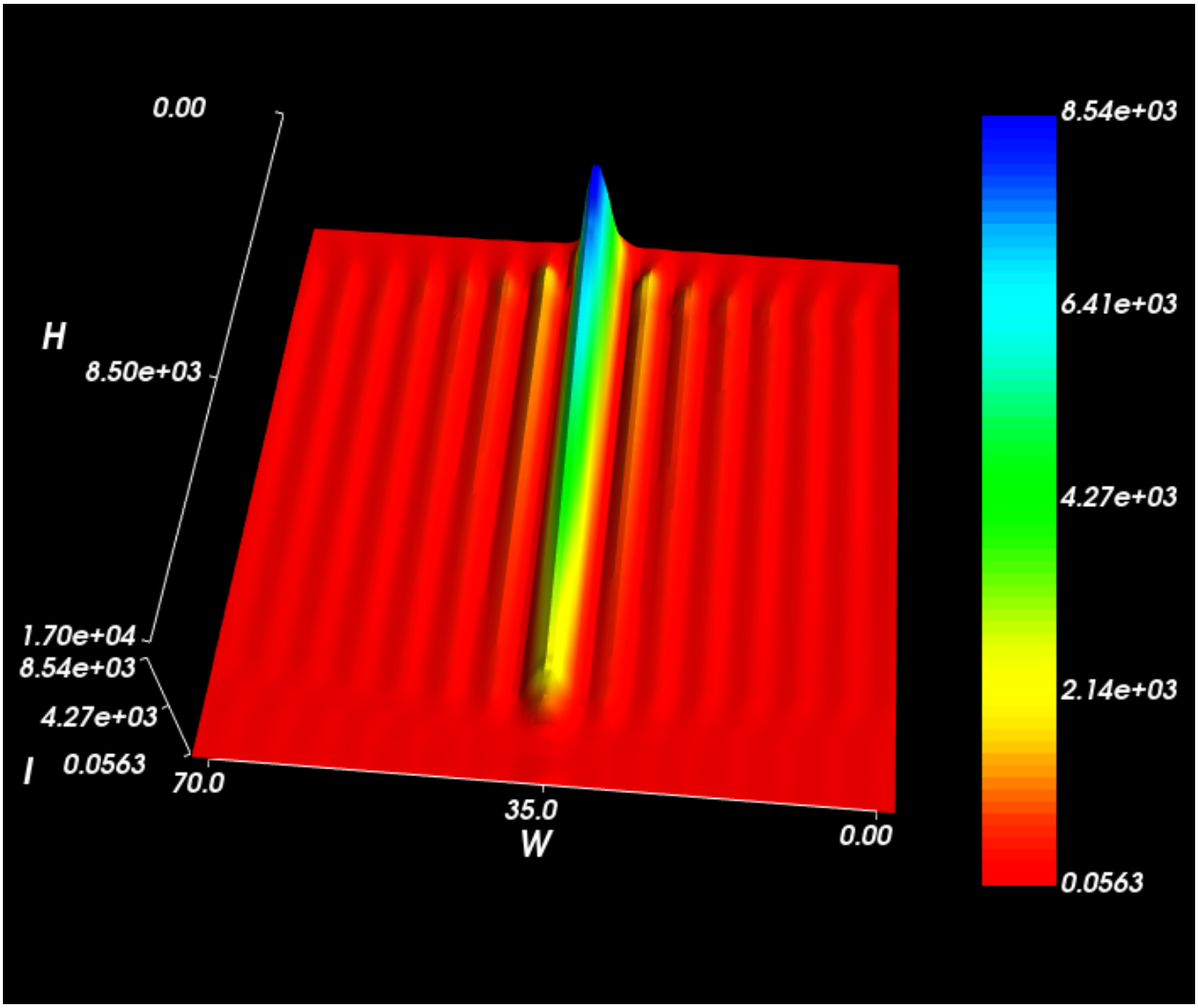


To get accurate results, turn up the resolution of the source object to about 30-40. Reduce the width of the EFieldPanel to ~0.1 to see the centre of the beam more clearly.

In the XY-plane, the characteristic Bessel rings are clear.



Looking along the Z-axis (the optical axis), the constant width of the central beam is observed.



14.2 Fresnel Diffraction From A BeamStop

We simply invert an aperture to create a beam-stop. Placing this the path of a collimated beam created interesting Fresnel-diffraction effects, including a bright spot at the centre of the blocked path, known as the Spot of Arago.

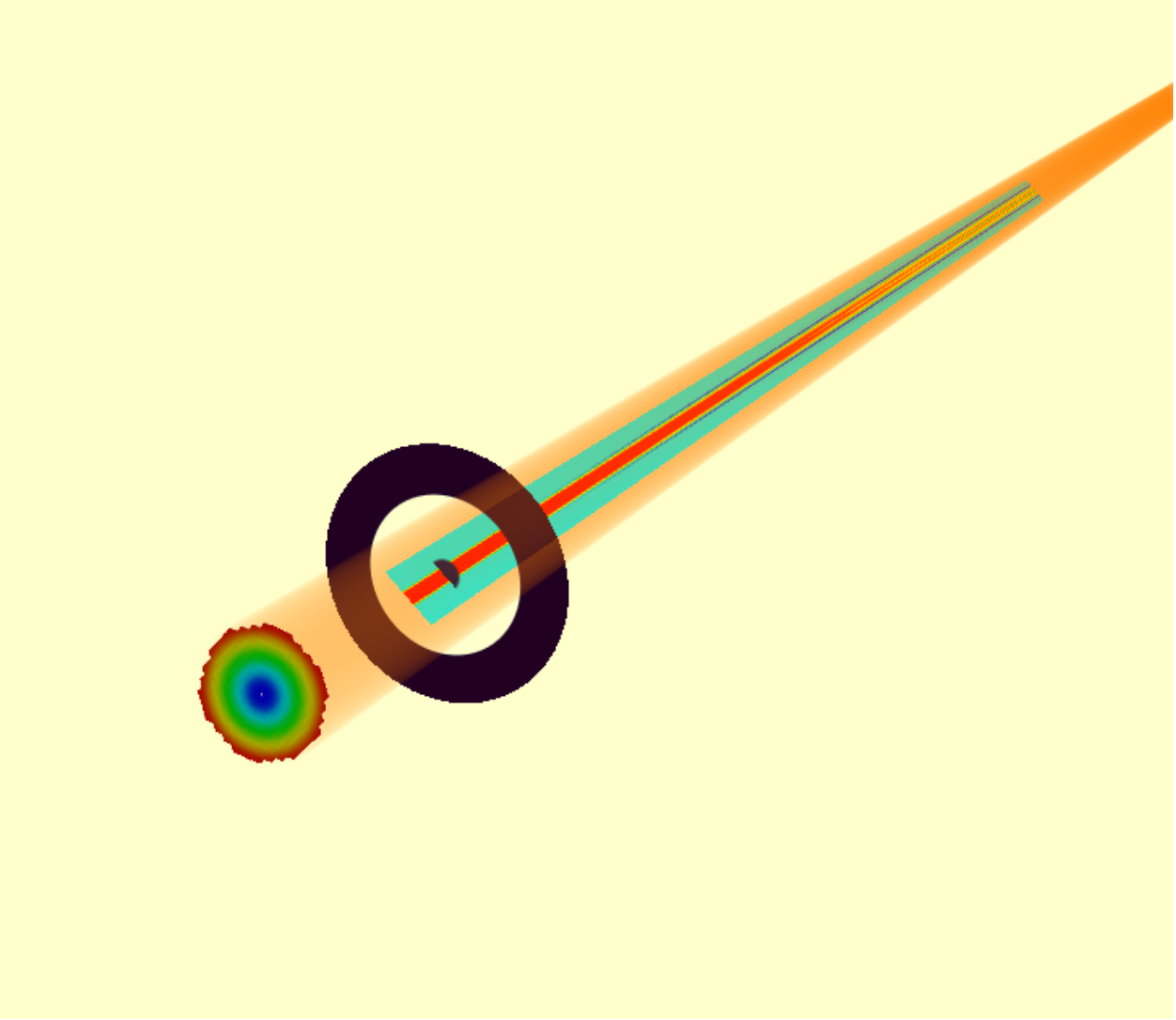
```
from raypier.tracer import RayTraceModel
from raypier.sources import HexagonalRayFieldSource, ConfocalRayFieldSource
from raypier.lenses import PlanoConvexLens
from raypier.apertures import CircularAperture
from raypier.fields import EFieldPlane
from raypier.constraints import BaseConstraint
from raypier.intensity_image import IntensityImageView
from raypier.intensity_surface import IntensitySurface

from traits.api import Range, on_trait_change
from traitsui.api import View, Item
```

(continues on next page)

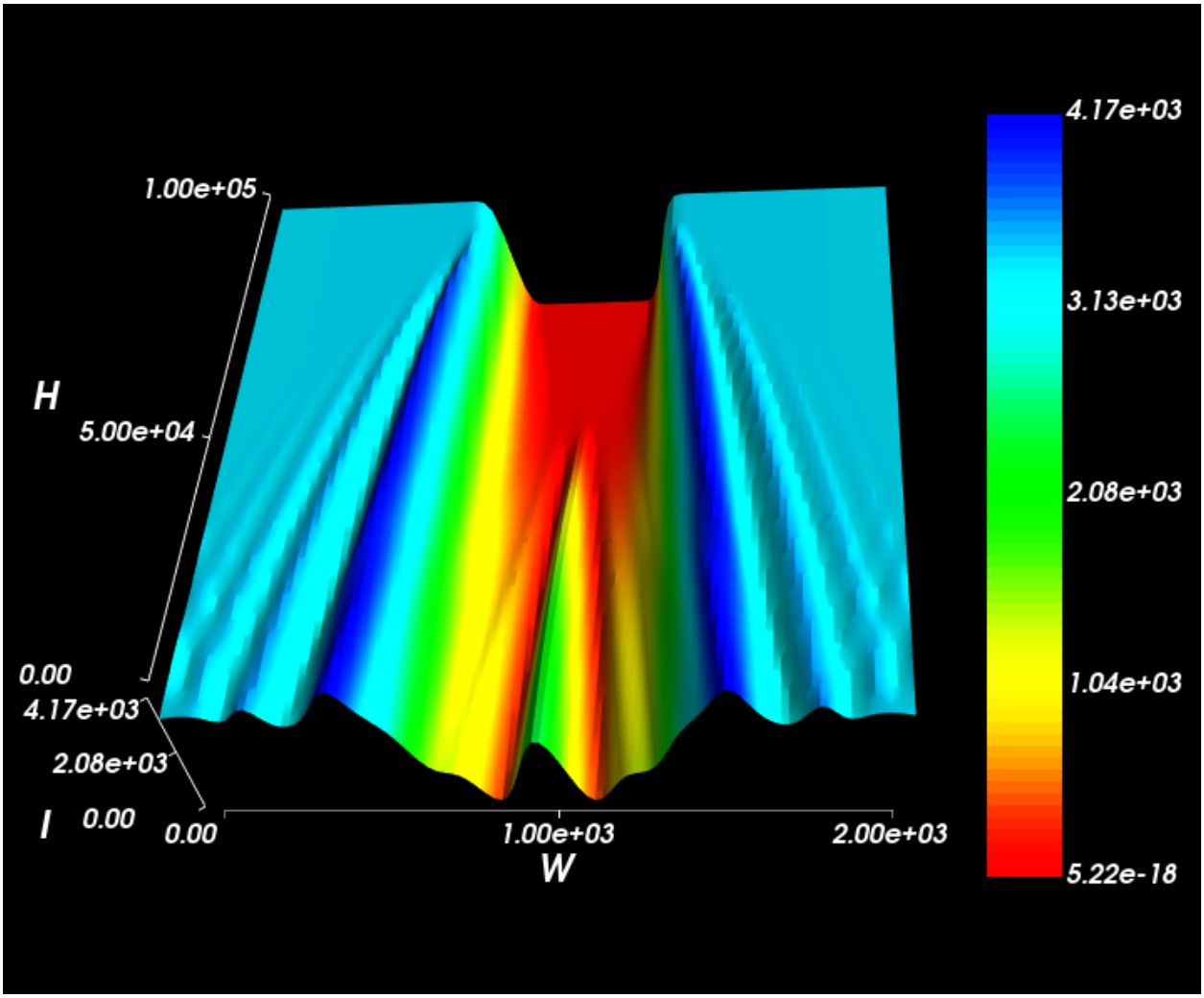
(continued from previous page)

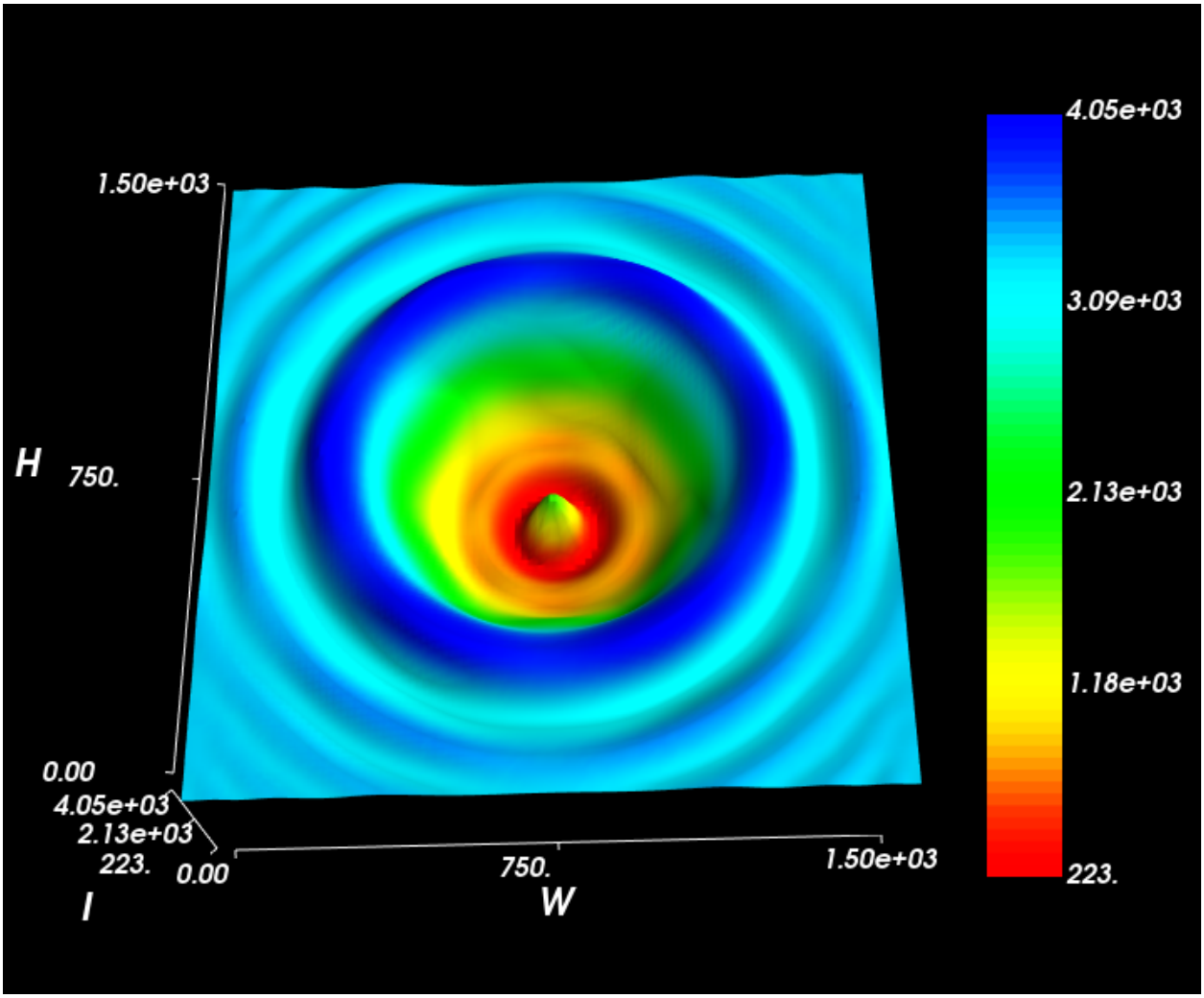
```
aperture = CircularAperture(centre=(0,0,10), direction=(0,0,1),
                            hole_diameter = 0.5, edge_width=0.001, invert=True)

src = HexagonalRayFieldSource(resolution=10.0, direction=(0,0,1),
                              radius=2.0,
                              wavelength=1.0)

probe = EFieldPlane(source=src,
                    centre=(0,0,50),
                    direction=(0,1,0),
                    exit_pupil_offset=100.,
                    width=2.0,
                    height=100.0,
                    size=100)

img = IntensityImageView(field_probe=probe)
surf = IntensitySurface(field_probe=probe)


class FocalPlane(BaseConstraint):
    z_pos = Range(50.0,130.0, 57.73)

    traits_view = View(Item("z_pos"))

    def __init__(self, *args, **kwds):
        super().__init__(*args, **kwds)
        self.on_change_z_pos()

    @on_trait_change("z_pos")
    def on_change_z_pos(self):
        probe.centre = (0,0,self.z_pos)



model = RayTraceModel(sources=[src], optics=[aperture],
                      probes=[probe], constraints=[FocalPlane()],
                      results=[img, surf])



model.configure_traits()
```

The model looks as follows:



However, the seem the detail in the Fresnel diffraction pattern, you need lots of Gausslets. Turn up the source resolution to at least 40.


4.17e+03
3.13e+03
2.08e+03
1.04e+03
5.22e-18
1.00e+05
H
5.00e+04
0.00
4.17e+03
2.08e+03
I
0.00
0.00
1.00e+03
2.00e+03
W



14.3 Michelson Interferometer

We can create a Michelson interferometer from a non-polarising beam-splitter and two mirrors. The second mirror has a spherical surface to generate a spherical wavefront at the detector-plane. The intensity distribution shows the classic Newton's Rings type pattern.

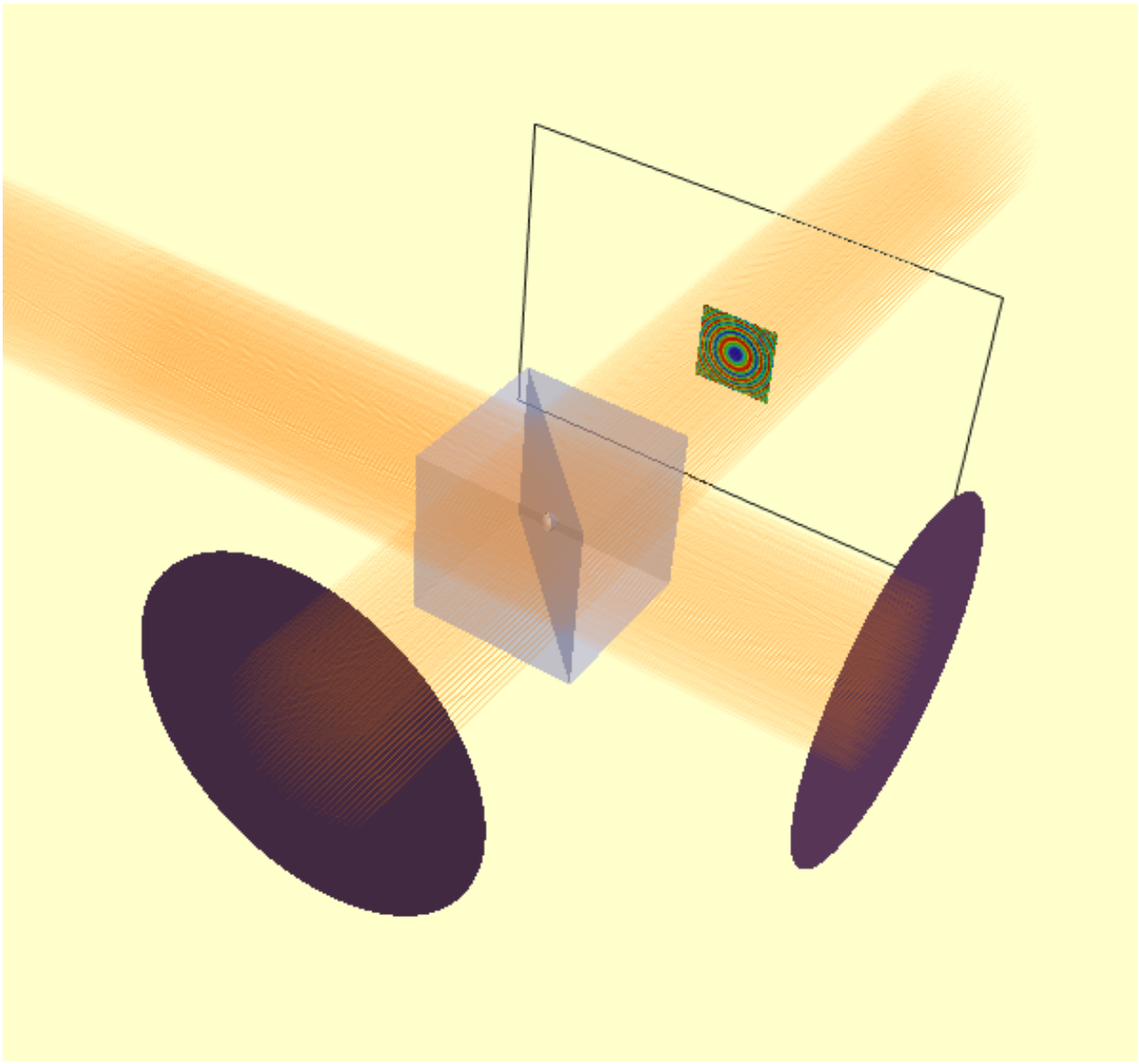
```
from raypier.api import RayTraceModel, UnpolarisingBeamsplitterCube,␣
↪CollimatedGaussletSource, CircleShape, GeneralLens,\
        SphericalFace, PlanarFace, OpticalMaterial, PolarisingBeamsplitterCube,␣
↪ParallelRaySource,\
        GaussletCapturePlane, EFieldPlane, IntensitySurface
from raypier.intensity_image import IntensityImageView



src=CollimatedGaussletSource(origin=(-30,0,0),
                             direction=(1,0,0),
```

(continues on next page)

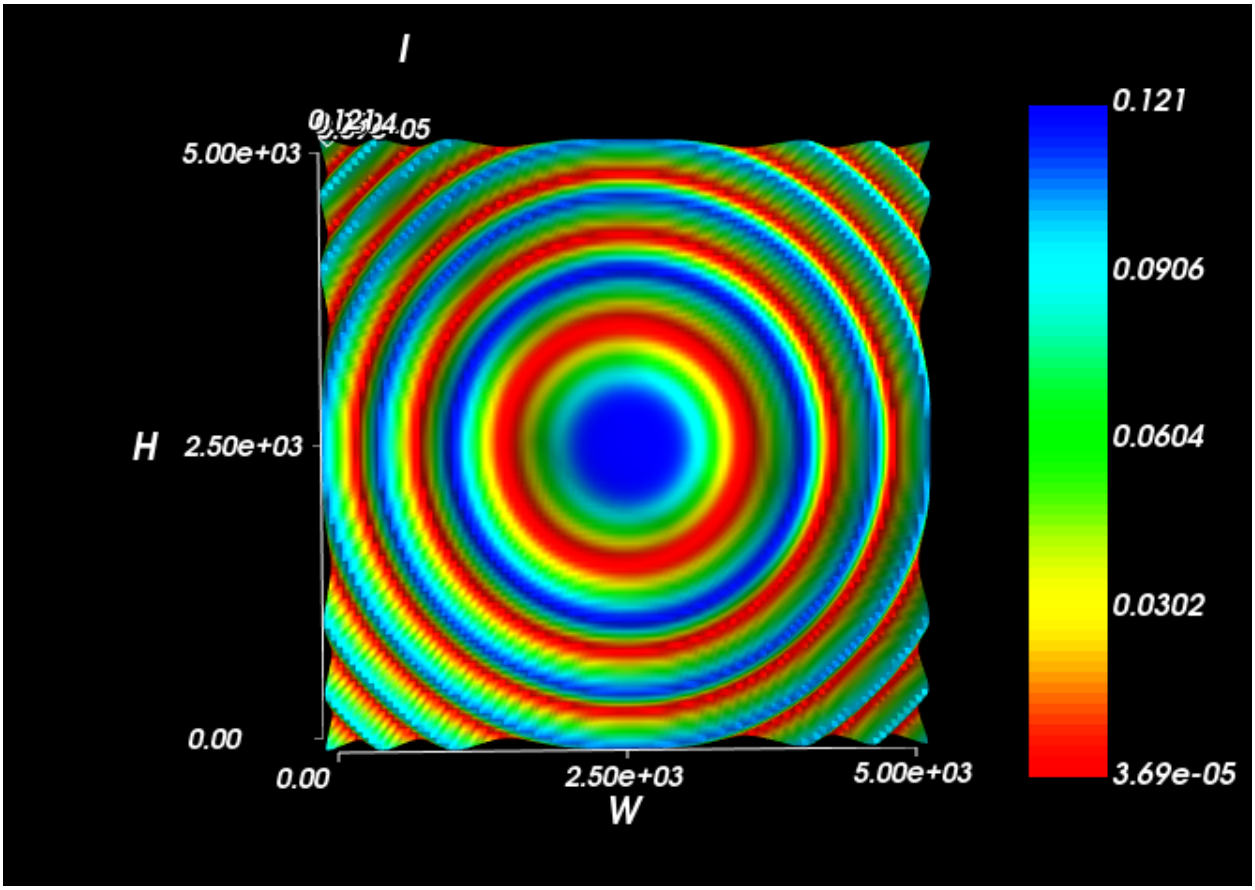
(continued from previous page)

```
                                radius=5.0,
                                beam_waist=10.0,
                                resolution=10.0,
                                E_vector=(0,1,0),
                                wavelength=1.0,
                                display="wires",
                                opacity=0.05,
                                max_ray_len=50.0)


bs = UnpolarisingBeamsplitterCube(centre=(0,0,0),
                                  size=10.0)

shape = CircleShape(radius=10.0)

f1 = PlanarFace(mirror=True)
f2 = SphericalFace(curvature=2000.0, mirror=True)

m1 = GeneralLens(name="Mirror 1",
                 shape=shape,
                 surfaces=[f1],
                 centre=(0,20,0),
                 direction=(0,-1,0))

m2 = GeneralLens(name="Mirror 2",
                 shape=shape,
                 surfaces=[f2],
                 centre=(20,0,0),
                 direction=(-1,0,0))

cap = GaussletCapturePlane(centre=(0,-20,0),
                           direction=(0,1,0))

field = EFieldPlane(centre=(0,-20,0),
                    direction=(0,1,0),
                    detector=cap,
                    align_detector=True,
                    width=5.0,
                    height=5.0,
                    size=100)

image = IntensityImageView(field_probe=field)
surf = IntensitySurface(field_probe=field)


model = RayTraceModel(optics=[bs, m1, m2],
                      sources=[src],
                      probes=[field, cap],
                      results=[image, surf])

model.configure_traits()
```

The rendered model looks as follows:



In this case, the E-field evaluation plane is large enough that we can see the intensity profile directly in the model. However, viewing it in a `IntensitySurfaceView` is more convenient.


I
5.00e+03
H
2.50e+03
0.00
0.00
2.50e+03
5.00e+03
W
0.121
0.0906
0.0604
0.0302
3.69e-05

CHAPTER

FIFTEEN

API REFERENCE

15.1 raypier.achromats

15.2 raypier.apertures

15.3 raypier.aspherics

15.4 raypier.bases

15.5 raypier.beamsplitters

15.6 raypier.beamstop

15.7 raypier.chirp_result

15.8 raypier.constraints

class raypier.constraints.**BaseConstraint**(**args*, ***kwds*)
Bases: raypier.has_queue.HasQueue

class raypier.constraints.**Constraint**(**args*, ***kwds*)
Bases: *raypier.constraints.BaseConstraint*

15.9 raypier.corner_cubes

15.10 raypier.decompositions

15.11 raypier.diffraction_gratings

15.12 raypier.dispertion

Created on 23 Nov 2018

author bryan

class raypier.dispersion.**FusedSilica**(*absorption=0.0*)
Bases: *raypier.core.cmaterials.BaseDispersionCurve*

A Dispersion curve for fused silica.

evaluate_n()
Calculates the complex refractive index for the given wavelengths.

Parameters **wavelen** (*double[:]*) – An array-like collection of wavelength, in microns.

Returns The refractive index for the given wavelength.

Return type complex128[:]

class raypier.dispersion.**NamedDispersionCurve**(*name=None*, *book=None*, *filename=None*, *absorption=0.0*)
Bases: *raypier.core.cmaterials.BaseDispersionCurve*

A Dispersion curve obtained from the materials database (http://refractiveindex.info).

evaluate_n()
Calculates the complex refractive index for the given wavelengths.

Parameters **wavelen** (*double[:]*) – An array-like collection of wavelength, in microns.

Returns The refractive index for the given wavelength.

Return type complex128[:]

class raypier.dispersion.**NondispersiveCurve**(*refractive_index=1.37*, *absorption=0.0*)
Bases: *raypier.core.cmaterials.BaseDispersionCurve*

A Dispersion curve for a non-dispersive material with a given refractive index and absorption.

evaluate_n()
Calculates the complex refractive index for the given wavelengths.

Parameters **wavelen** (*double[:]*) – An array-like collection of wavelength, in microns.

Returns The refractive index for the given wavelength.

Return type complex128[:]

15.13 raypier.distortions

15.14 raypier.editors

15.15 raypier.ellipsoids

15.16 raypier.faces

15.17 raypier.fields

15.18 raypier.gausslet_sources

15.19 raypier.general_optic

15.20 raypier.intensity_image

15.21 raypier.intensity_surface

15.22 raypier.lenses

15.23 raypier.materials

15.24 raypier.mirrors

15.25 raypier.parabolics

15.26 raypier.prisms

15.27 raypier.probes

15.28 raypier.results

15.29 raypier.shapes

15.30 raypier.sources

15.31 raypier.splines

15.32 raypier.step_export

15.33 raypier.tracer

15.34 raypier.utils

RequestData(*request*, *inInfo*, *outInfo*)
Overwritten by subclass to execute the algorithm.

RequestInformation(*request*, *inInfo*, *outInfo*)
Overwritten by subclass to provide meta-data to downstream pipeline.

class raypier.vtk_algorithms.**NumpyImageSource**(**args*, ***kwds*)
Bases: *raypier.vtk_algorithms.EmptyGridSource*

RequestData(*request*, *inInfo*, *outInfo*)
Overwritten by subclass to execute the algorithm.

class raypier.vtk_algorithms.**PythonAlgorithmBase**(**args*, ***kwds*)
Bases: tvtk.tvtk_classes.python_algorithm.PythonAlgorithm

FillInputPortInformation(*port*, *info*)
Sets the required input type to InputType.

FillOutputPortInformation(*port*, *info*)
Sets the default output type to OutputType.

class InternalAlgorithm
Bases: object

Internal class. Do not use.

FillInputPortInformation(*vtkself*, *port*, *info*)

FillOutputPortInformation(*vtkself*, *port*, *info*)

Initialize(*vtkself*)

ProcessRequest(*vtkself*, *request*, *inInfo*, *outInfo*)

ProcessRequest(*request*, *inInfo*, *outInfo*)
Splits a request to RequestXXX() methods.

RequestData(*request*, *inInfo*, *outInfo*)
Overwritten by subclass to execute the algorithm.

RequestDataObject(*request*, *inInfo*, *outInfo*)
Overwritten by subclass to manage data object creation. There is not need to overwrite this class if the output can be created based on the OutputType data member.

RequestInformation(*request*, *inInfo*, *outInfo*)
Overwritten by subclass to provide meta-data to downstream pipeline.

RequestUpdateExtent(*request*, *inInfo*, *outInfo*)
Overwritten by subclass to modify data request going to upstream pipeline.

get_input_data(*inInfo*, *i*, *j*)
Convenience method that returns an input data object given a vector of information objects and two indices.

get_output_data(*outInfo*, *i*)
Convenience method that returns an output data object given an information object and an index.

class raypier.vtk_algorithms.**VTKAlgorithm**
Bases: traits.has_traits.HasTraits

This is a superclass which can be derived to implement Python classes that work with vtkPythonAlgorithm. It implements Initialize(), ProcessRequest(), FillInputPortInformation() and FillOutputPortInformation().

Initialize() sets the input and output ports based on data members.

ProcessRequest() calls RequestXXX() methods to implement various pipeline passes.

FillInputPortInformation() and FillOutputPortInformation() set the input and output types based on data members.

FillInputPortInformation(*vtkself*, *port*, *info*)
Sets the required input type to InputType.

FillOutputPortInformation(*vtkself*, *port*, *info*)
Sets the default output type to OutputType.

GetInputData(*inInfo*, *i*, *j*)
Convenience method that returns an input data object given a vector of information objects and two indices.

GetOutputData(*outInfo*, *i*)
Convenience method that returns an output data object given an information object and an index.

Initialize(*vtkself*)
Sets up number of input and output ports based on NumberOfInputPorts and NumberOfOutputPorts.

ProcessRequest(*vtkself*, *request*, *inInfo*, *outInfo*)
Splits a request to RequestXXX() methods.

RequestData(*vtkself*, *request*, *inInfo*, *outInfo*)
Overwritten by subclass to execute the algorithm.

RequestDataObject(*vtkself*, *request*, *inInfo*, *outInfo*)
Overwritten by subclass to manage data object creation. There is not need to overwrite this class if the output can be created based on the OutputType data member.

RequestInformation(*vtkself*, *request*, *inInfo*, *outInfo*)
Overwritten by subclass to provide meta-data to downstream pipeline.

RequestUpdateExtent(*vtkself*, *request*, *inInfo*, *outInfo*)
Overwritten by subclass to modify data request going to upstream pipeline.

15.36 raypier.waveplates

15.37 raypier.windows

15.38 Raypier.Core

15.38.1 raypier.core.ctracer

Contains the core data-structures used in the tracing operation.

class raypier.core.ctracer.**Distortion**
Bases: object

A abstract base class to represents distortions on a face, a z-offset as a function of (x,y).

class raypier.core.ctracer.**FaceList**
Bases: object

A group of faces which share a transform

sync_transforms()
sets the transforms from the owner's VTKTransform

class raypier.core.ctracer.**GaussletBaseRayView**
Bases: *raypier.core.ctracer.RayArrayView*

get_ray_list()
Returns the contents of this RayCollection as a list of Rays

class raypier.core.ctracer.**GaussletCollection**
Bases: object

A list-like collection of ray_t objects.

The RayCollection is the primary data-structure used in the ray-tracing operation.

The RayCollection is of variable length, in that it can grow as individual rays are added to it. Internally, the memory allocated to the array of ray_t structures is re-allocated to increase its capacity.

add_gausslet()
Adds the given Ray instance to this collection

add_gausslet_list()
Adds the given list of Rays to this collection

clear_ray_list()
Empties this RayCollection (by setting the count to zero)

config_parabasal_rays()
Initialise the parabasal rays for a symmetric (i.e. circular) modes, using the base_ray data for wavelength, and the given beam waist 1/e^2 radius. 'working_dist' indicates the distance from the base_ray origin to the centre of the gaussian beam waist. Negative values imply a beam waist before the origin. 'radius' is given in mm.

copy_as_array()
Returns the contents of this RayCollection as a numpy array (the data is always copied).

from_array()
Creates a new RayCollection from the given numpy array. The array dtype should be a ctracer.ray_dtype. The data is copied into the RayCollection

get_gausslet_list()
Returns the contents of this RayCollection as a list of Rays

project_to_plane()
Project the rays in the collection onto the intersection with the given plane, defined by an origin point on the plane and the plane normal vector.

reset_length()
Sets the length of all rays in this RayCollection to Infinity

class raypier.core.ctracer.**InterfaceMaterial**
Bases: object

Abstract base class for objects describing the materials characterics of a Face

class raypier.core.ctracer.**Ray**
Bases: object

Ray - a wrapper around the ray_t C-structure.

The Ray extension class exists mainly as a convenience for manipulation of single or small numbers of rays from python. Large numbers of rays are more efficiently handled as either RayCollection objects, created in the tracing process, or as numpy arrays with the 'ray_dtype' dtype.

E1_amp
Complex amplitude of the electric field polarised parallel to the E_vection.

E2_amp
Complex amplitude of the electric field polarised perpendicular to the E_vection

E_vector
Unit vector, perpendicular to the ray direction, which gives the direction of E-field polarisation

accumulated_path
The total *optical* path up to the start-point of this ray.

amplitude
E field amplitude

direction
direction of the ray, normalised to a unit vector

ellipticity
Provide the ratio of power in the RH circular polarisation to the LH circular polarisation. A value of zero indicates linear polarisation. +1 indicate RH polarisation, -1 is LH polarisation. Or maybe the other way round.

end_face_idx
Index of the terminating face, in the global face list (created for each tracing operation)

jones_vector
Jones polarisation vector expressed as a tuple (alpha, beta) where alpha and beta are complex

length
The length of the ray. This is infinite in unterminated rays

major_minor_axes
Find the vector giving the major and minor axes of the polarisation ellipse. For fully circularly polarised light, the current E_vector will be returned

normal
normal vector for the face which created this ray

origin
Origin coordinates of the ray

parent_idx
Index of the parent ray in parent RayCollection

phase
An additional phase-factor for the ray. At present, this handles the 'grating phase' factor generated by diffraction gratings. All other material surfaces leave this unchanged

power
Optical power for the ray

project_E()
Rotate the E_vector onto the given axis, projecting E1_amp and E2_amp as necessary.

ray_type_id
Used to distinguish rays created by reflection vs transmission or some other mechanism. Transmission->0, Reflection->1

refractive_index
complex refractive index through which this ray is propagating

termination
the end-point of the ray (read only)

wavelength_idx
The wavelength of the ray in vacuum, in microns

class raypier.core.ctracer.**RayArrayView**
Bases: object

An abstract class to provide the API for ray_t member access from python / numpy

get_ray_list()
Returns the contents of this RayCollection as a list of Rays

class raypier.core.ctracer.**RayCollection**
Bases: *raypier.core.ctracer.RayArrayView*

A list-like collection of ray_t objects.

The RayCollection is the primary data-structure used in the ray-tracing operation.

The RayCollection is of variable length, in that it can grow as individual rays are added to it. Internally, the memory allocated to the array of ray_t structures is re-allocated to increase its capacity.

add_ray()
Adds the given Ray instance to this collection

add_ray_list()
Adds the given list of Rays to this collection

clear_ray_list()
Empties this RayCollection (by setting the count to zero)

copy_as_array()
Returns the contents of this RayCollection as a numpy array (the data is always copied).

from_array()
Creates a new RayCollection from the given numpy array. The array dtype should be a ctracer.ray_dtype. The data is copied into the RayCollection

get_ray_list()
Returns the contents of this RayCollection as a list of Rays

reset_length()
Sets the length of all rays in this RayCollection to Infinity

15.38.2 raypier.core.cmaterials

class raypier.core.cmaterials.**BaseDispersionCurve**
Bases: object

Base class for DispersionCurve objects. This extension class provides efficient C-API methods for evaluation of the refractive index for a given wavelength, based on one of various dispersion functions (e.g. Sellmeier curves) and a set of coefficients obtained from https://refractiveindex.info

Parameters

- **formula_id** (*int, readonly*) – Specifies the formula to use in evaluating this dispersion-curve
- **wavelength_min** (*float, readonly*) – Minimum wavelength that can be evaluated, in microns.
- **wavelength_max** (*float, readonly*) – Maximum wavelength that can be evaluated, in microns.

- **coefs** (*list[float], readonly*) – an array-like list of coefficients for the particular formula used.
- **absorption** (*float, readonly*) – The (constant) absorption coefficient used to evaluate the complex refractive index. Given in cm^-1.

evaluate_n()

Calculates the complex refractive index for the given wavelengths.

Parameters **wavelen** (*double[:]*) – An array-like collection of wavelength, in microns.

Returns The refractive index for the given wavelength.

Return type complex128[:]

class raypier.core.cmaterials.**CircularApertureMaterial**

Bases: *raypier.core.ctracer.InterfaceMaterial*

Similar to the TransparentMaterial i.e. it generates an outgoing ray with identical direction, polarisation etc. to the incoming ray. The material attenuates the E_field amplitudes according to the radial distance from the surface origin.

Parameters

- **outer_radius** (*double*) – Rays passing outside this radius are not intercepted.
- **radius** (*double*) – The radius of the inner hole.
- **edge_width** (*double*) – Rays passing through the inner hole are attenuated according to their proximity to the edge. The edge-width sets the width of the error-function (errf) used to calculate the attenuation.
- **invert** (*int*) – Inverts the aperture to make a field-stop.
- **origin** (*(double, double, double)*) – The centre point of the aperture.

class raypier.core.cmaterials.**CoatedDispersiveMaterial**

Bases: *raypier.core.ctracer.InterfaceMaterial*

A full implementation of the Fresnel Equations for a single-layer coated dielectric interface. The refractive index for each ray is obtained by look up of the provided dispersio-curve objects for the materials on each side of the interface.

Parameters

- **dispersion_inside** (raypier.core.cmaterials.BaseDispersionCurve) – The dispersion curve for the inside side of the interface
- **dispersion_outside** (raypier.core.cmaterials.BaseDispersionCurve) – The dispersion curve for the "outside" of the interface
- **dispersion_coating** (raypier.core.cmaterials.BaseDispersionCurve) – The dispersion curve for the coating material
- **coating_thickness** (*double*) – The coating thickness, in microns
- **reflection_threshold** (*double*) – Sets the amplitude threshold for generating a reflected ray.
- **transmission_threshold** (*double*) – Sets the amplitude threshold for generating a transmitted ray.

class raypier.core.cmaterials.**DielectricMaterial**

Bases: *raypier.core.ctracer.InterfaceMaterial*

Simulates Fresnel reflection and refraction at a normal dielectric interface.

The surface normal is assumed to be pointing "out" of the material.

Parameters

- **n_inside** (*complex*) – The refractive index on the inside of the material interface
- **n_outside** (*complex*) – The refractive index on the outside of the material interface

class raypier.core.cmaterials.**DiffractionGratingMaterial**
Bases: *raypier.core.ctracer.InterfaceMaterial*

A specialised material modelling the behaviour of a reflective diffraction grating.

Parameters

- **lines_per_mm** (*double*) – The grating line density
- **order** (*int*) – The order-of-diffraction
- **efficiency** (*double*) – A value from 0.0 to 1.0 giving the reflection efficiency of the grating
- **origin** (*(double, double, double)*) – A vector (3-tuple) indicating the origin of the grating. Unimportant in most cases. This affects the "grating phase" which is unobservable in most situations.

class raypier.core.cmaterials.**FullDielectricMaterial**
Bases: *raypier.core.cmaterials.DielectricMaterial*

Model for dielectric using full Fresnel equations to give true phase and amplitude response

Parameters

- **reflection_threshold** (*double*) – Sets the amplitude threshold for generating a reflected ray.
- **transmission_threshold** (*double*) – Sets the amplitude threshold for generating a transmitted ray.

class raypier.core.cmaterials.**LinearPolarisingMaterial**
Bases: *raypier.core.ctracer.InterfaceMaterial*

Simulates a perfect polarising beam splitter. P-polarisation is 100% transmitted while S- is reflected

class raypier.core.cmaterials.**OpaqueMaterial**
Bases: *raypier.core.ctracer.InterfaceMaterial*

A perfect absorber i.e. it generates no rays

class raypier.core.cmaterials.**PECMaterial**
Bases: *raypier.core.ctracer.InterfaceMaterial*

Simulates a Perfect Electrical Conductor. I.e. incident rays are reflected with 100% reflectivity.

class raypier.core.cmaterials.**PartiallyReflectiveMaterial**
Bases: *raypier.core.ctracer.InterfaceMaterial*

A simple materials with a fixed reflectivity.

Parameters **reflectivity** (*double*) – The material power-reflectivity given as a value from 0.0. to 1.0

class raypier.core.cmaterials.**RectangularApertureMaterial**
Bases: *raypier.core.ctracer.InterfaceMaterial*

A rectangular aperture.

Parameters

- **outer_width** (*double*) –
- **outer_height** (*double*) –
- **width** (*double*) –
- **height** (*double*) –
- **edge_width** (*double*) –
- **invert** (*int*) –
- **origin** (*(double, double, double)*) –

class raypier.core.cmaterials.**ResampleGaussletMaterial**
Bases: *raypier.core.ctracer.InterfaceMaterial*

This is a special pseudo-material which generates new rays not by the normal process of refraction or reflection of an incoming ray, but by computing a new set of Gausslets by computing the E-field at a set of grid points and launching new Gausslets from these points.

The material needs the set of new launch-positions to be given up front (i.e. before tracing).

Parameters

- **size** (*int*) – The size of the new GaussletCollection to allocate up front.
- **eval_func** (*callable*) – A callable taking a GaussletCollection as its single argument. The new outgoing rays will be returned by this callable.

class raypier.core.cmaterials.**SingleLayerCoatedMaterial**
Bases: *raypier.core.cmaterials.FullDielectricMaterial*

A material with a single-layer dielectric coating at the interface.

Parameters

- **n_coating** (*complex*) – The complex refractive index for the coating.
- **thickness** (*double*) – The thickness of the coating in microns

class raypier.core.cmaterials.**TransparentMaterial**
Bases: *raypier.core.ctracer.InterfaceMaterial*

A perfect transmitter i.e. it generates an outgoing ray with identical direction, polarisation etc. to the incoming ray. It does project the polarisation vectors to it's S- and P-directions, however.

class raypier.core.cmaterials.**WaveplateMaterial**
Bases: *raypier.core.ctracer.InterfaceMaterial*

An idealised optical retarder.

Parameters

- **retardance** (*double*) – The optical retardance, given in terms of numbers-of-wavelengths.
- **fast_axis** (*(double, double, double)*) – A vector giving the "fast" polarisation axis

apply_retardance()
Applies the retardance to the given Ray object.

Parameters **r** (raypier.core.ctracer.Ray) – an input Ray object.

Returns a new Ray instance

Return type *raypier.core.ctracer.Ray*

15.38.3 raypier.core.cfaces

Cython module for Face definitions

class raypier.core.cfaces.**AsphericFace**
Bases: raypier.core.cfaces.ShapedFace

This is the general aspheric lens surface formula.

curvature = radius of curvature

class raypier.core.cfaces.**AxiconFace**
Bases: raypier.core.cfaces.ShapedFace

While technically, we can use the conic surface to generate a cone, it requires setting some parameters to infinity which is often inaccurate to compute.

The gradient is the slope of the sides, dz/dr

class raypier.core.cfaces.**ConicRevolutionFace**
Bases: raypier.core.cfaces.ShapedFace

This is surface of revolution formed from a conic section. Spherical and ellipsoidal faces are a special case of this.

curvature = radius of curvature

class raypier.core.cfaces.**DistortionFace**
Bases: raypier.core.cfaces.ShapedFace

This class wraps another ShapedFace object, and applies a small distortion to it's surface geometry. The distortion is given by an instance of a Distortion subclass

15.38.4 raypier.core.cfields

Core C-level algorimths for computing the electric field from a set of traced rays.

15.38.5 raypier.core.cshapes

15.38.6 raypier.core.cdistortions

class raypier.core.cdistortions.**SimpleTestZernikeJ7**
Bases: *raypier.core.ctracer.Distortion*

Implement one low-order Zernike poly for testing purposes.

class raypier.core.cdistortions.**ZernikeDistortion**
Bases: *raypier.core.ctracer.Distortion*

15.38.7 raypier.core.tracer

Core python classes for tracing operations.

15.38.8 raypier.core.fields

Functions relating to the evaluation of the optical E-field by summation of General Astigmatic Gaussian Beams

class raypier.core.fields.**EFieldSummation**(*gausslet_collection*, *wavelengths=None*, *blending=1.0*)

Bases: object

For situations where you wish to evaluate the E-field from a set of Gausslets with different sets of evaluation points, this class provides a small optimisation by performing the maths to convert ray-intercepts to Gaussian mode parameters up front.

evaluate(*points*)

Called to calculate the E-field for the given points.

raypier.core.fields.**ExtractGamma**(*gausslet_collection*, *blending=1.0*)

Used in Testing

raypier.core.fields.**Gamma**(*z*, *A*, *B*, *C*)

Used in testing

raypier.core.fields.**eval_Efield_from_gausslets**(*gausslet_collection*, *points*, *wavelengths=None*, *blending=1.0*, ***kwds*)

Calculates the vector E-field is each of the points given. The returned array of field-vectors will have the same length as *points* and has *numpy.complex128* dtype.

Parameters

- **gc** (GaussletCollection) – The set of Gausslets for which the field should be calculated
- **points** (*ndarray[N,3]*) – An array of shape (N,3) giving the points at which the field will be evaluated.
- **wavelengths** (*ndarray[]*) – A 1d array containing the wavelengths to be used for the field calculation, overriding the wavelengths data contained by the GaussletCollection object.
- **blending** (*float*) – The 1/width of each Gaussian mode at the evaluation points. A value of unity (the default), means the parabasal rays are determined to be the 1/e point in the field amplitude.

raypier.core.fields.**evaluate_modes**(*neighbour_x*, *neighbour_y*, *dx*, *dy*, *blending=1.0*)

For each ray in rays, use its nearest neighbours to compute the best fit for the Astigmatic Gaussian Beam parameters.

rays - an array of ray_t with shape (N,) x,y,dx,dy - array of shape (N,6)

For N rays, return a Nx3 complex array of coeffs

raypier.core.fields.**evaluate_neighbours**(*rays*, *neighbours_idx*)

For each of a rays neighbours, we need to project the neighbour back onto the plane containing the main rays origin, to obtain the (x,y) cordinate for the neighbour ray, relative to the main ray origin

rays - a ray_t array of length N neighbours_idx - a array on ints of shape (N,6)

returns - a 5-tuple (rays, x,y,dx,dy) where x,y are N*6 arrays for the coordinate of each neighbour. dx and dy represent the change in direction of the neighbouring rays (i.e. curvature of the wavefront). The returns rays are the subset of the input rays with 6 neighbours (i.e. edge-rays are dropped).

raypier.core.fields.**project_to_sphere**(*rays*, *centre=(0, 0, 0)*, *radius=10.0*)
project the given set of rays back to their intercept with a sphere at the given centre and radius.

rays - an array of ray_t dtype

15.38.9 raypier.core.gausslets

15.38.10 raypier.core.find_focus

Utility module

raypier.core.find_focus.**find_focus**(*ray_origins*, *ray_directions*, *weights=None*)
givens Nx3 arrays for ray origin and directions, return the 3D point of closest intersection, found in the least-squares sense.

See https://math.stackexchange.com/q/1762491

15.38.11 raypier.core.utils

raypier.core.utils.**Convert_to_SP**(*input_v*, *normal_v*, *E1_vector*, *E1_amp*, *E2_amp*)
All inputs are 2D arrays

raypier.core.utils.**dotprod**(*a*, *b*)
dot-product along last axis

raypier.core.utils.**normaliseVector**(*a*)
normalise a (3,) vector or a (n,3) array of vectors

15.38.12 raypier.core.unwrap2d

Functions for performing 2D phase unwrapping.

raypier.core.unwrap2d.**unwrap2d**(*phase_array*, *anchor=(0, 0)*)
A basic phase unwrapper using numpy.unwrap and the Itoh method.

phase_array - a (N,M) shaped array of values in the range -pi .. +pi

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Z